Apache CouchDB®

Release 3.3.3

Mar 17, 2024

USER GUIDES

1 Introduction **1**
1.1 Technical Overview . . . 1
1.1.1 Document Storage . . . 1
1.1.2 ACID Properties . . . 2
1.1.3 Compaction . . . 2
1.1.4 Views . . . 2
1.1.5 Security and Validation . . . 4
1.1.6 Distributed Updates and Replication . . . 4
1.1.7 Implementation . . . 5
1.2 Why CouchDB? . . . 6
1.2.1 Relax . . . 6
1.2.2 A Different Way to Model Your Data . . . 7
1.2.3 A Better Fit for Common Applications . . . 7
1.2.4 Building Blocks for Larger Systems . . . 8
1.2.5 CouchDB Replication . . . 9
1.2.6 Local Data Is King . . . 10
1.2.7 Wrapping Up . . . 10
1.3 Eventual Consistency . . . 10
1.3.1 Working with the Grain . . . 10
1.3.2 The CAP Theorem . . . 11
1.3.3 Local Consistency . . . 12
1.3.4 Validation . . . 13
1.3.5 Distributed Consistency . . . 14
1.3.6 Incremental Replication . . . 14
1.3.7 Case Study . . . 15
1.3.8 Wrapping Up . . . 17
1.4 cURL: Your Command Line Friend . . . 17
1.5 Security . . . 19
1.5.1 Authentication . . . 19
1.5.2 Authentication Database . . . 22
1.5.3 Authorization . . . 25
1.6 Getting Started . . . 26
1.6.1 All Systems Are Go! . . . 26
1.6.2 Welcome to Fauxton . . . 28
1.6.3 Your First Database and Document . . . 29
1.6.4 Running a Mango Query . . . 29
1.6.5 Triggering Replication . . . 31
1.6.6 Wrapping Up . . . 31
1.7 The Core API . . . 31
1.7.1 Server . . . 32
1.7.2 Databases . . . 33
1.7.3 Documents . . . 36
1.7.4 Replication . . . 40
1.7.5 Wrapping Up . . . 42

2 Replication **43**
2.1 Introduction to Replication . . . 43
2.1.1 Transient and Persistent Replication . . . 43
2.1.2 Triggering, Stopping and Monitoring Replications . . . 43
2.1.3 Replication Procedure . . . 44
2.1.4 Master - Master replication . . . 44
2.1.5 Controlling which Documents to Replicate . . . 44
2.1.6 Migrating Data to Clients . . . 45
2.2 Replicator Database . . . 45
2.2.1 Basics . . . 45
2.2.2 Documents describing the same replication . . . 49
2.2.3 Replication Scheduler . . . 49
2.2.4 Replication states . . . 50
2.2.5 Compatibility Mode . . . 51
2.2.6 Canceling replications . . . 52
2.2.7 Server restart . . . 52
2.2.8 Clustering . . . 52
2.2.9 Additional Replicator Databases . . . 52
2.2.10 Fair Share Job Scheduling . . . 53
2.2.11 Replicating the replicator database . . . 54
2.2.12 Delegations . . . 54
2.2.13 Selector Objects . . . 55
2.2.14 Specifying Usernames and Passwords . . . 55
2.2.15 Replicate Winning Revisions Only . . . 56
2.3 Replication and conflict model . . . 57
2.3.1 CouchDB replication . . . 57
2.3.2 Conflict avoidance . . . 58
2.3.3 Revision tree . . . 59
2.3.4 Working with conflicting documents . . . 59
2.3.5 Multiple document API . . . 60
2.3.6 View map functions . . . 61
2.3.7 Merging and revision history . . . 64
2.3.8 Comparison with other replicating data stores . . . 65
2.4 CouchDB Replication Protocol . . . 68
2.4.1 Preface . . . 68
2.4.2 Replication Protocol Algorithm . . . 70
2.4.3 Protocol Robustness . . . 95
2.4.4 Error Responses . . . 95
2.4.5 Optimisations . . . 97
2.4.6 API Reference . . . 97
2.4.7 Reference . . . 98

3 Design Documents **99**
3.1 Design Documents . . . 99
3.1.1 Creation and Structure . . . 99
3.1.2 View Functions . . . 100
3.1.3 Show Functions . . . 104
3.1.4 List Functions . . . 106
3.1.5 Update Functions . . . 107
3.1.6 Filter Functions . . . 107
3.1.7 Validate Document Update Functions . . . 110
3.2 Guide to Views . . . 113
3.2.1 Introduction to Views . . . 113
3.2.2 Views Collation . . . 123
3.2.3 Joins With Views . . . 127
3.2.4 View Cookbook for SQL Jockeys . . . 134
3.2.5 Pagination Recipe . . . 141
3.3 Search . . . 145

3.3.1 Index functions . . . 145
3.3.2 Analyzers . . . 148
3.3.3 Queries . . . 151
3.3.4 Query syntax . . . 153
3.3.5 Geographical searches . . . 156
3.3.6 Highlighting search terms . . . 158
3.4 Nouveau . . . 159
3.4.1 Index functions . . . 160
3.4.2 Analyzers . . . 162
3.4.3 Queries . . . 164
3.4.4 Query syntax . . . 166

4 Best Practices **171**
4.1 Document Design Considerations . . . 171
4.1.1 Don't rely on CouchDB's auto-UUID generation . . . 171
4.1.2 Alternatives to auto-incrementing sequences . . . 171
4.1.3 Pre-aggregating your data . . . 172
4.1.4 Designing an application to work with replication . . . 172
4.1.5 Adding client-side security with a translucent database . . . 175
4.2 Document submission using HTML Forms . . . 176
4.2.1 The HTML form . . . 176
4.2.2 The update function . . . 176
4.2.3 Example output . . . 177
4.3 Using an ISO Formatted Date for Document IDs . . . 178
4.4 JavaScript development tips . . . 178
4.5 View recommendations . . . 179
4.5.1 Deploying a view change in a live environment . . . 179
4.6 Reverse Proxies . . . 179
4.6.1 Reverse proxying with HAProxy . . . 179
4.6.2 Reverse proxying with nginx . . . 180
4.6.3 Reverse Proxying with Caddy 2 . . . 182
4.6.4 Reverse Proxying with Apache HTTP Server . . . 183

5 Installation **185**
5.1 Installation on Unix-like systems . . . 185
5.1.1 Installation using the Apache CouchDB convenience binary packages . . . 185
5.1.2 Installation from source . . . 187
5.1.3 Dependencies . . . 187
5.1.4 Installing . . . 189
5.1.5 User Registration and Security . . . 190
5.1.6 First Run . . . 190
5.1.7 Running as a Daemon . . . 191
5.2 Installation on Windows . . . 192
5.2.1 Installation from binaries . . . 192
5.2.2 Installation from sources . . . 193
5.3 Installation on macOS . . . 193
5.3.1 Installation using the Apache CouchDB native application . . . 193
5.3.2 Installation with Homebrew . . . 194
5.3.3 Installation from source . . . 194
5.4 Installation on FreeBSD . . . 194
5.4.1 Installation . . . 194
5.4.2 Service Configuration . . . 194
5.4.3 Post Install . . . 195
5.5 Installation via Docker . . . 195
5.6 Installation via Snap . . . 195
5.7 Installation on Kubernetes . . . 197
5.8 Search Plugin Installation . . . 197
5.8.1 Installation of Binary Packages . . . 197

5.8.2 Chef . . . 198
5.8.3 Kubernetes . . . 198
5.8.4 Additional Details . . . 198
5.9 Nouveau Server Installation . . . 198
5.9.1 Enable Nouveau . . . 199
5.9.2 Installation of Binary Packages . . . 199
5.10 Upgrading from prior CouchDB releases . . . 199
5.10.1 Important Notes . . . 199
5.10.2 Upgrading from CouchDB 2.x . . . 199
5.10.3 Upgrading from CouchDB 1.x . . . 200
5.11 Troubleshooting an Installation . . . 201
5.11.1 First Install . . . 201
5.11.2 Quick Build . . . 202
5.11.3 Upgrading . . . 203
5.11.4 Runtime Errors . . . 203
5.11.5 macOS Known Issues . . . 205

6 Setup **207**
6.1 Single Node Setup . . . 207
6.2 Cluster Set Up . . . 207
6.2.1 Ports and Firewalls . . . 208
6.2.2 Configure and Test the Communication with Erlang . . . 208
6.2.3 Preparing CouchDB nodes to be joined into a cluster . . . 210
6.2.4 The Cluster Setup Wizard . . . 211
6.2.5 The Cluster Setup API . . . 211

7 Configuration **213**
7.1 Introduction To Configuring . . . 213
7.1.1 Configuration files . . . 213
7.1.2 Parameter names and values . . . 214
7.1.3 Setting parameters via the configuration file . . . 214
7.1.4 Setting parameters via the HTTP API . . . 214
7.1.5 Configuring the local node . . . 215
7.2 Base Configuration . . . 215
7.2.1 Base CouchDB Options . . . 215
7.3 Configuring Clustering . . . 218
7.3.1 Cluster Options . . . 218
7.3.2 RPC Performance Tuning . . . 219
7.4 Database Per User . . . 220
7.4.1 Database Per User Options . . . 220
7.5 Disk Monitor Configuration . . . 221
7.5.1 Disk Monitor Options . . . 221
7.6 CouchDB HTTP Server . . . 222
7.6.1 HTTP Server Options . . . 222
7.6.2 HTTPS (SSL/TLS) Options . . . 226
7.6.3 Cross-Origin Resource Sharing . . . 228
7.6.4 Virtual Hosts . . . 230
7.6.5 X-Frame-Options . . . 231
7.7 Authentication and Authorization . . . 231
7.7.1 Server Administrators . . . 231
7.7.2 Authentication Configuration . . . 233
7.8 Compaction . . . 239
7.8.1 Database Compaction Options . . . 239
7.8.2 View Compaction Options . . . 239
7.8.3 Compaction Daemon . . . 240
7.9 Background Indexing . . . 241
7.10 IO Queue . . . 242
7.10.1 Recommendations . . . 243

7.11 Logging . . . 243
7.11.1 Logging options . . . 243
7.12 Replicator . . . 245
7.12.1 Replicator Database Configuration . . . 245
7.12.2 Fair Share Replicator Share Allocation . . . 250
7.13 Query Servers . . . 250
7.13.1 Query Servers Definition . . . 250
7.13.2 Query Servers Configuration . . . 251
7.13.3 Native Erlang Query Server . . . 252
7.13.4 Search . . . 252
7.13.5 Nouveau . . . 253
7.13.6 Mango . . . 253
7.14 Miscellaneous Parameters . . . 254
7.14.1 Configuration of Attachment Storage . . . 254
7.14.2 Statistic Calculation . . . 254
7.14.3 UUIDs Configuration . . . 254
7.14.4 Vendor information . . . 256
7.14.5 Content-Security-Policy . . . 257
7.14.6 Configuration of Database Purge . . . 258
7.14.7 Configuration of Prometheus Endpoint . . . 258
7.15 Resharding . . . 259
7.15.1 Resharding Configuration . . . 259

8 Cluster Management **261**
8.1 Theory . . . 261
8.2 Node Management . . . 262
8.2.1 Adding a node . . . 262
8.2.2 Removing a node . . . 263
8.3 Database Management . . . 263
8.3.1 Creating a database . . . 263
8.3.2 Deleting a database . . . 263
8.3.3 Placing a database on specific nodes . . . 263
8.4 Shard Management . . . 264
8.4.1 Introduction . . . 264
8.4.2 Examining database shards . . . 265
8.4.3 Moving a shard . . . 267
8.4.4 Specifying database placement . . . 272
8.4.5 Splitting Shards . . . 273
8.4.6 Stopping Resharding Jobs . . . 275
8.4.7 Merging Shards . . . 275
8.5 Clustered Purge . . . 276
8.5.1 Internal Structures . . . 276
8.5.2 Compaction of Purges . . . 276
8.5.3 Local Purge Checkpoint Documents . . . 277
8.5.4 Internal Replication . . . 277
8.5.5 Indexes . . . 278
8.5.6 Config Settings . . . 278
8.6 TLS Erlang Distribution . . . 279
8.6.1 Generate Certificate . . . 279
8.6.2 Config Settings . . . 280
8.6.3 Connect to Remsh . . . 280
8.7 Troubleshooting CouchDB 3 with WeatherReport . . . 281
8.7.1 Overview . . . 281
8.7.2 Usage . . . 281

9 Maintenance **283**
9.1 Compaction . . . 283
9.1.1 Automatic Compaction . . . 283

9.1.2 Manual Database Compaction . . . 286
9.1.3 Manual View Compaction . . . 287
9.2 Performance . . . 288
9.2.1 Disk I/O . . . 288
9.2.2 System Resource Limits . . . 288
9.2.3 Network . . . 290
9.2.4 CouchDB . . . 291
9.2.5 Views . . . 291
9.3 Backing up CouchDB . . . 292
9.3.1 Database Backups . . . 292
9.3.2 Configuration Backups . . . 293
9.3.3 Log Backups . . . 293

10 Fauxton **295**
10.1 Fauxton Setup . . . 295
10.1.1 Fauxton Visual Guide . . . 295
10.1.2 Development Server . . . 295
10.1.3 Understanding Fauxton Code layout . . . 295

11 Experimental Features **297**
11.1 Content-Security-Policy (CSP) Header Support for /_utils (Fauxton) . . . 297
11.2 Nouveau Server (new Apache Lucene integration) . . . 297

12 API Reference **299**
12.1 API Basics . . . 299
12.1.1 Request Format and Responses . . . 299
12.1.2 HTTP Headers . . . 300
12.1.3 JSON Basics . . . 302
12.1.4 HTTP Status Codes . . . 305
12.2 Server . . . 307
12.2.1 `/` . . . 307
12.2.2 `/_active_tasks` . . . 308
12.2.3 `/_all_dbs` . . . 310
12.2.4 `/_dbs_info` . . . 311
12.2.5 `/_cluster_setup` . . . 314
12.2.6 `/_db_updates` . . . 316
12.2.7 `/_membership` . . . 318
12.2.8 `/_replicate` . . . 319
12.2.9 `/_scheduler/jobs` . . . 325
12.2.10 `/_scheduler/docs` . . . 327
12.2.11 `/_node/{node-name}` . . . 334
12.2.12 `/_node/{node-name}/_stats` . . . 335
12.2.13 `/_node/{node-name}/_prometheus` . . . 337
12.2.14 `/_node/{node-name}/_smoosh/status` . . . 340
12.2.15 `/_node/{node-name}/_system` . . . 342
12.2.16 `/_node/{node-name}/_restart` . . . 343
12.2.17 `/_node/{node-name}/_versions` . . . 343
12.2.18 `/_search_analyze` . . . 344
12.2.19 `/_nouveau_analyze` . . . 345
12.2.20 `/_utils` . . . 346
12.2.21 `/_up` . . . 346
12.2.22 `/_uuids` . . . 347
12.2.23 `/favicon.ico` . . . 348
12.2.24 `/_reshard` . . . 348
12.2.25 Authentication . . . 357
12.2.26 Configuration . . . 364
12.3 Databases . . . 370
12.3.1 `/{db}` . . . 370
12.3.2 `/{db}/_all_docs` . . . 378

12.3.3 /{db}/_design_docs . . . 380
12.3.4 /{db}/_bulk_get . . . 387
12.3.5 /{db}/_bulk_docs . . . 391
12.3.6 /{db}/_find . . . 397
12.3.7 /{db}/_index . . . 411
12.3.8 /{db}/_explain . . . 418
12.3.9 /{db}/_shards . . . 423
12.3.10 /{db}/_shards/{docid} . . . 424
12.3.11 /{db}/_sync_shards . . . 425
12.3.12 /{db}/_changes . . . 426
12.3.13 /{db}/_compact . . . 440
12.3.14 /{db}/_compact/{ddoc} . . . 442
12.3.15 /{db}/_ensure_full_commit . . . 443
12.3.16 /{db}/_view_cleanup . . . 444
12.3.17 /{db}/_security . . . 445
12.3.18 /{db}/_purge . . . 448
12.3.19 /{db}/_purged_infos . . . 451
12.3.20 /{db}/_purged_infos_limit . . . 452
12.3.21 /{db}/_missing_revs . . . 453
12.3.22 /{db}/_revs_diff . . . 455
12.3.23 /{db}/_revs_limit . . . 456
12.4 Documents . . . 458
12.4.1 /{db}/{docid} . . . 458
12.4.2 /{db}/{docid}/{attname} . . . 478
12.5 Design Documents . . . 483
12.5.1 /{db}/_design/{ddoc} . . . 483
12.5.2 /{db}/_design/{ddoc}/{attname} . . . 484
12.5.3 /{db}/_design/{ddoc}/_info . . . 485
12.5.4 /{db}/_design/{ddoc}/_view/{view} . . . 487
12.5.5 /{db}/_design/{ddoc}/_search/{index} . . . 503
12.5.6 /{db}/_design/{ddoc}/_search_info/{index} . . . 505
12.5.7 /{db}/_design/{ddoc}/_nouveau/{index} . . . 506
12.5.8 /{db}/_design/{ddoc}/_nouveau_info/{index} . . . 507
12.5.9 /{db}/_design/{ddoc}/_show/{func} . . . 508
12.5.10 /{db}/_design/{ddoc}/_show/{func}/{docid} . . . 509
12.5.11 /{db}/_design/{ddoc}/_list/{func}/{view} . . . 510
12.5.12 /{db}/_design/{ddoc}/_list/{func}/{other-ddoc}/{view} . . . 511
12.5.13 /{db}/_design/{ddoc}/_update/{func} . . . 512
12.5.14 /{db}/_design/{ddoc}/_update/{func}/{docid} . . . 513
12.5.15 /{db}/_design/{ddoc}/_rewrite/{path} . . . 514
12.6 Partitioned Databases . . . 517
12.6.1 /{db}/_partition/{partition_id} . . . 518
12.6.2 /{db}/_partition/{partition_id}/_all_docs . . . 518
12.6.3 /{db}/_partition/{partition_id}/_design/{ddoc}/_view/{view} . . . 519
12.6.4 /{db}/_partition/{partition_id}/_find . . . 520
12.6.5 /{db}/_partition/{partition_id}/_explain . . . 521
12.7 Local (non-replicating) Documents . . . 521
12.7.1 /{db}/_local_docs . . . 521
12.7.2 /{db}/_local_docs/queries . . . 524
12.7.3 /{db}/_local/{docid} . . . 526

13 JSON Structure Reference **527**
13.1 All Database Documents . . . 527
13.2 Bulk Document Response . . . 527
13.3 Bulk Documents . . . 527
13.4 Changes information for a database . . . 528
13.5 CouchDB Document . . . 528
13.6 CouchDB Error Status . . . 528

13.7 CouchDB database information object . . . 528
13.8 Design Document . . . 529
13.9 Design Document Information . . . 529
13.10 Document with Attachments . . . 529
13.11 List of Active Tasks . . . 529
13.12 Replication Settings . . . 530
13.13 Replication Status . . . 530
13.14 Request object . . . 531
13.15 Request2 object . . . 533
13.16 Response object . . . 533
13.17 Returned CouchDB Document with Detailed Revision Info . . . 533
13.18 Returned CouchDB Document with Revision Info . . . 534
13.19 Returned Document with Attachments . . . 534
13.20 Security Object . . . 534
13.21 User Context Object . . . 535
13.22 View Head Information . . . 535

14 Query Server **537**
14.1 Query Server Protocol . . . 537
14.1.1 `reset` . . . 537
14.1.2 `add_lib` . . . 538
14.1.3 `add_fun` . . . 538
14.1.4 `map_doc` . . . 539
14.1.5 `reduce` . . . 540
14.1.6 `rereduce` . . . 540
14.1.7 `ddoc` . . . 541
14.1.8 Returning errors . . . 554
14.1.9 Logging . . . 555
14.2 JavaScript . . . 555
14.2.1 Design functions context . . . 556
14.2.2 CommonJS Modules . . . 559
14.3 Erlang . . . 559

15 Partitioned Databases **563**
15.1 What is a partition? . . . 564
15.2 Why use partitions? . . . 564
15.3 Partitions By Example . . . 565

16 Release Notes **569**
16.1 3.3.x Branch . . . 569
16.1.1 Version 3.3.3 . . . 569
16.1.2 Version 3.3.2 . . . 570
16.1.3 Version 3.3.1 . . . 570
16.1.4 Version 3.3.0 . . . 570
16.2 3.2.x Branch . . . 575
16.2.1 Version 3.2.3 . . . 576
16.2.2 Version 3.2.2 . . . 576
16.2.3 Version 3.2.1 . . . 576
16.2.4 Version 3.2.0 . . . 577
16.3 3.1.x Branch . . . 581
16.3.1 Version 3.1.2 . . . 581
16.3.2 Version 3.1.1 . . . 581
16.3.3 Version 3.1.0 . . . 582
16.4 3.0.x Branch . . . 583
16.4.1 Upgrade Notes . . . 583
16.4.2 Version 3.0.1 . . . 585
16.4.3 Version 3.0.0 . . . 586
16.5 2.3.x Branch . . . 592
16.5.1 Upgrade Notes . . . 592

16.5.2 Version 2.3.1 . . . 593
16.5.3 Version 2.3.0 . . . 594
16.6 2.2.x Branch . . . 597
16.6.1 Upgrade Notes . . . 597
16.6.2 Version 2.2.0 . . . 598
16.7 2.1.x Branch . . . 602
16.7.1 Upgrade Notes . . . 602
16.7.2 Version 2.1.2 . . . 603
16.7.3 Version 2.1.1 . . . 603
16.7.4 Version 2.1.0 . . . 605
16.7.5 Fixed Issues . . . 607
16.8 2.0.x Branch . . . 609
16.8.1 Version 2.0.0 . . . 609
16.8.2 Upgrade Notes . . . 610
16.8.3 Known Issues . . . 610
16.8.4 Breaking Changes . . . 611
16.9 1.7.x Branch . . . 611
16.9.1 Version 1.7.2 . . . 611
16.9.2 Version 1.7.1 . . . 611
16.9.3 Version 1.7.0 . . . 611
16.10 1.6.x Branch . . . 613
16.10.1 Upgrade Notes . . . 613
16.10.2 Version 1.6.0 . . . 613
16.11 1.5.x Branch . . . 614
16.11.1 Version 1.5.1 . . . 614
16.11.2 Version 1.5.0 . . . 614
16.12 1.4.x Branch . . . 614
16.12.1 Upgrade Notes . . . 615
16.12.2 Version 1.4.0 . . . 615
16.13 1.3.x Branch . . . 615
16.13.1 Upgrade Notes . . . 615
16.13.2 Version 1.3.1 . . . 616
16.13.3 Version 1.3.0 . . . 616
16.14 1.2.x Branch . . . 619
16.14.1 Upgrade Notes . . . 619
16.14.2 Version 1.2.2 . . . 620
16.14.3 Version 1.2.1 . . . 620
16.14.4 Version 1.2.0 . . . 621
16.15 1.1.x Branch . . . 623
16.15.1 Upgrade Notes . . . 623
16.15.2 Version 1.1.2 . . . 623
16.15.3 Version 1.1.1 . . . 624
16.15.4 Version 1.1.0 . . . 625
16.16 1.0.x Branch . . . 626
16.16.1 Upgrade Notes . . . 626
16.16.2 Version 1.0.4 . . . 627
16.16.3 Version 1.0.3 . . . 627
16.16.4 Version 1.0.2 . . . 628
16.16.5 Version 1.0.1 . . . 629
16.16.6 Version 1.0.0 . . . 630
16.17 0.11.x Branch . . . 631
16.17.1 Upgrade Notes . . . 631
16.17.2 Version 0.11.2 . . . 632
16.17.3 Version 0.11.1 . . . 633
16.17.4 Version 0.11.0 . . . 635
16.18 0.10.x Branch . . . 637
16.18.1 Upgrade Notes . . . 637
16.18.2 Version 0.10.2 . . . 638

16.18.3 Version 0.10.1 . . . 638
16.18.4 Version 0.10.0 . . . 639
16.19 0.9.x Branch . . . 639
16.19.1 Upgrade Notes . . . 639
16.19.2 Version 0.9.2 . . . 640
16.19.3 Version 0.9.1 . . . 641
16.19.4 Version 0.9.0 . . . 642
16.20 0.8.x Branch . . . 644
16.20.1 Version 0.8.1-incubating . . . 644
16.20.2 Version 0.8.0-incubating . . . 645

17 Security Issues / CVEs **647**
17.1 CVE-2010-0009: Apache CouchDB Timing Attack Vulnerability . . . 647
17.1.1 Description . . . 647
17.1.2 Mitigation . . . 647
17.1.3 Example . . . 647
17.1.4 Credit . . . 647
17.2 CVE-2010-2234: Apache CouchDB Cross Site Request Forgery Attack . . . 648
17.2.1 Description . . . 648
17.2.2 Mitigation . . . 648
17.2.3 Example . . . 648
17.2.4 Credit . . . 648
17.3 CVE-2010-3854: Apache CouchDB Cross Site Scripting Issue . . . 648
17.3.1 Description . . . 649
17.3.2 Mitigation . . . 649
17.3.3 Example . . . 649
17.3.4 Credit . . . 649
17.4 CVE-2012-5641: Information disclosure via unescaped backslashes in URLs on Windows . . . 649
17.4.1 Description . . . 649
17.4.2 Mitigation . . . 650
17.4.3 Work-Around . . . 650
17.4.4 Acknowledgement . . . 650
17.4.5 References . . . 650
17.5 CVE-2012-5649: JSONP arbitrary code execution with Adobe Flash . . . 650
17.5.1 Description . . . 651
17.5.2 Mitigation . . . 651
17.5.3 Work-Around . . . 651
17.6 CVE-2012-5650: DOM based Cross-Site Scripting via Futon UI . . . 651
17.6.1 Description . . . 651
17.6.2 Mitigation . . . 651
17.6.3 Work-Around . . . 652
17.6.4 Acknowledgement . . . 652
17.7 CVE-2014-2668: DoS (CPU and memory consumption) via the count parameter to /_uuids . . . 652
17.7.1 Description . . . 652
17.7.2 Mitigation . . . 652
17.7.3 Work-Around . . . 653
17.8 CVE-2017-12635: Apache CouchDB Remote Privilege Escalation . . . 653
17.8.1 Description . . . 653
17.8.2 Mitigation . . . 653
17.8.3 Example . . . 653
17.8.4 Credit . . . 654
17.9 CVE-2017-12636: Apache CouchDB Remote Code Execution . . . 654
17.9.1 Description . . . 654
17.9.2 Mitigation . . . 654
17.9.3 Credit . . . 654
17.10 CVE-2018-11769: Apache CouchDB Remote Code Execution . . . 654
17.10.1 Description . . . 655
17.10.2 Mitigation . . . 655

17.11 CVE-2018-17188: Apache CouchDB Remote Privilege Escalations . . . 655
17.11.1 Description . . . 655
17.11.2 Mitigation . . . 656
17.11.3 Credit . . . 656
17.12 CVE-2018-8007: Apache CouchDB Remote Code Execution . . . 656
17.12.1 Description . . . 656
17.12.2 Mitigation . . . 656
17.12.3 Credit . . . 657
17.13 CVE-2020-1955: Apache CouchDB Remote Privilege Escalation . . . 657
17.13.1 Description . . . 657
17.13.2 Mitigation . . . 657
17.13.3 Credit . . . 657
17.14 CVE-2021-38295: Apache CouchDB Privilege Escalation . . . 657
17.14.1 Description . . . 658
17.14.2 Mitigation . . . 658
17.14.3 Credit . . . 658
17.15 CVE-2022-24706: Apache CouchDB Remote Privilege Escalation . . . 658
17.15.1 Description . . . 658
17.15.2 Mitigation . . . 659
17.15.3 Credit . . . 659
17.16 CVE-2023-26268: Apache CouchDB: Information sharing via couchjs processes . . . 659
17.16.1 Description . . . 659
17.16.2 Mitigation . . . 660
17.16.3 Workarounds . . . 660
17.16.4 Credit . . . 660
17.17 CVE-2023-45725: Apache CouchDB: Privilege Escalation Using Design Documents . . . 660
17.17.1 Description . . . 660
17.17.2 Mitigation . . . 661
17.17.3 Workarounds . . . 661
17.17.4 Credit . . . 662

18 Reporting New Security Problems with Apache CouchDB **663**

19 About CouchDB Documentation **665**
19.1 License . . . 665

20 Contributing to this Documentation **669**
20.1 Style Guidelines for this Documentation . . . 671
20.1.1 The guidelines . . . 671

API Quick Reference **673**

Configuration Quick Reference **675**

Index **679**

CHAPTER

ONE

INTRODUCTION

CouchDB is a database that completely embraces the web. Store your data with JSON documents. Access your documents with your web browser, *via HTTP*. *Query*, *combine*, and *transform* your documents with *JavaScript*. CouchDB works well with modern web and mobile apps. You can distribute your data, efficiently using CouchDB's *incremental replication*. CouchDB supports master-master setups with *automatic conflict* detection.

CouchDB comes with a suite of features, such as on-the-fly document transformation and real-time *change notifications*, that make web development a breeze. It even comes with an easy to use web administration console, served directly out of CouchDB! We care a lot about distributed scaling. CouchDB is highly available and partition tolerant, but is also *eventually consistent*. And we care *a lot* about your data. CouchDB has a fault-tolerant storage engine that puts the safety of your data first.

In this section you'll learn about every basic bit of CouchDB, see upon what conceptions and technologies it built and walk through short tutorial that teach how to use CouchDB.

1.1 Technical Overview

1.1.1 Document Storage

A CouchDB server hosts named databases, which store **documents**. Each document is uniquely named in the database, and CouchDB provides a RESTful *HTTP API* for reading and updating (add, edit, delete) database documents.

Documents are the primary unit of data in CouchDB and consist of any number of fields and attachments. Documents also include metadata that's maintained by the database system. Document fields are uniquely named and contain values of *varying types* (text, number, boolean, lists, etc), and there is no set limit to text size or element count.

The CouchDB document update model is lockless and optimistic. Document edits are made by client applications loading documents, applying changes, and saving them back to the database. If another client editing the same document saves their changes first, the client gets an edit conflict error on save. To resolve the update conflict, the latest document version can be opened, the edits reapplied and the update tried again.

Single document updates (add, edit, delete) are all or nothing, either succeeding entirely or failing completely. The database never contains partially saved or edited documents.

1.1.2 ACID Properties

The CouchDB file layout and commitment system features all *Atomic Consistent Isolated Durable* (ACID) properties. On-disk, CouchDB never overwrites committed data or associated structures, ensuring the database file is always in a consistent state. This is a "crash-only" design where the CouchDB server does not go through a shut down process, it's simply terminated.

Document updates (add, edit, delete) are serialized, except for binary blobs which are written concurrently. Database readers are never locked out and never have to wait on writers or other readers. Any number of clients can be reading documents without being locked out or interrupted by concurrent updates, even on the same document. CouchDB read operations use a *Multi-Version Concurrency Control* (MVCC) model where each client sees a consistent snapshot of the database from the beginning to the end of the read operation. This means that CouchDB can guarantee transactional semantics on a per-document basis.

Documents are indexed in B-trees by their name (DocID) and a Sequence ID. Each update to a database instance generates a new sequential number. Sequence IDs are used later for incrementally finding changes in a database. These B-tree indexes are updated simultaneously when documents are saved or deleted. The index updates always occur at the end of the file (append-only updates).

Documents have the advantage of data being already conveniently packaged for storage rather than split out across numerous tables and rows in most database systems. When documents are committed to disk, the document fields and metadata are packed into buffers, sequentially one document after another (helpful later for efficient building of views).

When CouchDB documents are updated, all data and associated indexes are flushed to disk and the transactional commit always leaves the database in a completely consistent state. Commits occur in two steps:

1. All document data and associated index updates are synchronously flushed to disk.
2. The updated database header is written in two consecutive, identical chunks to make up the first 4k of the file, and then synchronously flushed to disk.

In the event of an OS crash or power failure during step 1, the partially flushed updates are simply forgotten on restart. If such a crash happens during step 2 (committing the header), a surviving copy of the previous identical headers will remain, ensuring coherency of all previously committed data. Excepting the header area, consistency checks or fix-ups after a crash or a power failure are never necessary.

1.1.3 Compaction

Wasted space is recovered by occasional compaction. On schedule, or when the database file exceeds a certain amount of wasted space, the compaction process clones all the active data to a new file and then discards the old file. The database remains completely online the entire time and all updates and reads are allowed to complete successfully. The old database file is deleted only when all the data has been copied and all users transitioned to the new file.

1.1.4 Views

ACID properties only deal with storage and updates, but we also need the ability to show our data in interesting and useful ways. Unlike SQL databases where data must be carefully decomposed into tables, data in CouchDB is stored in semi-structured documents. CouchDB documents are flexible and each has its own implicit structure, which alleviates the most difficult problems and pitfalls of bi-directionally replicating table schemas and their contained data.

But beyond acting as a fancy file server, a simple document model for data storage and sharing is too simple to build real applications on – it simply doesn't do enough of the things we want and expect. We want to slice and dice and see our data in many different ways. What is needed is a way to filter, organize and report on data that hasn't been decomposed into tables.

See also:

Guide to Views

View Model

To address this problem of adding structure back to unstructured and semi-structured data, CouchDB integrates a view model. Views are the method of aggregating and reporting on the documents in a database, and are built on-demand to aggregate, join and report on database documents. Because views are built dynamically and don't affect the underlying document, you can have as many different view representations of the same data as you like.

View definitions are strictly virtual and only display the documents from the current database instance, making them separate from the data they display and compatible with replication. CouchDB views are defined inside special **design documents** and can replicate across database instances like regular documents, so that not only data replicates in CouchDB, but entire application designs replicate too.

JavaScript View Functions

Views are defined using JavaScript functions acting as the map part in a map-reduce system. A *view function* takes a CouchDB document as an argument and then does whatever computation it needs to do to determine the data that is to be made available through the view, if any. It can add multiple rows to the view based on a single document, or it can add no rows at all.

See also:

View Functions

View Indexes

Views are a dynamic representation of the actual document contents of a database, and CouchDB makes it easy to create useful views of data. But generating a view of a database with hundreds of thousands or millions of documents is time and resource consuming, it's not something the system should do from scratch each time.

To keep view querying fast, the view engine maintains indexes of its views, and incrementally updates them to reflect changes in the database. CouchDB's core design is largely optimized around the need for efficient, incremental creation of views and their indexes.

Views and their functions are defined inside special "design" documents, and a design document may contain any number of uniquely named view functions. When a user opens a view and its index is automatically updated, all the views in the same design document are indexed as a single group.

The view builder uses the database sequence ID to determine if the view group is fully up-to-date with the database. If not, the view engine examines all database documents (in packed sequential order) changed since the last refresh. Documents are read in the order they occur in the disk file, reducing the frequency and cost of disk head seeks.

The views can be read and queried simultaneously while also being refreshed. If a client is slowly streaming out the contents of a large view, the same view can be concurrently opened and refreshed for another client without blocking the first client. This is true for any number of simultaneous client readers, who can read and query the view while the index is concurrently being refreshed for other clients without causing problems for the readers.

As documents are processed by the view engine through your 'map' and 'reduce' functions, their previous row values are removed from the view indexes, if they exist. If the document is selected by a view function, the function results are inserted into the view as a new row.

When view index changes are written to disk, the updates are always appended at the end of the file, serving to both reduce disk head seek times during disk commits and to ensure crashes and power failures can not cause corruption of indexes. If a crash occurs while updating a view index, the incomplete index updates are simply lost and rebuilt incrementally from its previously committed state.

1.1.5 Security and Validation

To protect who can read and update documents, CouchDB has a simple reader access and update validation model that can be extended to implement custom security models.

See also:

/{db}/_security

Administrator Access

CouchDB database instances have administrator accounts. Administrator accounts can create other administrator accounts and update design documents. Design documents are special documents containing view definitions and other special formulas, as well as regular fields and blobs.

Update Validation

As documents are written to disk, they can be validated dynamically by JavaScript functions for both security and data validation. When the document passes all the formula validation criteria, the update is allowed to continue. If the validation fails, the update is aborted and the user client gets an error response.

Both the user's credentials and the updated document are given as inputs to the validation formula, and can be used to implement custom security models by validating a user's permissions to update a document.

A basic "author only" update document model is trivial to implement, where document updates are validated to check if the user is listed in an "author" field in the existing document. More dynamic models are also possible, like checking a separate user account profile for permission settings.

The update validations are enforced for both live usage and replicated updates, ensuring security and data validation in a shared, distributed system.

See also:

Validate Document Update Functions

1.1.6 Distributed Updates and Replication

CouchDB is a peer-based distributed database system. It allows users and servers to access and update the same shared data while disconnected. Those changes can then be replicated bi-directionally later.

The CouchDB document storage, view and security models are designed to work together to make true bi-directional replication efficient and reliable. Both documents and designs can replicate, allowing full database applications (including application design, logic and data) to be replicated to laptops for offline use, or replicated to servers in remote offices where slow or unreliable connections make sharing data difficult.

The replication process is incremental. At the database level, replication only examines documents updated since the last replication. If replication fails at any step, due to network problems or crash for example, the next replication restarts at the last checkpoint.

Partial replicas can be created and maintained. Replication can be filtered by a JavaScript function, so that only particular documents or those meeting specific criteria are replicated. This can allow users to take subsets of a large shared database application offline for their own use, while maintaining normal interaction with the application and that subset of data.

Conflicts

Conflict detection and management are key issues for any distributed edit system. The CouchDB storage system treats edit conflicts as a common state, not an exceptional one. The conflict handling model is simple and "non-destructive" while preserving single document semantics and allowing for decentralized conflict resolution.

CouchDB allows for any number of conflicting documents to exist simultaneously in the database, with each database instance deterministically deciding which document is the "winner" and which are conflicts. Only the winning document can appear in views, while "losing" conflicts are still accessible and remain in the database until deleted or purged during database compaction. Because conflict documents are still regular documents, they replicate just like regular documents and are subject to the same security and validation rules.

When distributed edit conflicts occur, every database replica sees the same winning revision and each has the opportunity to resolve the conflict. Resolving conflicts can be done manually or, depending on the nature of the data and the conflict, by automated agents. The system makes decentralized conflict resolution possible while maintaining single document database semantics.

Conflict management continues to work even if multiple disconnected users or agents attempt to resolve the same conflicts. If resolved conflicts result in more conflicts, the system accommodates them in the same manner, determining the same winner on each machine and maintaining single document semantics.

See also:

Replication and conflict model

Applications

Using just the basic replication model, many traditionally single server database applications can be made distributed with almost no extra work. CouchDB replication is designed to be immediately useful for basic database applications, while also being extendable for more elaborate and full-featured uses.

With very little database work, it is possible to build a distributed document management application with granular security and full revision histories. Updates to documents can be implemented to exploit incremental field and blob replication, where replicated updates are nearly as efficient and incremental as the actual edit differences ("diffs").

1.1.7 Implementation

CouchDB is built on the Erlang OTP platform, a functional, concurrent programming language and development platform. Erlang was developed for real-time telecom applications with an extreme emphasis on reliability and availability.

Both in syntax and semantics, Erlang is very different from conventional programming languages like C or Java. Erlang uses lightweight "processes" and message passing for concurrency, it has no shared state threading and all data is immutable. The robust, concurrent nature of Erlang is ideal for a database server.

CouchDB is designed for lock-free concurrency, in the conceptual model and the actual Erlang implementation. Reducing bottlenecks and avoiding locks keeps the entire system working predictably under heavy loads. CouchDB can accommodate many clients replicating changes, opening and updating documents, and querying views whose indexes are simultaneously being refreshed for other clients, without needing locks.

For higher availability and more concurrent users, CouchDB is designed for "shared nothing" clustering. In a "shared nothing" cluster, each machine is independent and replicates data with its cluster mates, allowing individual server failures with zero downtime. And because consistency scans and fix-ups aren't needed on restart, if the entire cluster fails – due to a power outage in a datacenter, for example – the entire CouchDB distributed system becomes immediately available after a restart.

CouchDB is built from the start with a consistent vision of a distributed document database system. Unlike cumbersome attempts to bolt distributed features on top of the same legacy models and databases, it is the result of careful ground-up design, engineering and integration. The document, view, security and replication models, the special purpose query language, the efficient and robust disk layout and the concurrent and reliable nature of the Erlang platform are all carefully integrated for a reliable and efficient system.

1.2 Why CouchDB?

Apache CouchDB is one of a new breed of database management systems. This topic explains why there's a need for new systems as well as the motivations behind building CouchDB.

As CouchDB developers, we're naturally very excited to be using CouchDB. In this topic we'll share with you the reasons for our enthusiasm. We'll show you how CouchDB's schema-free document model is a better fit for common applications, how the built-in query engine is a powerful way to use and process your data, and how CouchDB's design lends itself to modularization and scalability.

1.2.1 Relax

If there's one word to describe CouchDB, it is *relax*. It is the byline to CouchDB's official logo and when you start CouchDB, you see:

```
Apache CouchDB has started. Time to relax.
```

Why is relaxation important? Developer productivity roughly doubled in the last five years. The chief reason for the boost is more powerful tools that are easier to use. Take Ruby on Rails as an example. It is an infinitely complex framework, but it's easy to get started with. Rails is a success story because of the core design focus on ease of use. This is one reason why CouchDB is relaxing: learning CouchDB and understanding its core concepts should feel natural to most everybody who has been doing any work on the Web. And it is still pretty easy to explain to non-technical people.

Getting out of the way when creative people try to build specialized solutions is in itself a core feature and one thing that CouchDB aims to get right. We found existing tools too cumbersome to work with during development or in production, and decided to focus on making CouchDB easy, even a pleasure, to use.

Another area of relaxation for CouchDB users is the production setting. If you have a live running application, CouchDB again goes out of its way to avoid troubling you. Its internal architecture is fault-tolerant, and failures occur in a controlled environment and are dealt with gracefully. Single problems do not cascade through an entire server system but stay isolated in single requests.

CouchDB's core concepts are simple (yet powerful) and well understood. Operations teams (if you have a team; otherwise, that's you) do not have to fear random behavior and untraceable errors. If anything should go wrong, you can easily find out what the problem is, but these situations are rare.

CouchDB is also designed to handle varying traffic gracefully. For instance, if a website is experiencing a sudden spike in traffic, CouchDB will generally absorb a lot of concurrent requests without falling over. It may take a little more time for each request, but they all get answered. When the spike is over, CouchDB will work with regular speed again.

The third area of relaxation is growing and shrinking the underlying hardware of your application. This is commonly referred to as scaling. CouchDB enforces a set of limits on the programmer. On first look, CouchDB might seem inflexible, but some features are left out by design for the simple reason that if CouchDB supported them, it would allow a programmer to create applications that couldn't deal with scaling up or down.

Note: CouchDB doesn't let you do things that would get you in trouble later on. This sometimes means you'll have to unlearn best practices you might have picked up in your current or past work.

1.2.2 A Different Way to Model Your Data

We believe that CouchDB will drastically change the way you build document-based applications. CouchDB combines an intuitive document storage model with a powerful query engine in a way that's so simple you'll probably be tempted to ask, "Why has no one built something like this before?"

> Django may be built for the Web, but CouchDB is built of the Web. I've never seen software that so completely embraces the philosophies behind HTTP. CouchDB makes Django look old-school in the same way that Django makes ASP look outdated.
>
> —Jacob Kaplan-Moss, Django developer

CouchDB's design borrows heavily from web architecture and the concepts of resources, methods, and representations. It augments this with powerful ways to query, map, combine, and filter your data. Add fault tolerance, extreme scalability, and incremental replication, and CouchDB defines a sweet spot for document databases.

1.2.3 A Better Fit for Common Applications

We write software to improve our lives and the lives of others. Usually this involves taking some mundane information such as contacts, invoices, or receipts and manipulating it using a computer application. CouchDB is a great fit for common applications like this because it embraces the natural idea of evolving, self-contained documents as the very core of its data model.

Self-Contained Data

An invoice contains all the pertinent information about a single transaction the seller, the buyer, the date, and a list of the items or services sold. As shown in *Figure 1. Self-contained documents*, there's no abstract reference on this piece of paper that points to some other piece of paper with the seller's name and address. Accountants appreciate the simplicity of having everything in one place. And given the choice, programmers appreciate that, too.

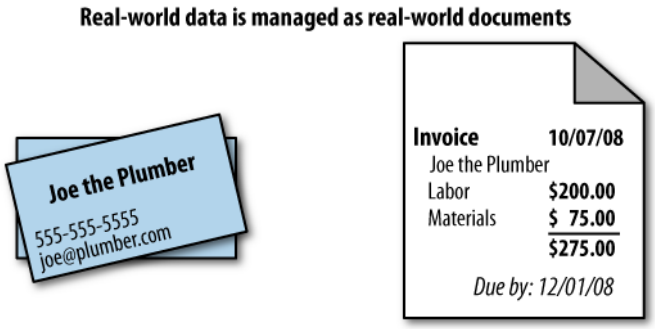

Fig. 1: Figure 1. Self-contained documents

Yet using references is exactly how we model our data in a relational database! Each invoice is stored in a table as a row that refers to other rows in other tables one row for seller information, one for the buyer, one row for each item billed, and more rows still to describe the item details, manufacturer details, and so on and so forth.

This isn't meant as a detraction of the relational model, which is widely applicable and extremely useful for a number of reasons. Hopefully, though, it illustrates the point that sometimes your model may not "fit" your data in the way it occurs in the real world.

Let's take a look at the humble contact database to illustrate a different way of modeling data, one that more closely "fits" its real-world counterpart – a pile of business cards. Much like our invoice example, a business card contains all the important information, right there on the cardstock. We call this "self-contained" data, and it's an important concept in understanding document databases like CouchDB.

Syntax and Semantics

Most business cards contain roughly the same information – someone's identity, an affiliation, and some contact information. While the exact form of this information can vary between business cards, the general information being conveyed remains the same, and we're easily able to recognize it as a business card. In this sense, we can describe a business card as a *real-world document*.

Jan's business card might contain a phone number but no fax number, whereas J. Chris's business card contains both a phone and a fax number. Jan does not have to make his lack of a fax machine explicit by writing something as ridiculous as "Fax: None" on the business card. Instead, simply omitting a fax number implies that he doesn't have one.

We can see that real-world documents of the same type, such as business cards, tend to be very similar in *semantics* – the sort of information they carry, but can vary hugely in *syntax*, or how that information is structured. As human beings, we're naturally comfortable dealing with this kind of variation.

While a traditional relational database requires you to model your data *up front*, CouchDB's schema-free design unburdens you with a powerful way to aggregate your data *after the fact*, just like we do with real-world documents. We'll look in depth at how to design applications with this underlying storage paradigm.

1.2.4 Building Blocks for Larger Systems

CouchDB is a storage system useful on its own. You can build many applications with the tools CouchDB gives you. But CouchDB is designed with a bigger picture in mind. Its components can be used as building blocks that solve storage problems in slightly different ways for larger and more complex systems.

Whether you need a system that's crazy fast but isn't too concerned with reliability (think logging), or one that guarantees storage in two or more physically separated locations for reliability, but you're willing to take a performance hit, CouchDB lets you build these systems.

There are a multitude of knobs you could turn to make a system work better in one area, but you'll affect another area when doing so. One example would be the CAP theorem discussed in *Eventual Consistency*. To give you an idea of other things that affect storage systems, see *Figure 2* and *Figure 3*.

By reducing latency for a given system (and that is true not only for storage systems), you affect concurrency and throughput capabilities.

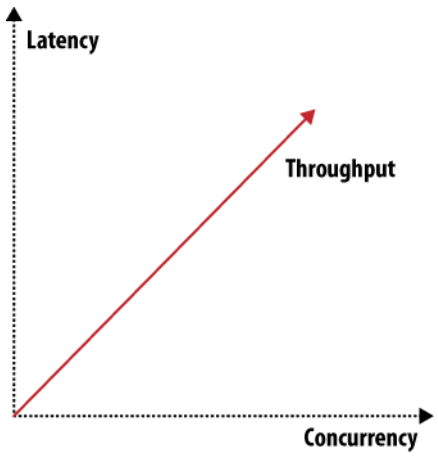

Fig. 2: Figure 2. Throughput, latency, or concurrency

When you want to scale out, there are three distinct issues to deal with: scaling read requests, write requests, and data. Orthogonal to all three and to the items shown in *Figure 2* and *Figure 3* are many more attributes like reliability or simplicity. You can draw many of these graphs that show how different features or attributes pull into different directions and thus shape the system they describe.

CouchDB is very flexible and gives you enough building blocks to create a system shaped to suit your exact problem. That's not saying that CouchDB can be bent to solve any problem – CouchDB is no silver bullet – but in the area

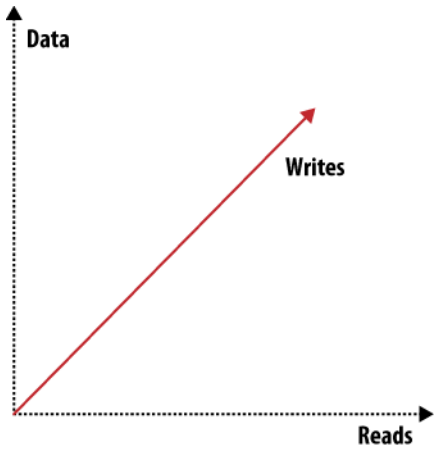

Fig. 3: Figure 3. Scaling: read requests, write requests, or data

of data storage, it can get you a long way.

1.2.5 CouchDB Replication

CouchDB replication is one of these building blocks. Its fundamental function is to synchronize two or more CouchDB databases. This may sound simple, but the simplicity is key to allowing replication to solve a number of problems: reliably synchronize databases between multiple machines for redundant data storage; distribute data to a cluster of CouchDB instances that share a subset of the total number of requests that hit the cluster (load balancing); and distribute data between physically distant locations, such as one office in New York and another in Tokyo.

CouchDB replication uses the same REST API all clients use. HTTP is ubiquitous and well understood. Replication works incrementally; that is, if during replication anything goes wrong, like dropping your network connection, it will pick up where it left off the next time it runs. It also only transfers data that is needed to synchronize databases.

A core assumption CouchDB makes is that things can go wrong, like network connection troubles, and it is designed for graceful error recovery instead of assuming all will be well. The replication system's incremental design shows that best. The ideas behind "things that can go wrong" are embodied in the Fallacies of Distributed Computing:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Existing tools often try to hide the fact that there is a network and that any or all of the previous conditions don't exist for a particular system. This usually results in fatal error scenarios when something finally goes wrong. In contrast, CouchDB doesn't try to hide the network; it just handles errors gracefully and lets you know when actions on your end are required.

1.2.6 Local Data Is King

CouchDB takes quite a few lessons learned from the Web, but there is one thing that could be improved about the Web: latency. Whenever you have to wait for an application to respond or a website to render, you almost always wait for a network connection that isn't as fast as you want it at that point. Waiting a few seconds instead of milliseconds greatly affects user experience and thus user satisfaction.

What do you do when you are offline? This happens all the time – your DSL or cable provider has issues, or your iPhone, G1, or Blackberry has no bars, and no connectivity means no way to get to your data.

CouchDB can solve this scenario as well, and this is where scaling is important again. This time it is scaling down. Imagine CouchDB installed on phones and other mobile devices that can synchronize data with centrally hosted CouchDBs when they are on a network. The synchronization is not bound by user interface constraints like sub-second response times. It is easier to tune for high bandwidth and higher latency than for low bandwidth and very low latency. Mobile applications can then use the local CouchDB to fetch data, and since no remote networking is required for that, latency is low by default.

Can you really use CouchDB on a phone? Erlang, CouchDB's implementation language has been designed to run on embedded devices magnitudes smaller and less powerful than today's phones.

1.2.7 Wrapping Up

The next document *Eventual Consistency* further explores the distributed nature of CouchDB. We should have given you enough bites to whet your interest. Let's go!

1.3 Eventual Consistency

In the previous document *Why CouchDB?*, we saw that CouchDB's flexibility allows us to evolve our data as our applications grow and change. In this topic, we'll explore how working "with the grain" of CouchDB promotes simplicity in our applications and helps us naturally build scalable, distributed systems.

1.3.1 Working with the Grain

A *distributed system* is a system that operates robustly over a wide network. A particular feature of network computing is that network links can potentially disappear, and there are plenty of strategies for managing this type of network segmentation. CouchDB differs from others by accepting eventual consistency, as opposed to putting absolute consistency ahead of raw availability, like RDBMS or Paxos. What these systems have in common is an awareness that data acts differently when many people are accessing it simultaneously. Their approaches differ when it comes to which aspects of *consistency*, *availability*, or *partition* tolerance they prioritize.

Engineering distributed systems is tricky. Many of the caveats and "gotchas" you will face over time aren't immediately obvious. We don't have all the solutions, and CouchDB isn't a panacea, but when you work with CouchDB's grain rather than against it, the path of least resistance leads you to naturally scalable applications.

Of course, building a distributed system is only the beginning. A website with a database that is available only half the time is next to worthless. Unfortunately, the traditional relational database approach to consistency makes it very easy for application programmers to rely on global state, global clocks, and other high availability no-nos, without even realizing that they're doing so. Before examining how CouchDB promotes scalability, we'll look at the constraints faced by a distributed system. After we've seen the problems that arise when parts of your application can't rely on being in constant contact with each other, we'll see that CouchDB provides an intuitive and useful way for modeling applications around high availability.

1.3.2 The CAP Theorem

The CAP theorem describes a few different strategies for distributing application logic across networks. CouchDB's solution uses replication to propagate application changes across participating nodes. This is a fundamentally different approach from consensus algorithms and relational databases, which operate at different intersections of consistency, availability, and partition tolerance.

The CAP theorem, shown in *Figure 1. The CAP theorem*, identifies three distinct concerns:

- **Consistency**: All database clients see the same data, even with concurrent updates.
- **Availability**: All database clients are able to access some version of the data.
- **Partition tolerance**: The database can be split over multiple servers.

Pick two.

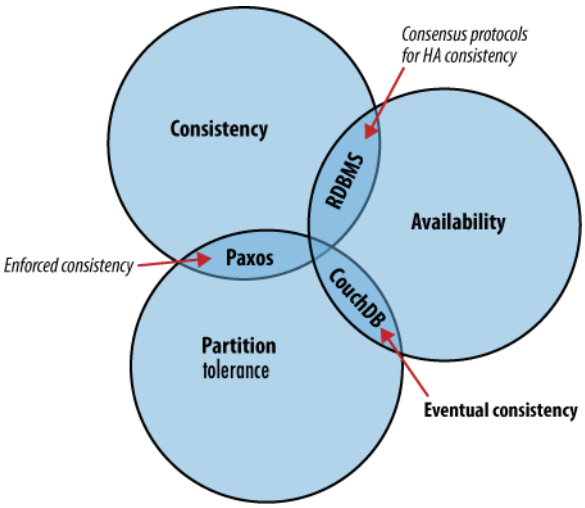

Fig. 4: Figure 1. The CAP theorem

When a system grows large enough that a single database node is unable to handle the load placed on it, a sensible solution is to add more servers. When we add nodes, we have to start thinking about how to partition data between them. Do we have a few databases that share exactly the same data? Do we put different sets of data on different database servers? Do we let only certain database servers write data and let others handle the reads?

Regardless of which approach we take, the one problem we'll keep bumping into is that of keeping all these database servers in sync. If you write some information to one node, how are you going to make sure that a read request to another database server reflects this newest information? These events might be milliseconds apart. Even with a modest collection of database servers, this problem can become extremely complex.

When it's absolutely critical that all clients see a consistent view of the database, the users of one node will have to wait for any other nodes to come into agreement before being able to read or write to the database. In this instance, we see that availability takes a backseat to consistency. However, there are situations where availability trumps consistency:

> Each node in a system should be able to make decisions purely based on local state. If you need to do something under high load with failures occurring and you need to reach agreement, you're lost. If you're concerned about scalability, any algorithm that forces you to run agreement will eventually become your bottleneck. Take that as a given.
>
> —Werner Vogels, Amazon CTO and Vice President

If availability is a priority, we can let clients write data to one node of the database without waiting for other nodes to come into agreement. If the database knows how to take care of reconciling these operations between nodes, we achieve a sort of "eventual consistency" in exchange for high availability. This is a surprisingly applicable trade-off for many applications.

Unlike traditional relational databases, where each action performed is necessarily subject to database-wide consistency checks, CouchDB makes it really simple to build applications that sacrifice immediate consistency for the huge performance improvements that come with simple distribution.

1.3.3 Local Consistency

Before we attempt to understand how CouchDB operates in a cluster, it's important that we understand the inner workings of a single CouchDB node. The CouchDB API is designed to provide a convenient but thin wrapper around the database core. By taking a closer look at the structure of the database core, we'll have a better understanding of the API that surrounds it.

The Key to Your Data

At the heart of CouchDB is a powerful *B-tree* storage engine. A B-tree is a sorted data structure that allows for searches, insertions, and deletions in logarithmic time. As *Figure 2. Anatomy of a view request* illustrates, CouchDB uses this B-tree storage engine for all internal data, documents, and views. If we understand one, we will understand them all.

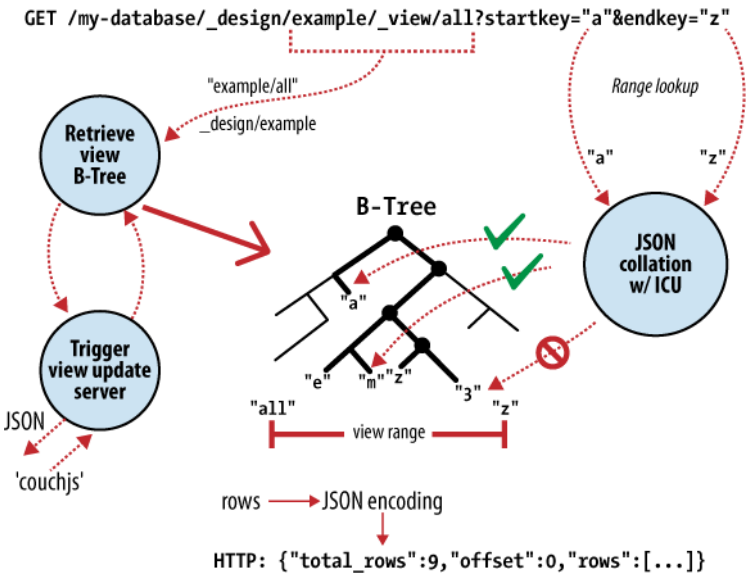

Fig. 5: Figure 2. Anatomy of a view request

CouchDB uses MapReduce to compute the results of a view. MapReduce makes use of two functions, "map" and "reduce", which are applied to each document in isolation. Being able to isolate these operations means that view computation lends itself to parallel and incremental computation. More important, because these functions produce key/value pairs, CouchDB is able to insert them into the B-tree storage engine, sorted by key. Lookups by key, or key range, are extremely efficient operations with a B-tree, described in *big O* notation as `O(log N)` and `O(log N + K)`, respectively.

In CouchDB, we access documents and view results by key or key range. This is a direct mapping to the underlying operations performed on CouchDB's B-tree storage engine. Along with document inserts and updates, this direct mapping is the reason we describe CouchDB's API as being a thin wrapper around the database core.

Being able to access results by key alone is a very important restriction because it allows us to make huge performance gains. As well as the massive speed improvements, we can partition our data over multiple nodes, without affecting our ability to query each node in isolation. BigTable, Hadoop, SimpleDB, and memcached restrict object lookups by key for exactly these reasons.

No Locking

A table in a relational database is a single data structure. If you want to modify a table – say, update a row – the database system must ensure that nobody else is trying to update that row and that nobody can read from that row while it is being updated. The common way to handle this uses what's known as a lock. If multiple clients want to access a table, the first client gets the lock, making everybody else wait. When the first client's request is processed, the next client is given access while everybody else waits, and so on. This serial execution of requests, even when they arrived in parallel, wastes a significant amount of your server's processing power. Under high load, a relational database can spend more time figuring out who is allowed to do what, and in which order, than it does doing any actual work.

Note: Modern relational databases avoid locks by implementing MVCC under the hood, but hide it from the end user, requiring them to coordinate concurrent changes of single rows or fields.

Instead of locks, CouchDB uses *Multi-Version Concurrency Control* (MVCC) to manage concurrent access to the database. *Figure 3. MVCC means no locking* illustrates the differences between MVCC and traditional locking mechanisms. MVCC means that CouchDB can run at full speed, all the time, even under high load. Requests are run in parallel, making excellent use of every last drop of processing power your server has to offer.

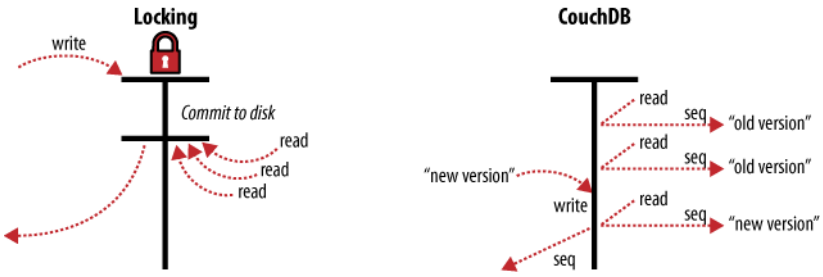

Fig. 6: Figure 3. MVCC means no locking

Documents in CouchDB are versioned, much like they would be in a regular version control system such as Subversion. If you want to change a value in a document, you create an entire new version of that document and save it over the old one. After doing this, you end up with two versions of the same document, one old and one new.

How does this offer an improvement over locks? Consider a set of requests wanting to access a document. The first request reads the document. While this is being processed, a second request changes the document. Since the second request includes a completely new version of the document, CouchDB can simply append it to the database without having to wait for the read request to finish.

When a third request wants to read the same document, CouchDB will point it to the new version that has just been written. During this whole process, the first request could still be reading the original version.

A read request will always see the most recent snapshot of your database at the time of the beginning of the request.

1.3.4 Validation

As application developers, we have to think about what sort of input we should accept and what we should reject. The expressive power to do this type of validation over complex data within a traditional relational database leaves a lot to be desired. Fortunately, CouchDB provides a powerful way to perform per-document validation from within the database.

CouchDB can validate documents using JavaScript functions similar to those used for MapReduce. Each time you try to modify a document, CouchDB will pass the validation function a copy of the existing document, a copy of the new document, and a collection of additional information, such as user authentication details. The validation function now has the opportunity to approve or deny the update.

By working with the grain and letting CouchDB do this for us, we save ourselves a tremendous amount of CPU cycles that would otherwise have been spent serializing object graphs from SQL, converting them into domain objects, and using those objects to do application-level validation.

1.3.5 Distributed Consistency

Maintaining consistency within a single database node is relatively easy for most databases. The real problems start to surface when you try to maintain consistency between multiple database servers. If a client makes a write operation on server *A*, how do we make sure that this is consistent with server *B*, or *C*, or *D*? For relational databases, this is a very complex problem with entire books devoted to its solution. You could use multi-master, single-master, partitioning, sharding, write-through caches, and all sorts of other complex techniques.

1.3.6 Incremental Replication

CouchDB's operations take place within the context of a single document. As CouchDB achieves eventual consistency between multiple databases by using incremental replication you no longer have to worry about your database servers being able to stay in constant communication. Incremental replication is a process where document changes are periodically copied between servers. We are able to build what's known as a *shared nothing* cluster of databases where each node is independent and self-sufficient, leaving no single point of contention across the system.

Need to scale out your CouchDB database cluster? Just throw in another server.

As illustrated in *Figure 4. Incremental replication between CouchDB nodes*, with CouchDB's incremental replication, you can synchronize your data between any two databases however you like and whenever you like. After replication, each database is able to work independently.

You could use this feature to synchronize database servers within a cluster or between data centers using a job scheduler such as cron, or you could use it to synchronize data with your laptop for offline work as you travel. Each database can be used in the usual fashion, and changes between databases can be synchronized later in both directions.

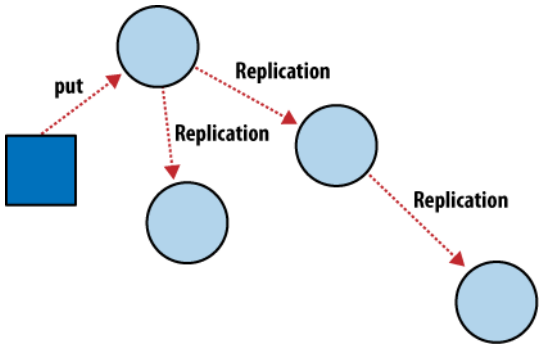

Fig. 7: Figure 4. Incremental replication between CouchDB nodes

What happens when you change the same document in two different databases and want to synchronize these with each other? CouchDB's replication system comes with automatic conflict detection and resolution. When CouchDB detects that a document has been changed in both databases, it flags this document as being in conflict, much like they would be in a regular version control system.

This isn't as troublesome as it might first sound. When two versions of a document conflict during replication, the winning version is saved as the most recent version in the document's history. Instead of throwing the losing version away, as you might expect, CouchDB saves this as a previous version in the document's history, so that you can access it if you need to. This happens automatically and consistently, so both databases will make exactly the same choice.

It is up to you to handle conflicts in a way that makes sense for your application. You can leave the chosen document versions in place, revert to the older version, or try to merge the two versions and save the result.

1.3.7 Case Study

Greg Borenstein, a friend and coworker, built a small library for converting Songbird playlists to JSON objects and decided to store these in CouchDB as part of a backup application. The completed software uses CouchDB's MVCC and document revisions to ensure that Songbird playlists are backed up robustly between nodes.

Note: Songbird is a free software media player with an integrated web browser, based on the Mozilla XULRunner platform. Songbird is available for Microsoft Windows, Apple Mac OS X, Solaris, and Linux.

Let's examine the workflow of the Songbird backup application, first as a user backing up from a single computer, and then using Songbird to synchronize playlists between multiple computers. We'll see how document revisions turn what could have been a hairy problem into something that *just works*.

The first time we use this backup application, we feed our playlists to the application and initiate a backup. Each playlist is converted to a JSON object and handed to a CouchDB database. As illustrated in *Figure 5. Backing up to a single database*, CouchDB hands back the document ID and revision of each playlist as it's saved to the database.

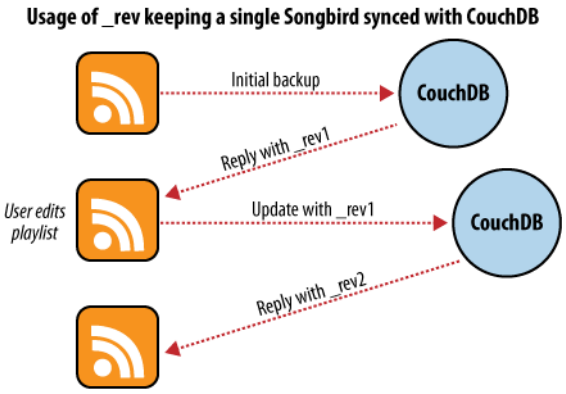

Fig. 8: Figure 5. Backing up to a single database

After a few days, we find that our playlists have been updated and we want to back up our changes. After we have fed our playlists to the backup application, it fetches the latest versions from CouchDB, along with the corresponding document revisions. When the application hands back the new playlist document, CouchDB requires that the document revision is included in the request.

CouchDB then makes sure that the document revision handed to it in the request matches the current revision held in the database. Because CouchDB updates the revision with every modification, if these two are out of sync it suggests that someone else has made changes to the document between the time we requested it from the database and the time we sent our updates. Making changes to a document after someone else has modified it without first inspecting those changes is usually a bad idea.

Forcing clients to hand back the correct document revision is the heart of CouchDB's optimistic concurrency.

We have a laptop we want to keep synchronized with our desktop computer. With all our playlists on our desktop, the first step is to "restore from backup" onto our laptop. This is the first time we've done this, so afterward our laptop should hold an exact replica of our desktop playlist collection.

After editing our Argentine Tango playlist on our laptop to add a few new songs we've purchased, we want to save our changes. The backup application replaces the playlist document in our laptop CouchDB database and a new document revision is generated. A few days later, we remember our new songs and want to copy the playlist across to our desktop computer. As illustrated in *Figure 6. Synchronizing between two databases*, the backup application copies the new document and the new revision to the desktop CouchDB database. Both CouchDB databases now have the same document revision.

Because CouchDB tracks document revisions, it ensures that updates like these will work only if they are based on current information. If we had made modifications to the playlist backups between synchronization, things wouldn't go as smoothly.

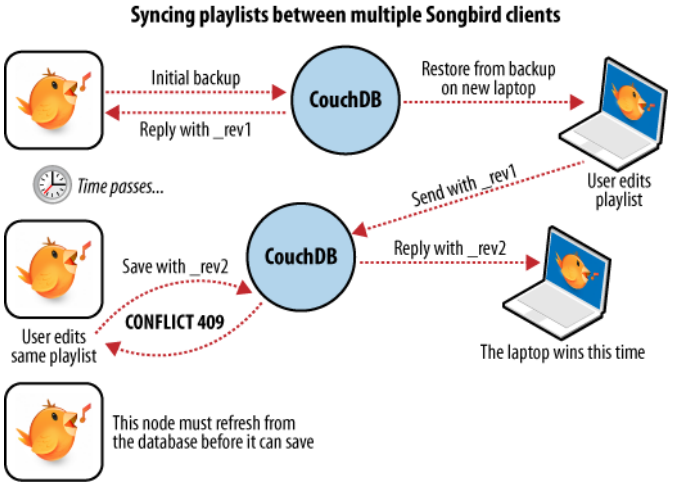

Fig. 9: Figure 6. Synchronizing between two databases

We back up some changes on our laptop and forget to synchronize. A few days later, we're editing playlists on our desktop computer, make a backup, and want to synchronize this to our laptop. As illustrated in *Figure 7. Synchronization conflicts between two databases*, when our backup application tries to replicate between the two databases, CouchDB sees that the changes being sent from our desktop computer are modifications of out-of-date documents and helpfully informs us that there has been a conflict.

Recovering from this error is easy to accomplish from an application perspective. Just download CouchDB's version of the playlist and provide an opportunity to merge the changes or save local modifications into a new playlist.

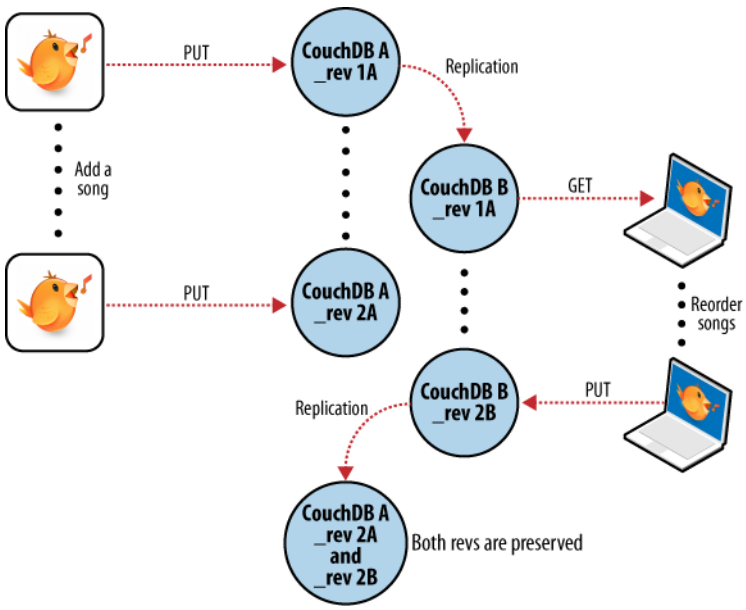

Fig. 10: Figure 7. Synchronization conflicts between two databases

1.3.8 Wrapping Up

CouchDB's design borrows heavily from web architecture and the lessons learned deploying massively distributed systems on that architecture. By understanding why this architecture works the way it does, and by learning to spot which parts of your application can be easily distributed and which parts cannot, you'll enhance your ability to design distributed and scalable applications, with CouchDB or without it.

We've covered the main issues surrounding CouchDB's consistency model and hinted at some of the benefits to be had when you work *with* CouchDB and not against it. But enough theory – let's get up and running and see what all the fuss is about!

1.4 cURL: Your Command Line Friend

The `curl` utility is a command line tool available on Unix, Linux, Mac OS X, Windows, and many other platforms. `curl` provides easy access to the HTTP protocol (among others) directly from the command line and is therefore an ideal way of interacting with CouchDB over the HTTP REST API.

For simple `GET` requests you can supply the URL of the request. For example, to get the database information:

```
shell> curl http://127.0.0.1:5984
```

This returns the database information (formatted in the output below for clarity):

```
{
  "couchdb": "Welcome",
  "version": "3.0.0",
  "git_sha": "83bdcf693",
  "uuid": "56f16e7c93ff4a2dc20eb6acc7000b71",
  "features": [
    "access-ready",
    "partitioned",
    "pluggable-storage-engines",
    "reshard",
    "scheduler"
  ],
  "vendor": {
    "name": "The Apache Software Foundation"
  }
}
```

Note: For some URLs, especially those that include special characters such as ampersand, exclamation mark, or question mark, you should quote the URL you are specifying on the command line. For example:

```
shell> curl 'http://couchdb:5984/_uuids?count=5'
```

Note: On Microsoft Windows, use doubled double-quotes ("") anywhere you see single double-quotes. For example, if you see:

```
shell> curl -X PUT 'http://adm:pass@127.0.0.1:5984/demo/doc' -d '{"motto": "I love␣
↪gnomes"}'
```

you should replace it with:

```
shell> curl -X PUT "http://adm:pass@127.0.0.1:5984/demo/doc" -d "{""motto"": ""I love␣
↪gnomes""}"
```

If you prefer, ^" and \" may be used to escape the double-quote character in quoted strings instead.

You can explicitly set the HTTP command using the -X command line option. For example, when creating a database, you set the name of the database in the URL you send using a PUT request:

```
shell> curl -X PUT http://adm:pass@127.0.0.1:5984/demo
{"ok":true}
```

But to obtain the database information you use a GET request (with the return information formatted for clarity):

```
shell> curl -X GET http://adm:pass@127.0.0.1:5984/demo
{
    "compact_running" : false,
    "doc_count" : 0,
    "db_name" : "demo",
    "purge_seq" : 0,
    "committed_update_seq" : 0,
    "doc_del_count" : 0,
    "disk_format_version" : 5,
    "update_seq" : 0,
    "instance_start_time" : "0",
    "disk_size" : 79
}
```

For certain operations, you must specify the content type of request, which you do by specifying the Content-Type header using the -H command-line option:

```
shell> curl -H 'Content-Type: application/json' http://127.0.0.1:5984/_uuids
```

You can also submit 'payload' data, that is, data in the body of the HTTP request using the -d option. This is useful if you need to submit JSON structures, for example document data, as part of the request. For example, to submit a simple document to the demo database:

```
shell> curl -H 'Content-Type: application/json' \
            -X POST http://adm:pass@127.0.0.1:5984/demo \
            -d '{"company": "Example, Inc."}'
{"ok":true,"id":"8843faaf0b831d364278331bc3001bd8",
 "rev":"1-33b9fbce46930280dab37d672bbc8bb9"}
```

In the above example, the argument after the -d option is the JSON of the document we want to submit.

The document can be accessed by using the automatically generated document ID that was returned:

```
shell> curl -X GET http://adm:pass@127.0.0.1:5984/demo/
↪8843faaf0b831d364278331bc3001bd8
{"_id":"8843faaf0b831d364278331bc3001bd8",
 "_rev":"1-33b9fbce46930280dab37d672bbc8bb9",
 "company":"Example, Inc."}
```

The API samples in the *API Basics* show the HTTP command, URL and any payload information that needs to be submitted (and the expected return value). All of these examples can be reproduced using curl with the command-line examples shown above.

1.5 Security

In this document, we'll look at the basic security mechanisms in CouchDB: *Basic Authentication* and *Cookie Authentication*. This is how CouchDB handles users and protects their credentials.

1.5.1 Authentication

CouchDB has the idea of an *admin user* (e.g. an administrator, a super user, or root) that is allowed to do anything to a CouchDB installation. By default, one admin user **must** be created for CouchDB to start up successfully.

CouchDB also defines a set of requests that only admin users are allowed to do. If you have defined one or more specific admin users, CouchDB will ask for identification for certain requests:

- Creating a database (*PUT /database*)
- Deleting a database (*DELETE /database*)
- Setup a database security (*PUT /database/_security*)
- Creating a design document (*PUT /database/_design/app*)
- Updating a design document (*PUT /database/_design/app?rev=1-4E2*)
- Deleting a design document (*DELETE /database/_design/app?rev=2-6A7*)
- Triggering compaction (*POST /database/_compact*)
- Reading the task status list (*GET /_active_tasks*)
- Restarting the server on a given node (*POST /_node/{node-name}/_restart*)
- Reading the active configuration (*GET /_node/{node-name}/_config*)
- Updating the active configuration (*PUT /_node/{node-name}/_config/{section}/{key}*)

Creating a New Admin User

If your installation process did not set up an admin user, you will have to add one to the configuration file by hand and restart CouchDB first. For the purposes of this example, we'll create a default `admin` user with the password `password`.

> **Warning:** Don't just type in the following without thinking! Pick a good name for your administrator user that isn't easily guessable, and pick a secure password.

To the end of your `etc/local.ini` file, after the `[admins]` line, add the text `admin = password`, so it looks like this:

```
[admins]
admin = password
```

(Don't worry about the password being in plain text; we'll come back to this.)

Now, restart CouchDB using the method appropriate for your operating system. You should now be able to access CouchDB using your new administrator account:

```
> curl http://admin:password@127.0.0.1:5984/_up
{"status":"ok","seeds":{}}
```

Great!

Let's create an admin user through the HTTP API. We'll call her `anna`, and her password is `secret`. Note the double quotes in the following code; they are needed to denote a string value for the *configuration API*:

```
> HOST="http://admin:password@127.0.0.1:5984"
> NODENAME="_local"
> curl -X PUT $HOST/_node/$NODENAME/_config/admins/anna -d '"secret"'
""
```

As per the *_config* API's behavior, we're getting the previous value for the config item we just wrote. Since our admin user didn't exist, we get an empty string.

Please note that `_local` serves as an alias for the local node name, so for all configuration URLs, `NODENAME` may be set to `_local`, to interact with the local node's configuration.

See also:

Node Management

Hashing Passwords

Seeing the plain-text password is scary, isn't it? No worries, CouchDB doesn't show the plain-text password anywhere. It gets hashed right away. Go ahead and look at your `local.ini` file now. You'll see that CouchDB has rewritten the plain text passwords so they are hashed:

```
[admins]
admin = -pbkdf2-71c01cb429088ac1a1e95f3482202622dc1e53fe,
↪226701bece4ae0fc9a373a5e02bf5d07,10
anna = -pbkdf2-2d86831c82b440b8887169bd2eebb356821d621b,
↪5e11b9a9228414ab92541beeeacbf125,10
```

The hash is that big, ugly, long string that starts out with `-pbkdf2-`.

To compare a plain-text password during authentication with the stored hash, the hashing algorithm is run and the resulting hash is compared to the stored hash. The probability of two identical hashes for different passwords is too insignificant to mention (c.f. Bruce Schneier). Should the stored hash fall into the hands of an attacker, it is, by current standards, way too inconvenient (i.e., it'd take a lot of money and time) to find the plain-text password from the hash.

When CouchDB starts up, it reads a set of `.ini` files with config settings. It loads these settings into an internal data store (not a database). The config API lets you read the current configuration as well as change it and create new entries. CouchDB writes any changes back to the `.ini` files.

The `.ini` files can also be edited by hand when CouchDB is not running. Instead of creating the admin user as we showed previously, you could have stopped CouchDB, opened your `local.ini`, added `anna = secret` to the *admins*, and restarted CouchDB. Upon reading the new line from `local.ini`, CouchDB would run the hashing algorithm and write back the hash to `local.ini`, replacing the plain-text password — just as it did for our original `admin` user. To make sure CouchDB only hashes plain-text passwords and not an existing hash a second time, it prefixes the hash with `-pbkdf2-`, to distinguish between plain-text passwords and PBKDF2 hashed passwords. This means your plain-text password can't start with the characters `-pbkdf2-`, but that's pretty unlikely to begin with.

Basic Authentication

CouchDB will not allow us to create new databases unless we give the correct admin user credentials. Let's verify:

```
> HOST="http://127.0.0.1:5984"
> curl -X PUT $HOST/somedatabase
{"error":"unauthorized","reason":"You are not a server admin."}
```

That looks about right. Now we try again with the correct credentials:

```
> HOST="http://anna:secret@127.0.0.1:5984"
> curl -X PUT $HOST/somedatabase
{"ok":true}
```

If you have ever accessed a website or FTP server that was password-protected, the `username:password@` URL variant should look familiar.

If you are security conscious, the missing `s` in `http://` will make you nervous. We're sending our password to CouchDB in plain text. This is a bad thing, right? Yes, but consider our scenario: CouchDB listens on `127.0.0.1` on a development box that we're the sole user of. Who could possibly sniff our password?

If you are in a production environment, however, you need to reconsider. Will your CouchDB instance communicate over a public network? Even a LAN shared with other collocation customers is public. There are multiple ways to secure communication between you or your application and CouchDB that exceed the scope of this documentation. CouchDB as of version *1.1.0* comes with *SSL built in*.

See also:

Basic Authentication API Reference

Cookie Authentication

Basic authentication that uses plain-text passwords is nice and convenient, but not very secure if no extra measures are taken. It is also a very poor user experience. If you use basic authentication to identify admins, your application's users need to deal with an ugly, unstylable browser modal dialog that says non-professional at work more than anything else.

To remedy some of these concerns, CouchDB supports cookie authentication. With cookie authentication your application doesn't have to include the ugly login dialog that the users' browsers come with. You can use a regular HTML form to submit logins to CouchDB. Upon receipt, CouchDB will generate a one-time token that the client can use in its next request to CouchDB. When CouchDB sees the token in a subsequent request, it will authenticate the user based on the token without the need to see the password again. By default, a token is valid for 10 minutes.

To obtain the first token and thus authenticate a user for the first time, the username and password must be sent to the *_session* API. The API is smart enough to decode HTML form submissions, so you don't have to resort to any smarts in your application.

If you are not using HTML forms to log in, you need to send an HTTP request that looks as if an HTML form generated it. Luckily, this is super simple:

```
> HOST="http://127.0.0.1:5984"
> curl -vX POST $HOST/_session \
       -H 'Content-Type:application/x-www-form-urlencoded' \
       -d 'name=anna&password=secret'
```

CouchDB replies, and we'll give you some more detail:

```
< HTTP/1.1 200 OK
< Set-Cookie: AuthSession=YW5uYTo0QUIzOTdFQjrC4ipN-D-53hw1sJepVzcVxnriEw;
< Version=1; Path=/; HttpOnly
> ...
<
{"ok":true}
```

A 200 OK response code tells us all is well, a Set-Cookie header includes the token we can use for the next request, and the standard JSON response tells us again that the request was successful.

Now we can use this token to make another request as the same user without sending the username and password again:

```
> curl -vX PUT $HOST/mydatabase \
       --cookie AuthSession=YW5uYTo0QUIzOTdFQjrC4ipN-D-53hw1sJepVzcVxnriEw \
       -H "X-CouchDB-WWW-Authenticate: Cookie" \
       -H "Content-Type:application/x-www-form-urlencoded"
{"ok":true}
```

You can keep using this token for 10 minutes by default. After 10 minutes you need to authenticate your user again. The token lifetime can be configured with the timeout (in seconds) setting in the *chttpd_auth* configuration section.

See also:

Cookie Authentication API Reference

1.5.2 Authentication Database

You may already note that CouchDB administrators are defined within the config file and are wondering if regular users are also stored there. No, they are not. CouchDB has a special *authentication database*, named `_users` by default, that stores all registered users as JSON documents.

This special database is a *system database*. This means that while it shares the common *database API*, there are some special security-related constraints applied. Below is a list of how the *authentication database* is different from the other databases.

- Only administrators may browse list of all documents (`GET /_users/_all_docs`)
- Only administrators may listen to *changes feed* (`GET /_users/_changes`)
- Only administrators may execute design functions like *views*.
- There is a special design document `_auth` that cannot be modified
- Every document except the *design documents* represent registered CouchDB users and belong to them
- By default, the `_security` settings of the `_users` database disallow users from accessing or modifying documents

Note: Settings can be changed so that users do have access to the `_users` database, but even then they may only access (`GET /_users/org.couchdb.user:Jan`) or modify (`PUT /_users/org.couchdb.user:Jan`) documents that they own. This will not be possible in CouchDB 4.0.

These draconian rules are necessary since CouchDB cares about its users' personal information and will not disclose it to just anyone. Often, user documents contain system information like *login*, *password hash* and *roles*, apart from sensitive personal information like real name, email, phone, special internal identifications and more. This is not information that you want to share with the World.

Users Documents

Each CouchDB user is stored in document format. These documents contain several *mandatory* fields, that CouchDB needs for authentication:

- **_id** (*string*): Document ID. Contains user's login with special prefix *Why the org.couchdb.user: prefix?*
- **derived_key** (*string*): PBKDF2 key derived from prf/salt/iterations.
- **name** (*string*): User's name aka login. **Immutable** e.g. you cannot rename an existing user - you have to create new one
- **roles** (*array* of *string*): List of user roles. CouchDB doesn't provide any built-in roles, so you're free to define your own depending on your needs. However, you cannot set system roles like `_admin` there. Also, only administrators may assign roles to users - by default all users have no roles

- **password** (*string*): A plaintext password can be provided, but will be replaced by hashed fields before the document is actually stored.
- **password_sha** (*string*): Hashed password with salt. Used for `simple` *password_scheme*
- **password_scheme** (*string*): Password hashing scheme. May be `simple` or `pbkdf2`
- **salt** (*string*): Hash salt. Used for both `simple` and `pbkdf2` `password_scheme` options.
- **iterations** (*integer*): Number of iterations to derive key, used for `pbkdf2` `password_scheme` See the *configuration API*:: for details.
- **pbkdf2_prf** (*string*): The PRF to use for `pbkdf2`. If missing, `sha` is assumed. Can be any of `sha`, `sha224`, `sha256`, `sha384`, `sha512`.
- **type** (*string*): Document type. Constantly has the value `user`

Additionally, you may specify any custom fields that relate to the target user.

Why the `org.couchdb.user:` prefix?

The reason there is a special prefix before a user's login name is to have namespaces that users belong to. This prefix is designed to prevent replication conflicts when you try merging two or more *_user* databases.

For current CouchDB releases, all users belong to the same `org.couchdb.user` namespace and this cannot be changed. This may be changed in future releases.

Creating a New User

Creating a new user is a very trivial operation. You just need to do a PUT request with the user's data to CouchDB. Let's create a user with login *jan* and password *apple*:

```
curl -X PUT http://admin:password@localhost:5984/_users/org.couchdb.user:jan \
     -H "Accept: application/json" \
     -H "Content-Type: application/json" \
     -d '{"name": "jan", "password": "apple", "roles": [], "type": "user"}'
```

This *curl* command will produce the following HTTP request:

```
PUT /_users/org.couchdb.user:jan HTTP/1.1
Accept: application/json
Content-Length: 62
Content-Type: application/json
Host: localhost:5984
User-Agent: curl/7.31.0
```

And CouchDB responds with:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 83
Content-Type: application/json
Date: Fri, 27 Sep 2013 07:33:28 GMT
ETag: "1-e0ebfb84005b920488fc7a8cc5470cc0"
Location: http://localhost:5984/_users/org.couchdb.user:jan
Server: CouchDB (Erlang OTP)

{"ok":true,"id":"org.couchdb.user:jan","rev":"1-e0ebfb84005b920488fc7a8cc5470cc0"}
```

The document was successfully created! The user *jan* should now exist in our database. Let's check if this is true:

```
curl -X POST http://localhost:5984/_session -d 'name=jan&password=apple'
```

CouchDB should respond with:

```
{"ok":true,"name":"jan","roles":[]}
```

This means that the username was recognized and the password's hash matches with the stored one. If we specify an incorrect login and/or password, CouchDB will notify us with the following error message:

```
{"error":"unauthorized","reason":"Name or password is incorrect."}
```

Password Changing

Let's define what is password changing from the point of view of CouchDB and the authentication database. Since "users" are "documents", this operation is just updating the document with a special field `password` which contains the *plain text password*. Scared? No need to be. The authentication database has a special internal hook on document update which looks for this field and replaces it with the *secured hash* depending on the chosen `password_scheme`.

Summarizing the above process - we need to get the document's content, add the `password` field with the new password in plain text and then store the JSON result to the authentication database.

```
curl -X GET http://admin:password@localhost:5984/_users/org.couchdb.user:jan
```

```
{
    "_id": "org.couchdb.user:jan",
    "_rev": "1-e0ebfb84005b920488fc7a8cc5470cc0",
    "derived_key": "e579375db0e0c6a6fc79cd9e36a36859f71575c3",
    "iterations": 10,
    "name": "jan",
    "password_scheme": "pbkdf2",
    "roles": [],
    "salt": "1112283cf988a34f124200a050d308a1",
    "type": "user"
}
```

Here is our user's document. We may strip hashes from the stored document to reduce the amount of posted data:

```
curl -X PUT http://admin:password@localhost:5984/_users/org.couchdb.user:jan \
     -H "Accept: application/json" \
     -H "Content-Type: application/json" \
     -H "If-Match: 1-e0ebfb84005b920488fc7a8cc5470cc0" \
     -d '{"name":"jan", "roles":[], "type":"user", "password":"orange"}'
```

```
{"ok":true,"id":"org.couchdb.user:jan","rev":"2-ed293d3a0ae09f0c624f10538ef33c6f"}
```

Updated! Now let's check that the password was really changed:

```
curl -X POST http://localhost:5984/_session -d 'name=jan&password=apple'
```

CouchDB should respond with:

```
{"error":"unauthorized","reason":"Name or password is incorrect."}
```

Looks like the password `apple` is wrong, what about `orange`?

```
curl -X POST http://localhost:5984/_session -d 'name=jan&password=orange'
```

CouchDB should respond with:

```
{"ok":true,"name":"jan","roles":[]}
```

Hooray! You may wonder why this was so complex - we need to retrieve user's document, add a special field to it, and post it back.

Note: There is no password confirmation for API request: you should implement it in your application layer.

1.5.3 Authorization

Now that you have a few users who can log in, you probably want to set up some restrictions on what actions they can perform based on their identity and their roles. Each database on a CouchDB server can contain its own set of authorization rules that specify which users are allowed to read and write documents, create design documents, and change certain database configuration parameters. The authorization rules are set up by a server admin and can be modified at any time.

Database authorization rules assign a user into one of two classes:

- *members*, who are allowed to read all documents and create and modify any document except for design documents.
- *admins*, who can read and write all types of documents, modify which users are members or admins, and set certain per-database configuration options.

Note that a database admin is not the same as a server admin – the actions of a database admin are restricted to a specific database.

All databases are created as admin-only by default. That is, only database admins may read or write. The default behavior can be configured with the `[couchdb] default_security` *option*. If you set that option to `everyone`, HTTP requests that have no authentication credentials or have credentials for a normal user are treated as members, and those with server admin credentials are treated as database admins.

You can also modify the permissions after the database is created by modifying the *security* document in the database:

```
> curl -X PUT http://localhost:5984/mydatabase/_security \
     -u anna:secret \
     -H "Content-Type: application/json" \
     -d '{"admins": { "names": [], "roles": [] }, "members": { "names": ["jan"],
↪"roles": [] } }'
```

The HTTP request to create or update the *_security* document must contain the credentials of a server admin. CouchDB will respond with:

```
{"ok":true}
```

The database is now secured against anonymous reads and writes:

```
> curl http://localhost:5984/mydatabase/
```

```
{"error":"unauthorized","reason":"You are not authorized to access this db."}
```

You declared user "jan" as a member in this database, so he is able to read and write normal documents:

```
> curl -u jan:orange http://localhost:5984/mydatabase/
```

```
{"db_name":"mydatabase","doc_count":1,"doc_del_count":0,"update_seq":3,"purge_seq":0,
"compact_running":false,"sizes":{"active":272,"disk":12376,"external":350},
"instance_start_time":"0","disk_format_version":6,"committed_update_seq":3}
```

If Jan attempted to create a design doc, however, CouchDB would return a 401 Unauthorized error because the username "jan" is not in the list of admin names and the */_users/org.couchdb.user:jan* document doesn't contain a role that matches any of the declared admin roles. If you want to promote Jan to an admin, you can update the security document to add *"jan"* to the *names* array under *admin*. Keeping track of individual database admin usernames is tedious, though, so you would likely prefer to create a database admin role and assign that role to the *org.couchdb.user:jan* user document:

```
> curl -X PUT http://localhost:5984/mydatabase/_security \
     -u anna:secret \
     -H "Content-Type: application/json" \
     -d '{"admins": { "names": [], "roles": ["mydatabase_admin"] }, "members": {
↪"names": [], "roles": [] } }'
```

See the *_security document reference page* for additional details about specifying database members and admins.

1.6 Getting Started

In this document, we'll take a quick tour of CouchDB's features. We'll create our first document and experiment with CouchDB views.

1.6.1 All Systems Are Go!

We'll have a very quick look at CouchDB's bare-bones Application Programming Interface (API) by using the command-line utility curl. Please note that this is not the only way of talking to CouchDB. We will show you plenty more throughout the rest of the documents. What's interesting about curl is that it gives you control over raw HTTP requests, and you can see exactly what is going on "underneath the hood" of your database.

Make sure CouchDB is still running, and then do:

```
curl http://127.0.0.1:5984/
```

This issues a GET request to your newly installed CouchDB instance.

The reply should look something like:

```
{
  "couchdb": "Welcome",
  "version": "3.0.0",
  "git_sha": "83bdcf693",
  "uuid": "56f16e7c93ff4a2dc20eb6acc7000b71",
  "features": [
    "access-ready",
    "partitioned",
    "pluggable-storage-engines",
    "reshard",
    "scheduler"
  ],
  "vendor": {
    "name": "The Apache Software Foundation"
  }
}
```

Not all that spectacular. CouchDB is saying "hello" with the running version number.

Next, we can get a list of databases:

```
curl -X GET http://admin:password@127.0.0.1:5984/_all_dbs
```

All we added to the previous request is the _all_dbs string, and our admin user name and password (set when installing CouchDB).

The response should look like:

```
["_replicator","_users"]
```

Note: In case this returns an empty Array for you, it means you haven't finished installation correctly. Please refer to *Setup* for further information on this.

For the purposes of this example, we'll not be showing the system databases past this point. In *your* installation, any time you `GET /_all_dbs`, you should see the system databases in the list, too.

Oh, that's right, we didn't create any user databases yet!

Note: The curl command issues GET requests by default. You can issue POST requests using `curl -X POST`. To make it easy to work with our terminal history, we usually use the `-X` option even when issuing GET requests. If we want to send a POST next time, all we have to change is the method.

HTTP does a bit more under the hood than you can see in the examples here. If you're interested in every last detail that goes over the wire, pass in the `-v` option (e.g., `curl -vX GET`), which will show you the server curl tries to connect to, the request headers it sends, and response headers it receives back. Great for debugging!

Let's create a database:

```
curl -X PUT http://admin:password@127.0.0.1:5984/baseball
```

CouchDB will reply with:

```
{"ok":true}
```

Retrieving the list of databases again shows some useful results this time:

```
curl -X GET http://admin:password@127.0.0.1:5984/_all_dbs
```

```
["baseball"]
```

Note: We should mention JavaScript Object Notation (JSON) here, the data format CouchDB speaks. JSON is a lightweight data interchange format based on JavaScript syntax. Because JSON is natively compatible with JavaScript, your web browser is an ideal client for CouchDB.

Brackets (`[]`) represent ordered lists, and curly braces (`{}`) represent key/value dictionaries. Keys must be strings, delimited by quotes (`"`), and values can be strings, numbers, booleans, lists, or key/value dictionaries. For a more detailed description of JSON, see Appendix E, JSON Primer.

Let's create another database:

```
curl -X PUT http://admin:password@127.0.0.1:5984/baseball
```

CouchDB will reply with:

```
{"error":"file_exists","reason":"The database could not be created,
the file already exists."}
```

We already have a database with that name, so CouchDB will respond with an error. Let's try again with a different database name:

```
curl -X PUT http://admin:password@127.0.0.1:5984/plankton
```

CouchDB will reply with:

```
{"ok":true}
```

Retrieving the list of databases yet again shows some useful results:

```
curl -X GET http://admin:password@127.0.0.1:5984/_all_dbs
```

CouchDB will respond with:

```
["baseball", "plankton"]
```

To round things off, let's delete the second database:

```
curl -X DELETE http://admin:password@127.0.0.1:5984/plankton
```

CouchDB will reply with:

```
{"ok":true}
```

The list of databases is now the same as it was before:

```
curl -X GET http://admin:password@127.0.0.1:5984/_all_dbs
```

CouchDB will respond with:

```
["baseball"]
```

For brevity, we'll skip working with documents, as the next section covers a different and potentially easier way of working with CouchDB that should provide experience with this. As we work through the example, keep in mind that "under the hood" everything is being done by the application exactly as you have been doing here manually. Everything is done using GET, PUT, POST, and DELETE with a URI.

1.6.2 Welcome to Fauxton

After having seen CouchDB's raw API, let's get our feet wet by playing with Fauxton, the built-in administration interface. Fauxton provides full access to all of CouchDB's features and makes it easy to work with some of the more complex ideas involved. With Fauxton we can create and destroy databases; view and edit documents; compose and run MapReduce views; and trigger replication between databases.

To load Fauxton in your browser, visit:

```
http://127.0.0.1:5984/_utils/
```

and log in when prompted with your admin password.

In later documents, we'll focus on using CouchDB from server-side languages such as Ruby and Python. As such, this document is a great opportunity to showcase an example of natively serving up a dynamic web application using nothing more than CouchDB's integrated web server, something you may wish to do with your own applications.

The first thing we should do with a fresh installation of CouchDB is run the test suite to verify that everything is working properly. This assures us that any problems we may run into aren't due to bothersome issues with our

setup. By the same token, failures in the Fauxton test suite are a red flag, telling us to double-check our installation before attempting to use a potentially broken database server, saving us the confusion when nothing seems to be working quite like we expect!

To validate your installation, click on the *Verify* link on the left-hand side, then press the green *Verify Installation* button. All tests should pass with a check mark. If any fail, re-check your installation steps.

1.6.3 Your First Database and Document

Creating a database in Fauxton is simple. From the overview page, click "Create Database." When asked for a name, enter `hello-world` and click the Create button.

After your database has been created, Fauxton will display a list of all its documents. This list will start out empty, so let's create our first document. Click the plus sign next to "All Documents" and select the "New Doc" link. CouchDB will generate a UUID for you.

For demoing purposes, having CouchDB assign a UUID is fine. When you write your first programs, we recommend assigning your own UUIDs. If you rely on the server to generate the UUID and you end up making two POST requests because the first POST request bombed out, you might generate two docs and never find out about the first one because only the second one will be reported back. Generating your own UUIDs makes sure that you'll never end up with duplicate documents.

Fauxton will display the newly created document, with its _id field. To create a new field, simply use the editor to write valid JSON. Add a new field by appending a comma to the `_id` value, then adding the text:

```
"hello": "my new value"
```

Click the green Create Document button to finalize creating the document.

You can experiment with other JSON values; e.g., `[1, 2, "c"]` or `{"foo": "bar"}`.

You'll notice that the document's _rev has been added. We'll go into more detail about this in later documents, but for now, the important thing to note is that _rev acts like a safety feature when saving a document. As long as you and CouchDB agree on the most recent _rev of a document, you can successfully save your changes.

For clarity, you may want to display the contents of the document in the all document view. To enable this, from the upper-right corner of the window, select Options, then check the Include Docs option. Finally, press the Run Query button. The full document should be displayed along with the `_id` and `_rev` values.

1.6.4 Running a Mango Query

Now that we have stored documents successfully, we want to be able to query them. The easiest way to do this in CouchDB is running a Mango Query. There are always two parts to a Mango Query: the index and the selector.

The index specifies which fields we want to be able to query on, and the selector includes the actual query parameters that define what we are looking for exactly.

Indexes are stored as rows that are kept sorted by the fields you specify. This makes retrieving data from a range of keys efficient even when there are thousands or millions of rows.

Before we can run an example query, we'll need some data to run it on. We'll create documents with information about movies. Let's create documents for three movies. (Allow CouchDB to generate the `_id` and `_rev` fields.) Use Fauxton to create documents that have a final JSON structure that look like this:

```
{
    "_id": "00a271787f89c0ef2e10e88a0c0001f4",
    "type": "movie",
    "title": "My Neighbour Totoro",
    "year": 1988,
    "director": "miyazaki",
```

(continues on next page)

(continued from previous page)

```
    "rating": 8.2
}
```

```
{
    "_id": "00a271787f89c0ef2e10e88a0c0003f0",
    "type": "movie",
    "title": "Kikis Delivery Service",
    "year": 1989,
    "director": "miyazaki",
    "rating": 7.8
}
```

```
{
    "_id": "00a271787f89c0ef2e10e88a0c00048b",
    "type": "movie",
    "title": "Princess Mononoke",
    "year": 1997,
    "director": "miyazaki",
    "rating": 8.4
}
```

Now we want to be able to find a movie by its release year, we need to create a Mango Index. To do this, go to "Run A Query with Mango" in the Database overview. Then click on "manage indexes", and change the index field on the left to look like this:

```
{
   "index": {
      "fields": [
         "year"
      ]
   },
   "name": "year-json-index",
   "type": "json"
}
```

This defines an index on the field `year` and allows us to send queries for documents from a specific year.

Next, click on "edit query" and change the Mango Query to look like this:

```
{
   "selector": {
      "year": {
         "$eq": 1988
      }
   }
}
```

Then click on "Run Query".

The result should be a single result, the movie "My Neighbour Totoro" which has the year value of 1988. `$eq` here stands for "equal".

Note: Note that if you skip adding the index, the query will still return the correct results, although you will see a warning about not using a pre-existing index. Not using an index will work fine on small databases and is acceptable for testing out queries in development or training, but we very strongly discourage doing this in any other case, since an index is absolutely vital to good query performance.

You can also query for all movies during the 1980s, with this selector:

```
{
   "selector": {
      "year": {
         "$lt": 1990,
         "$gte": 1980
      }
   }
}
```

The result are the two movies from 1988 and 1989. `$lt` here means "lower than", and `$gte` means "greater than or equal to". The latter currently doesn't have any effect, given that all of our movies are more recent than 1980, but this makes the query future-proof and allows us to add older movies later.

1.6.5 Triggering Replication

Fauxton can trigger replication between two local databases, between a local and remote database, or even between two remote databases. We'll show you how to replicate data from one local database to another, which is a simple way of making backups of your databases as we're working through the examples.

First we'll need to create an empty database to be the target of replication. Return to the Databases overview and create a database called `hello-replication`. Now click "Replication" in the sidebar and choose `hello-world` as the source and `hello-replication` as the target. Click "Replicate" to replicate your database.

To view the result of your replication, click on the Databases tab again. You should see the `hello-replication` database has the same number of documents as the `hello-world` database, and it should take up roughly the same size as well.

Note: For larger databases, replication can take much longer. It is important to leave the browser window open while replication is taking place. As an alternative, you can trigger replication via curl or some other HTTP client that can handle long-running connections. If your client closes the connection before replication finishes, you'll have to retrigger it. Luckily, CouchDB's replication can take over from where it left off instead of starting from scratch.

1.6.6 Wrapping Up

Now that you've seen most of Fauxton's features, you'll be prepared to dive in and inspect your data as we build our example application in the next few documents. Fauxton's pure JavaScript approach to managing CouchDB shows how it's possible to build a fully featured web application using only CouchDB's HTTP API and integrated web server.

But before we get there, we'll have another look at CouchDB's HTTP API – now with a magnifying glass. Let's curl up on the couch and relax.

1.7 The Core API

This document explores the CouchDB in minute detail. It shows all the nitty-gritty and clever bits. We show you best practices and guide you around common pitfalls.

We start out by revisiting the basic operations we ran in the previous document *Getting Started*, looking behind the scenes. We also show what Fauxton needs to do behind its user interface to give us the nice features we saw earlier.

This document is both an introduction to the core CouchDB API as well as a reference. If you can't remember how to run a particular request or why some parameters are needed, you can always come back here and look things up (we are probably the heaviest users of this document).

While explaining the API bits and pieces, we sometimes need to take a larger detour to explain the reasoning for a particular request. This is a good opportunity for us to tell you why CouchDB works the way it does.

The API can be subdivided into the following sections. We'll explore them individually:

- *Server*
- *Databases*
- *Documents*
- *Replication*
- *Wrapping Up*

1.7.1 Server

This one is basic and simple. It can serve as a sanity check to see if CouchDB is running at all. It can also act as a safety guard for libraries that require a certain version of CouchDB. We're using the curl utility again:

```
curl http://127.0.0.1:5984/
```

CouchDB replies, all excited to get going:

```
{
  "couchdb": "Welcome",
  "version": "3.0.0",
  "git_sha": "83bdcf693",
  "uuid": "56f16e7c93ff4a2dc20eb6acc7000b71",
  "features": [
    "access-ready",
    "partitioned",
    "pluggable-storage-engines",
    "reshard",
    "scheduler"
  ],
  "vendor": {
    "name": "The Apache Software Foundation"
  }
}
```

You get back a JSON string, that, if parsed into a native object or data structure of your programming language, gives you access to the welcome string and version information.

This is not terribly useful, but it illustrates nicely the way CouchDB behaves. You send an HTTP request and you receive a JSON string in the HTTP response as a result.

1.7.2 Databases

Now let's do something a little more useful: *create databases*. For the strict, CouchDB is a *database management system* (DMS). That means it can hold multiple databases. A database is a bucket that holds "related data". We'll explore later what that means exactly. In practice, the terminology is overlapping – often people refer to a DMS as "a database" and also a database within the DMS as "a database." We might follow that slight oddity, so don't get confused by it. In general, it should be clear from the context if we are talking about the whole of CouchDB or a single database within CouchDB.

Now let's make one! We want to store our favorite music albums, and we creatively give our database the name albums. Note that we're now using the `-X` option again to tell curl to send a PUT request instead of the default GET request:

```
curl -X PUT http://admin:password@127.0.0.1:5984/albums
```

CouchDB replies:

```
{"ok":true}
```

That's it. You created a database and CouchDB told you that all went well. What happens if you try to create a database that already exists? Let's try to create that database again:

```
curl -X PUT http://admin:password@127.0.0.1:5984/albums
```

CouchDB replies:

```
{"error":"file_exists","reason":"The database could not be created, the file already␣
↪exists."}
```

We get back an error. This is pretty convenient. We also learn a little bit about how CouchDB works. CouchDB stores each database in a single file. Very simple.

Let's create another database, this time with curl's `-v` (for "verbose") option. The verbose option tells curl to show us not only the essentials – the HTTP response body – but all the underlying request and response details:

```
curl -vX PUT http://admin:password@127.0.0.1:5984/albums-backup
```

curl elaborates:

```
* About to connect() to 127.0.0.1 port 5984 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 5984 (#0)
> PUT /albums-backup HTTP/1.1
> User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3 OpenSSL/0.9.7l␣
↪zlib/1.2.3
> Host: 127.0.0.1:5984
> Accept: */*
>
< HTTP/1.1 201 Created
< Server: CouchDB (Erlang/OTP)
< Date: Sun, 05 Jul 2009 22:48:28 GMT
< Content-Type: text/plain;charset=utf-8
< Content-Length: 12
< Cache-Control: must-revalidate
<
{"ok":true}
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

What a mouthful. Let's step through this line by line to understand what's going on and find out what's important. Once you've seen this output a few times, you'll be able to spot the important bits more easily.

```
* About to connect() to 127.0.0.1 port 5984 (#0)
```

This is curl telling us that it is going to establish a TCP connection to the CouchDB server we specified in our request URI. Not at all important, except when debugging networking issues.

```
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 5984 (#0)
```

curl tells us it successfully connected to CouchDB. Again, not important if you aren't trying to find problems with your network.

The following lines are prefixed with > and < characters. The > means the line was sent to CouchDB verbatim (without the actual >). The < means the line was sent back to curl by CouchDB.

```
> PUT /albums-backup HTTP/1.1
```

This initiates an HTTP request. Its *method* is PUT, the *URI* is `/albums-backup`, and the HTTP version is `HTTP/1.1`. There is also `HTTP/1.0`, which is simpler in some cases, but for all practical reasons you should be using `HTTP/1.1`.

Next, we see a number of *request headers*. These are used to provide additional details about the request to CouchDB.

```
> User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3 OpenSSL/0.9.7l␣
↪zlib/1.2.3
```

The User-Agent header tells CouchDB which piece of client software is doing the HTTP request. We don't learn anything new: it's curl. This header is often useful in web development when there are known errors in client implementations that a server might want to prepare the response for. It also helps to determine which platform a user is on. This information can be used for technical and statistical reasons. For CouchDB, the User-Agent header is irrelevant.

```
> Host: 127.0.0.1:5984
```

The Host header is required by `HTTP 1.1`. It tells the server the hostname that came with the request.

```
> Accept: */*
```

The Accept header tells CouchDB that curl accepts any media type. We'll look into why this is useful a little later.

```
>
```

An empty line denotes that the request headers are now finished and the rest of the request contains data we're sending to the server. In this case, we're not sending any data, so the rest of the curl output is dedicated to the HTTP response.

```
< HTTP/1.1 201 Created
```

The first line of CouchDB's HTTP response includes the HTTP version information (again, to acknowledge that the requested version could be processed), an HTTP *status code*, and a *status code message*. Different requests trigger different response codes. There's a whole range of them telling the client (curl in our case) what effect the request had on the server. Or, if an error occurred, what kind of error. **RFC 2616** (the HTTP 1.1 specification) defines clear behavior for response codes. CouchDB fully follows the RFC.

The 201 Created status code tells the client that the resource the request was made against was successfully created. No surprise here, but if you remember that we got an error message when we tried to create this database twice, you now know that this response could include a different response code. Acting upon responses based on response codes is a common practice. For example, all response codes of 400 Bad Request or larger tell you that some error occurred. If you want to shortcut your logic and immediately deal with the error, you could just check a `>= 400` response code.

```
< Server: CouchDB (Erlang/OTP)
```

The Server header is good for diagnostics. It tells us which CouchDB version and which underlying Erlang version we are talking to. In general, you can ignore this header, but it is good to know it's there if you need it.

```
< Date: Sun, 05 Jul 2009 22:48:28 GMT
```

The Date header tells you the time of the server. Since client and server time are not necessarily synchronized, this header is purely informational. You shouldn't build any critical application logic on top of this!

```
< Content-Type: text/plain;charset=utf-8
```

The Content-Type header tells you which MIME type the HTTP response body is and its encoding. We already know CouchDB returns JSON strings. The appropriate Content-Type header is *application/json*. Why do we see *text/plain*? This is where pragmatism wins over purity. Sending an *application/json* Content-Type header will make a browser offer you the returned JSON for download instead of just displaying it. Since it is extremely useful to be able to test CouchDB from a browser, CouchDB sends a *text/plain* content type, so all browsers will display the JSON as text.

Note: There are some extensions that make your browser JSON-aware, but they are not installed by default. For more information, look at the popular JSONView extension, available for both Firefox and Chrome.

Do you remember the Accept request header and how it is set to `*/*` to express interest in any MIME type? If you send `Accept: application/json` in your request, CouchDB knows that you can deal with a pure JSON response with the proper Content-Type header and will use it instead of *text/plain*.

```
< Content-Length: 12
```

The Content-Length header simply tells us how many bytes the response body has.

```
< Cache-Control: must-revalidate
```

This Cache-Control header tells you, or any proxy server between CouchDB and you, not to cache this response.

```
<
```

This empty line tells us we're done with the response headers and what follows now is the response body.

```
{"ok":true}
```

We've seen this before.

```
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

The last two lines are curl telling us that it kept the TCP connection it opened in the beginning open for a moment, but then closed it after it received the entire response.

Throughout the documents, we'll show more requests with the `-v` option, but we'll omit some of the headers we've seen here and include only those that are important for the particular request.

Creating databases is all fine, but how do we get rid of one? Easy – just change the HTTP method:

```
> curl -vX DELETE http://admin:password@127.0.0.1:5984/albums-backup
```

This deletes a CouchDB database. The request will remove the file that the database contents are stored in. There is no *"Are you sure?"* safety net or any *"Empty the trash"* magic you've got to do to delete a database. Use this command with care. Your data will be deleted without a chance to bring it back easily if you don't have a backup copy.

This section went knee-deep into HTTP and set the stage for discussing the rest of the core CouchDB API. Next stop: documents.

1.7.3 Documents

Documents are CouchDB's central data structure. The idea behind a document is, unsurprisingly, that of a real-world document – a sheet of paper such as an invoice, a recipe, or a business card. We already learned that CouchDB uses the JSON format to store documents. Let's see how this storing works at the lowest level.

Each document in CouchDB has an *ID*. This ID is unique per database. You are free to choose any string to be the ID, but for best results we recommend a UUID (or GUID), i.e., a Universally (or Globally) Unique IDentifier. UUIDs are random numbers that have such a low collision probability that everybody can make thousands of UUIDs a minute for millions of years without ever creating a duplicate. This is a great way to ensure two independent people cannot create two different documents with the same ID. Why should you care what somebody else is doing? For one, that somebody else could be you at a later time or on a different computer; secondly, CouchDB replication lets you share documents with others and using UUIDs ensures that it all works. But more on that later; let's make some documents:

```
curl -X PUT http://admin:password@127.0.0.1:5984/albums/
↪6e1295ed6c29495e54cc05947f18c8af -d '{"title":"There is Nothing Left to Lose",
↪"artist":"Foo Fighters"}'
```

CouchDB replies:

```
{"ok":true,"id":"6e1295ed6c29495e54cc05947f18c8af","rev":"1-2902191555"}
```

The curl command appears complex, but let's break it down. First, `-X PUT` tells curl to make a PUT request. It is followed by the URL that specifies your CouchDB IP address and port. The resource part of the URL `/albums/6e1295ed6c29495e54cc05947f18c8af` specifies the location of a document inside our albums database. The wild collection of numbers and characters is a UUID. This UUID is your document's ID. Finally, the `-d` flag tells curl to use the following string as the body for the PUT request. The string is a simple JSON structure including `title` and `artist` attributes with their respective values.

Note: If you don't have a UUID handy, you can ask CouchDB to give you one (in fact, that is what we did just now without showing you). Simply send a *GET /_uuids* request:

```
curl -X GET http://127.0.0.1:5984/_uuids
```

CouchDB replies:

```
{"uuids":["6e1295ed6c29495e54cc05947f18c8af"]}
```

Voilà, a UUID. If you need more than one, you can pass in the `?count=10` HTTP parameter to request 10 UUIDs, or really, any number you need.

To double-check that CouchDB isn't lying about having saved your document (it usually doesn't), try to retrieve it by sending a GET request:

```
curl -X GET http://admin:password@127.0.0.1:5984/albums/
↪6e1295ed6c29495e54cc05947f18c8af
```

We hope you see a pattern here. Everything in CouchDB has an address, a URI, and you use the different HTTP methods to operate on these URIs.

CouchDB replies:

```
{"_id":"6e1295ed6c29495e54cc05947f18c8af","_rev":"1-2902191555","title":"There is␣
↪Nothing Left to Lose","artist":"Foo Fighters"}
```

This looks a lot like the document you asked CouchDB to save, which is good. But you should notice that CouchDB added two fields to your JSON structure. The first is `_id`, which holds the UUID we asked CouchDB to save our document under. We always know the ID of a document if it is included, which is very convenient.

The second field is `_rev`. It stands for *revision*.

Revisions

If you want to change a document in CouchDB, you don't tell it to go and find a field in a specific document and insert a new value. Instead, you load the full document out of CouchDB, make your changes in the JSON structure (or object, when you are doing actual programming), and save the entire new revision (or version) of that document back into CouchDB. Each revision is identified by a new `_rev` value.

If you want to update or delete a document, CouchDB expects you to include the `_rev` field of the revision you wish to change. When CouchDB accepts the change, it will generate a new revision number. This mechanism ensures that, in case somebody else made a change without you knowing before you got to request the document update, CouchDB will not accept your update because you are likely to overwrite data you didn't know existed. Or simplified: whoever saves a change to a document first, wins. Let's see what happens if we don't provide a `_rev` field (which is equivalent to providing a outdated value):

```
curl -X PUT http://admin:password@127.0.0.1:5984/albums/
↪6e1295ed6c29495e54cc05947f18c8af \
     -d '{"title":"There is Nothing Left to Lose","artist":"Foo Fighters","year":"1997
↪"}'
```

CouchDB replies:

```
{"error":"conflict","reason":"Document update conflict."}
```

If you see this, add the latest revision number of your document to the JSON structure:

```
curl -X PUT http://admin:password@127.0.0.1:5984/albums/
↪6e1295ed6c29495e54cc05947f18c8af \
     -d '{"_rev":"1-2902191555","title":"There is Nothing Left to Lose","artist":"Foo␣
↪Fighters","year":"1997"}'
```

Now you see why it was handy that CouchDB returned that `_rev` when we made the initial request. CouchDB replies:

```
{"ok":true,"id":"6e1295ed6c29495e54cc05947f18c8af","rev":"2-
↪8aff9ee9d06671fa89c99d20a4b3ae"}
```

CouchDB accepted your write and also generated a new revision number. The revision number is the *MD5 hash* of the transport representation of a document with an `N-` prefix denoting the number of times a document got updated. This is useful for replication. See *Replication and conflict model* for more information.

There are multiple reasons why CouchDB uses this revision system, which is also called Multi-Version Concurrency Control (MVCC). They all work hand-in-hand, and this is a good opportunity to explain some of them.

One of the aspects of the HTTP protocol that CouchDB uses is that it is stateless. What does that mean? When talking to CouchDB you need to make requests. Making a request includes opening a network connection to CouchDB, exchanging bytes, and closing the connection. This is done every time you make a request. Other protocols allow you to open a connection, exchange bytes, keep the connection open, exchange more bytes later – maybe depending on the bytes you exchanged at the beginning – and eventually close the connection. Holding a connection open for later use requires the server to do extra work. One common pattern is that for the lifetime of a connection, the client has a consistent and static view of the data on the server. Managing huge amounts of parallel connections is a significant amount of work. HTTP connections are usually short-lived, and making the same guarantees is a lot easier. As a result, CouchDB can handle many more concurrent connections.

Another reason CouchDB uses MVCC is that this model is simpler conceptually and, as a consequence, easier to program. CouchDB uses less code to make this work, and less code is always good because the ratio of defects per lines of code is static.

The revision system also has positive effects on replication and storage mechanisms, but we'll explore these later in the documents.

Warning: The terms *version* and *revision* might sound familiar (if you are programming without version control, stop reading this guide right now and start learning one of the popular systems). Using new versions for document changes works a lot like version control, but there's an important difference: **CouchDB does not guarantee that older versions are kept around. Don't use the ``_rev`` token in CouchDB as a revision control system for your documents.**

Documents in Detail

Now let's have a closer look at our document creation requests with the curl `-v` flag that was helpful when we explored the database API earlier. This is also a good opportunity to create more documents that we can use in later examples.

We'll add some more of our favorite music albums. Get a fresh UUID from the `/_uuids` resource. If you don't remember how that works, you can look it up a few pages back.

```
curl -vX PUT http://admin:password@127.0.0.1:5984/albums/
↪70b50bfa0a4b3aed1f8aff9e92dc16a0 \
    -d '{"title":"Blackened Sky","artist":"Biffy Clyro","year":2002}'
```

Note: By the way, if you happen to know more information about your favorite albums, don't hesitate to add more properties. And don't worry about not knowing all the information for all the albums. CouchDB's schema-less documents can contain whatever you know. After all, you should relax and not worry about data.

Now with the `-v` option, CouchDB's reply (with only the important bits shown) looks like this:

```
> PUT /albums/70b50bfa0a4b3aed1f8aff9e92dc16a0 HTTP/1.1
>
< HTTP/1.1 201 Created
< Location: http://127.0.0.1:5984/albums/70b50bfa0a4b3aed1f8aff9e92dc16a0
< ETag: "1-e89c99d29d06671fa0a4b3ae8aff9e"
<
{"ok":true,"id":"70b50bfa0a4b3aed1f8aff9e92dc16a0","rev":"1-
↪e89c99d29d06671fa0a4b3ae8aff9e"}
```

We're getting back the 201 Created HTTP status code in the response headers, as we saw earlier when we created a database. The Location header gives us a full URL to our newly created document. And there's a new header. An ETag in HTTP-speak identifies a specific version of a resource. In this case, it identifies a specific version (the first one) of our new document. Sound familiar? Yes, conceptually, an ETag is the same as a CouchDB document revision number, and it shouldn't come as a surprise that CouchDB uses revision numbers for ETags. ETags are useful for caching infrastructures.

Attachments

CouchDB documents can have attachments just like an email message can have attachments. An attachment is identified by a name and includes its MIME type (or Content-Type) and the number of bytes the attachment contains. Attachments can be any data. It is easiest to think about attachments as files attached to a document. These files can be text, images, Word documents, music, or movie files. Let's make one.

Attachments get their own URL where you can upload data. Say we want to add the album artwork to the `6e1295ed6c29495e54cc05947f18c8af` document (*"There is Nothing Left to Lose"*), and let's also say the artwork is in a file *artwork.jpg* in the current directory:

```
curl -vX PUT http://admin:password@127.0.0.1:5984/albums/
↪6e1295ed6c29495e54cc05947f18c8af/artwork.jpg?rev=2-2739352689 \
    --data-binary @artwork.jpg -H "Content-Type:image/jpg"
```

Note: The `--data-binary @` option tells curl to read a file's contents into the HTTP request body. We're using the `-H` option to tell CouchDB that we're uploading a JPEG file. CouchDB will keep this information around and will send the appropriate header when requesting this attachment; in case of an image like this, a browser will render the image instead of offering you the data for download. This will come in handy later. Note that you need to provide the current revision number of the document you're attaching the artwork to, just as if you would update the document. Because, after all, attaching some data is changing the document.

You should now see your artwork image if you point your browser to http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af/artwork.jpg

If you request the document again, you'll see a new member:

```
curl http://admin:password@127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af
```

CouchDB replies:

```
{
    "_id": "6e1295ed6c29495e54cc05947f18c8af",
    "_rev": "3-131533518",
    "title": "There is Nothing Left to Lose",
    "artist": "Foo Fighters",
    "year": "1997",
    "_attachments": {
        "artwork.jpg": {
            "stub": true,
            "content_type": "image/jpg",
            "length": 52450
        }
    }
}
```

`_attachments` is a list of keys and values where the values are JSON objects containing the attachment metadata. `stub=true` tells us that this entry is just the metadata. If we use the `?attachments=true` HTTP option when requesting this document, we'd get a Base64 encoded string containing the attachment data.

We'll have a look at more document request options later as we explore more features of CouchDB, such as replication, which is the next topic.

1.7.4 Replication

CouchDB replication is a mechanism to synchronize databases. Much like rsync synchronizes two directories locally or over a network, replication synchronizes two databases locally or remotely.

In a simple POST request, you tell CouchDB the *source* and the *target* of a replication and CouchDB will figure out which documents and new document revisions are on *source* that are not yet on *target*, and will proceed to move the missing documents and revisions over.

We'll take an in-depth look at replication in the document *Introduction to Replication*; in this document, we'll just show you how to use it.

First, we'll create a target database. Note that CouchDB won't automatically create a target database for you, and will return a replication failure if the target doesn't exist (likewise for the source, but that mistake isn't as easy to make):

```
curl -X PUT http://admin:password@127.0.0.1:5984/albums-replica
```

Now we can use the database *albums-replica* as a replication target:

```
curl -X POST http://admin:password@127.0.0.1:5984/_replicate \
     -d '{"source":"http://admin:password@127.0.0.1:5984/albums","target":"http://
↪admin:password@127.0.0.1:5984/albums-replica"}' \
     -H "Content-Type: application/json"
```

Note: As of CouchDB 2.0.0, fully qualified URLs are required for both the replication `source` and `target` parameters.

Note: CouchDB supports the option `"create_target":true` placed in the JSON POSTed to the *_replicate* URL. It implicitly creates the target database if it doesn't exist.

CouchDB replies (this time we formatted the output so you can read it more easily):

```
{
    "ok": true,
    "session_id": "30bb4ac013ca69369c0f32be78864d6e",
    "source_last_seq": "2-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYExlzgQLsBiaphqYpSZjKcRqRxwIkGRqA1H8Uk4wszJIskg0wdWUBAFHwJD4",
    "replication_id_version": 4,
    "history": [
        {
            "session_id": "30bb4ac013ca69369c0f32be78864d6e",
            "start_time": "Sun, 05 Mar 2023 20:30:26 GMT",
            "end_time": "Sun, 05 Mar 2023 20:30:29 GMT",
            "start_last_seq": 0,
            "end_last_seq": "2-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9___
↪_szKYExlzgQLsBiaphqYpSZjKcRqRxwIkGRqA1H8Uk4wszJIskg0wdWUBAFHwJD4",
            "recorded_seq": "2-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9___
↪_szKYExlzgQLsBiaphqYpSZjKcRqRxwIkGRqA1H8Uk4wszJIskg0wdWUBAFHwJD4",
            "missing_checked": 2,
            "missing_found": 2,
            "docs_read": 2,
            "docs_written": 2,
            "doc_write_failures": 0,
            "bulk_get_docs": 2,
            "bulk_get_attempts": 2
```

(continues on next page)

(continued from previous page)

```
        }
    ]
}
```

CouchDB maintains a *session history* of replications. The response for a replication request contains the history entry for this *replication session*. It is also worth noting that the request for replication will stay open until replication closes. If you have a lot of documents, it'll take a while until they are all replicated and you won't get back the replication response until all documents are replicated. It is important to note that replication replicates the database only as it was at the point in time when replication was started. So, any additions, modifications, or deletions subsequent to the start of replication will not be replicated.

We'll punt on the details again – the `"ok": true` at the beginning tells us all went well. If you now have a look at the albums-replica database, you should see all the documents that you created in the albums database. Neat, eh?

What you just did is called local replication in CouchDB terms. You created a local copy of a database. This is useful for backups or to keep snapshots of a specific state of your data around for later. You might want to do this if you are developing your applications but want to be able to roll back to a stable version of your code and data.

There are more types of replication useful in other situations. The source and target members of our replication request are actually links (like in HTML) and so far we've seen links relative to the server we're working on (hence local). You can also specify a remote database as the target:

```
curl -X POST http://admin:password@127.0.0.1:5984/_replicate \
     -d '{"source":"http://admin:password@127.0.0.1:5984/albums","target":"http://
↪user:password@example.org:5984/albums-replica"}' \
     -H "Content-Type:application/json"
```

Using a *local source* and a *remote target* database is called *push replication*. We're pushing changes to a remote server.

Note: Since we don't have a second CouchDB server around just yet, we'll just use the absolute address of our single server, but you should be able to infer from this that you can put any remote server in there.

This is great for sharing local changes with remote servers or buddies next door.

You can also use a *remote source* and a *local target* to do a *pull replication*. This is great for getting the latest changes from a server that is used by others:

```
curl -X POST http://admin:password@127.0.0.1:5984/_replicate \
     -d '{"source":"http://user:password@example.org:5984/albums-replica","target":
↪"http://admin:password@127.0.0.1:5984/albums"}' \
     -H "Content-Type:application/json"
```

Finally, you can run remote replication, which is mostly useful for management operations:

```
curl -X POST http://admin:password@127.0.0.1:5984/_replicate \
     -d '{"source":"http://user:password@example.org:5984/albums","target":"http://
↪user:password@example.org:5984/albums-replica"}' \
     -H "Content-Type: application/json"
```

Note: CouchDB and REST

CouchDB prides itself on having a RESTful API, but these replication requests don't look very RESTy to the trained eye. What's up with that? While CouchDB's core database, document, and attachment API are RESTful, not all of CouchDB's API is. The replication API is one example. There are more, as we'll see later in the documents.

Why are there RESTful and non-RESTful APIs mixed up here? Have the developers been too lazy to go REST all the way? Remember, REST is an architectural style that lends itself to certain architectures (such as the CouchDB

document API). But it is not a one-size-fits-all. Triggering an event like replication does not make a whole lot of sense in the REST world. It is more like a traditional remote procedure call. And there is nothing wrong with this.

We very much believe in the "use the right tool for the job" philosophy, and REST does not fit every job. For support, we refer to Leonard Richardson and Sam Ruby who wrote RESTful Web Services (O'Reilly), as they share our view.

1.7.5 Wrapping Up

This is still not the full CouchDB API, but we discussed the essentials in great detail. We're going to fill in the blanks as we go. For now, we believe you're ready to start building CouchDB applications.

See also:

Complete HTTP API Reference:

- *Server API Reference*
- *Database API Reference*
- *Document API Reference*
- *Replication API*

CHAPTER

TWO

REPLICATION

Replication is an incremental one way process involving two databases (a source and a destination).

The aim of replication is that at the end of the process, all active documents in the source database are also in the destination database and all documents that were deleted in the source database are also deleted in the destination database (if they even existed).

The replication process only copies the last revision of a document, so all previous revisions that were only in the source database are not copied to the destination database.

2.1 Introduction to Replication

One of CouchDB's strengths is the ability to synchronize two copies of the same database. This enables users to distribute data across several nodes or data centers, but also to move data more closely to clients.

Replication involves a source and a destination database, which can be on the same or on different CouchDB instances. The aim of replication is that at the end of the process, all active documents in the source database are also in the destination database and all documents that were deleted in the source database are also deleted in the destination database (if they even existed).

2.1.1 Transient and Persistent Replication

There are two different ways to set up a replication. The first one that was introduced into CouchDB leads to a replication that could be called *transient*. Transient means that there are no documents backing up the replication. So after a restart of the CouchDB server the replication will disappear. Later, the *_replicator* database was introduced, which keeps documents containing your replication parameters. Such a replication can be called *persistent*. Transient replications were kept for backward compatibility. Both replications can have different *replication states*.

2.1.2 Triggering, Stopping and Monitoring Replications

A persistent replication is controlled through a document in the *_replicator* database, where each document describes one replication process (see *Replication Settings*). For setting up a transient replication the api endpoint */_replicate* can be used. A replication is triggered by sending a JSON object either to the `_replicate` endpoint or storing it as a document into the `_replicator` database.

If a replication is currently running its status can be inspected through the active tasks API (see */_active_tasks*, *Replication Status* and */_scheduler/jobs*).

For document based-replications, */_scheduler/docs* can be used to get a complete state summary. This API is preferred as it will show the state of the replication document before it becomes a replication job.

For transient replications there is no way to query their state when the job is finished.

A replication can be stopped by deleting the document, or by updating it with its `cancel` property set to `true`.

2.1.3 Replication Procedure

During replication, CouchDB will compare the source and the destination database to determine which documents differ between the source and the destination database. It does so by following the *Changes Feeds* on the source and comparing the documents to the destination. Changes are submitted to the destination in batches where they can introduce conflicts. Documents that already exist on the destination in the same revision are not transferred. As the deletion of documents is represented by a new revision, a document deleted on the source will also be deleted on the target.

A replication task will finish once it reaches the end of the changes feed. If its `continuous` property is set to true, it will wait for new changes to appear until the task is canceled. Replication tasks also create checkpoint documents on the destination to ensure that a restarted task can continue from where it stopped, for example after it has crashed.

When a replication task is initiated on the sending node, it is called *push* replication, if it is initiated by the receiving node, it is called *pull* replication.

2.1.4 Master - Master replication

One replication task will only transfer changes in one direction. To achieve master-master replication, it is possible to set up two replication tasks in opposite direction. When a change is replicated from database A to B by the first task, the second task from B to A will discover that the new change on B already exists in A and will wait for further changes.

2.1.5 Controlling which Documents to Replicate

There are three options for controlling which documents are replicated, and which are skipped:

1. Defining documents as being local.
2. Using *Selector Objects*.
3. Using *Filter Functions*.

Local documents are never replicated (see *Local (non-replicating) Documents*).

Selector Objects can be included in a replication document (see *Replication Settings*). A selector object contains a query expression that is used to test whether a document should be replicated.

Filter Functions can be used in a replication (see *Replication Settings*). The replication task evaluates the filter function for each document in the changes feed. The document is only replicated if the filter returns `true`.

Note: Using a selector provides performance benefits when compared with using a *Filter Functions*. You should use *Selector Objects* where possible.

Note: When using replication filters that depend on the document's content, deleted documents may pose a problem, since the document passed to the filter will not contain any of the document's content. This can be resolved by adding a `_deleted:true` field to the document instead of using the DELETE HTTP method, paired with the use of a *validate document update* handler to ensure the fields required for replication filters are always present. Take note, though, that the deleted document will still contain all of its data (including attachments)!

2.1.6 Migrating Data to Clients

Replication can be especially useful for bringing data closer to clients. PouchDB implements the replication algorithm of CouchDB in JavaScript, making it possible to make data from a CouchDB database available in an offline browser application, and synchronize changes back to CouchDB.

2.2 Replicator Database

Changed in version 2.1.0: Scheduling replicator was introduced. Replication states, by default are not written back to documents anymore. There are new replication job states and new API endpoints `_scheduler/jobs` and `_scheduler/docs`.

Changed in version 3.2.0: Fair share scheduling was introduced. Multiple `_replicator` databases get an equal chance (configurable) of running their jobs. Previously replication jobs were scheduled without any regard of their originating database.

Changed in version 3.3.0: `winning_revs_only: true` replicator option to replicate the winning document revisions.

The `_replicator` database works like any other in CouchDB, but documents added to it will trigger replications. Create (`PUT` or `POST`) a document to start replication. `DELETE` a replication document to cancel an ongoing replication.

These documents have exactly the same content as the JSON objects we used to `POST` to `_replicate` (fields `source`, `target`, `create_target`, `create_target_params`, `continuous`, `doc_ids`, `filter`, `query_params`, `use_checkpoints`, `checkpoint_interval`).

Replication documents can have a user defined `_id` (handy for finding a specific replication request later). Design Documents (and `_local` documents) added to the replicator database are ignored.

The default replicator database is `_replicator`. Additional replicator databases can be created. To be recognized as such by the system, their database names should end with `/_replicator`.

2.2.1 Basics

Let's say you POST the following document into `_replicator`:

```
{
    "_id": "my_rep",
    "source": "http://user:password@myserver.com/foo",
    "target": {
        "url": "http://localhost:5984/bar",
        "auth": {
            "basic": {
                "username": "adm",
                "password": "pass"
            }
        }
    },
    "create_target": true,
    "continuous": true
}
```

In the couch log you'll see 2 entries like these:

```
[notice] 2017-04-05T17:16:19.646716Z node1@127.0.0.1 <0.29432.0> -------- Replication␣
↪`"a81a78e822837e66df423d54279c15fe+continuous+create_target"` is using:
    4 worker processes
```

(continues on next page)

(continued from previous page)

```
    a worker batch size of 500
    20 HTTP connections
    a connection timeout of 30000 milliseconds
    10 retries per request
    socket options are: [{keepalive,true},{nodelay,false}]
[notice] 2017-04-05T17:16:19.646759Z node1@127.0.0.1 <0.29432.0> -------- Document␣
↪`my_rep` triggered replication `a81a78e822837e66df423d54279c15fe+continuous+create_
↪target`
```

Replication state of this document can then be queried from `http://adm:pass@localhost:5984/_scheduler/docs/_replicator/my_rep`

```
{
    "database": "_replicator",
    "doc_id": "my_rep",
    "error_count": 0,
    "id": "a81a78e822837e66df423d54279c15fe+continuous+create_target",
    "info": {
        "revisions_checked": 113,
        "missing_revisions_found": 113,
        "docs_read": 113,
        "docs_written": 113,
        "changes_pending": 0,
        "doc_write_failures": 0,
        "checkpointed_source_seq": "113-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ",
        "source_seq": "113-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ",
        "through_seq": "113-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ"
    },
    "last_updated": "2017-04-05T19:18:15Z",
    "node": "node1@127.0.0.1",
    "source_proxy": null,
    "target_proxy": null,
    "source": "http://myserver.com/foo/",
    "start_time": "2017-04-05T19:18:15Z",
    "state": "running",
    "target": "http://localhost:5984/bar/"
}
```

The state is `running`. That means replicator has scheduled this replication job to run. Replication document contents stay the same. Previously, before version 2.1, it was updated with the `triggered` state.

The replication job will also appear in

`http://adm:pass@localhost:5984/_scheduler/jobs`

```
{
    "jobs": [
        {
            "database": "_replicator",
            "doc_id": "my_rep",
            "history": [
                {
                    "timestamp": "2017-04-05T19:18:15Z",
                    "type": "started"
```

(continues on next page)

(continued from previous page)

```
                },
                {
                    "timestamp": "2017-04-05T19:18:15Z",
                    "type": "added"
                }
            ],
            "id": "a81a78e822837e66df423d54279c15fe+continuous+create_target",
            "info": {
                "changes_pending": 0,
                "checkpointed_source_seq": "113-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ",
                "doc_write_failures": 0,
                "docs_read": 113,
                "docs_written": 113,
                "missing_revisions_found": 113,
                "revisions_checked": 113,
                "source_seq": "113-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ",
                "through_seq": "113-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ"
            },
            "node": "node1@127.0.0.1",
            "pid": "<0.1174.0>",
            "source": "http://myserver.com/foo/",
            "start_time": "2017-04-05T19:18:15Z",
            "target": "http://localhost:5984/bar/",
            "user": null
        }
    ],
    "offset": 0,
    "total_rows": 1
}
```

`_scheduler/jobs` shows more information, such as a detailed history of state changes. If a persistent replication has not yet started, has failed, or is completed, information about its state can only be found in `_scheduler/docs`. Keep in mind that some replication documents could be invalid and could not become a replication job. Others might be delayed because they are fetching data from a slow source database.

If there is an error, for example if the source database is missing, the replication job will crash and retry after a wait period. Each successive crash will result in a longer waiting period.

For example, POST-ing this document

```
{
    "_id": "my_rep_crashing",
    "source": "http://user:password@myserver.com/missing",
    "target": {
        "url": "http://localhost:5984/bar",
        "auth": {
            "basic": {
                "username": "adm",
                "password": "pass"
            }
        }
```

(continues on next page)

(continued from previous page)

```
    },
    "create_target": true,
    "continuous": true
}
```

when source database is missing, will result in periodic starts and crashes with an increasingly larger interval. The `history` list from `_scheduler/jobs` for this replication would look something like this:

```
[
      {
          "reason": "db_not_found: could not open http://adm:*****@localhost:5984/
↪missing/",
          "timestamp": "2017-04-05T20:55:10Z",
          "type": "crashed"
      },
      {
          "timestamp": "2017-04-05T20:55:10Z",
          "type": "started"
      },
      {
          "reason": "db_not_found: could not open http://adm:*****@localhost:5984/
↪missing/",
          "timestamp": "2017-04-05T20:47:10Z",
          "type": "crashed"
      },
      {
          "timestamp": "2017-04-05T20:47:10Z",
          "type": "started"
      }
]
```

`_scheduler/docs` shows a shorter summary:

```
{
      "database": "_replicator",
      "doc_id": "my_rep_crashing",
      "error_count": 6,
      "id": "cb78391640ed34e9578e638d9bb00e44+create_target",
      "info": {
           "error": "db_not_found: could not open http://myserver.com/missing/"
      },
      "last_updated": "2017-04-05T20:55:10Z",
      "node": "node1@127.0.0.1",
      "source_proxy": null,
      "target_proxy": null,
      "source": "http://myserver.com/missing/",
      "start_time": "2017-04-05T20:38:34Z",
      "state": "crashing",
      "target": "http://localhost:5984/bar/"
}
```

Repeated crashes are described as a `crashing` state. `-ing` suffix implies this is a temporary state. User at any moment could create the missing database and then replication job could return back to the normal.

2.2.2 Documents describing the same replication

Let's suppose 2 documents are added to the `_replicator` database in the following order:

```
{
    "_id": "my_rep",
    "source": "http://user:password@myserver.com/foo",
    "target": "http://adm:pass@localhost:5984/bar",
    "create_target": true,
    "continuous": true
}
```

and

```
{
    "_id": "my_rep_dup",
    "source": "http://user:password@myserver.com/foo",
    "target": "http://adm:pass@localhost:5984/bar",
    "create_target": true,
    "continuous": true
}
```

Both describe exactly the same replication (only their `_ids` differ). In this case document `my_rep` triggers the replication, while `my_rep_dup` will fail. Inspecting `_scheduler/docs` explains exactly why it failed:

```
{
    "database": "_replicator",
    "doc_id": "my_rep_dup",
    "error_count": 1,
    "id": null,
    "info": {
        "error": "Replication `a81a78e822837e66df423d54279c15fe+continuous+create_
↪target` specified by document `my_rep_dup` already started, triggered by document␣
↪`my_rep` from db `_replicator`"
    },
    "last_updated": "2017-04-05T21:41:51Z",
    "source": "http://myserver.com/foo/",
    "start_time": "2017-04-05T21:41:51Z",
    "state": "failed",
    "target": "http://user:****@localhost:5984/bar",
}
```

Notice the state for this replication is `failed`. Unlike `crashing`, `failed` state is terminal. As long as both documents are present the replicator will not retry to run `my_rep_dup` replication. Another reason could be malformed documents. For example if worker process count is specified as a string (`"worker_processes": "a few"`) instead of an integer, failure will occur.

2.2.3 Replication Scheduler

Once replication jobs are created they are managed by the scheduler. The scheduler is the replication component which periodically stops some jobs and starts others. This behavior makes it possible to have a larger number of jobs than the cluster could run simultaneously. Replication jobs which keep failing will be penalized and forced to wait. The wait time increases exponentially with each consecutive failure.

When deciding which jobs to stop and which to start, the scheduler uses a round-robin algorithm to ensure fairness. Jobs which have been running the longest time will be stopped, and jobs which have been waiting the longest time will be started.

Note: Non-continuous (normal) replication are treated differently once they start running. See *Normal vs Continuous Replications* section for more information.

The behavior of the scheduler can configured via `max_jobs`, `interval` and `max_churn` options. See *Replicator configuration section* for additional information.

2.2.4 Replication states

Replication jobs during their life-cycle pass through various states. This is a diagram of all the states and transitions between them:

Fig. 1: Replication state diagram

Blue and yellow shapes represent replication job states.

Trapezoidal shapes represent external APIs, that's how users interact with the replicator. Writing documents to `_replicator` is the preferred way of creating replications, but posting to the `_replicate` HTTP endpoint is also supported.

Six-sided shapes are internal API boundaries. They are optional for this diagram and are only shown as additional information to help clarify how the replicator works. There are two processing stages: the first is where replication documents are parsed and become replication jobs, and the second is the scheduler itself. The scheduler runs replication jobs, periodically stopping and starting some. Jobs posted via the `_replicate` endpoint bypass the first component and go straight to the scheduler.

States descriptions

Before explaining the details of each state, it is worth noticing that color and shape of each state in the diagram:

Blue vs *yellow* partitions states into "healthy" and "unhealthy", respectively. Unhealthy states indicate something has gone wrong and it might need user's attention.

Rectangle vs *oval* separates "terminal" states from "non-terminal" ones. Terminal states are those which will not transition to other states any more. Informally, jobs in a terminal state will not be retried and don't consume memory or CPU resources.

- `Initializing`: Indicates replicator has noticed the change from the replication document. Jobs should transition quickly through this state. Being stuck here for a while could mean there is an internal error.
- `Failed`: Replication document could not be processed and turned into a valid replication job for the scheduler. This state is terminal and requires user intervention to fix the problem. A typical reason for ending up in this state is a malformed document. For example, specifying an integer for a parameter which accepts a boolean. Another reason for failure could be specifying a duplicate replication. A duplicate replication is a replication with identical parameters but a different document ID.
- `Error`: Replication document update could not be turned into a replication job. Unlike the `Failed` state, this one is temporary, and replicator will keep retrying periodically. There is an exponential backoff applied in case of consecutive failures. The main reason this state exists is to handle filtered replications with custom user functions. Filter function content is needed in order to calculate the replication ID. A replication job could not be created until the function code is retrieved. Because retrieval happens over the network, temporary failures have to be handled.
- `Running`: Replication job is running normally. This means, there might be a change feed open, and if changes are noticed, they would be processed and posted to the target. Job is still considered `Running` even if its workers are currently not streaming changes from source to target and are just waiting on the change feed. Continuous replications will most likely end up in this state.

- `Pending`: Replication job is not running and is waiting its turn. This state is reached when the number of replication jobs added to the scheduler exceeds `replicator.max_jobs`. In that case scheduler will periodically stop and start subsets of jobs trying to give each one a fair chance at making progress.
- `Crashing`: Replication job has been successfully added to the replication scheduler. However an error was encountered during the last run. Error could be a network failure, a missing source database, a permissions error, etc. Repeated consecutive crashes result in an exponential backoff. This state is considered temporary (non-terminal) and replication jobs will be periodically retried.
- `Completed`: This is a terminal, successful state for non-continuous replications. Once in this state the replication is "forgotten" by the scheduler and it doesn't consume any more CPU or memory resources. Continuous replication jobs will never reach this state.

Note: Maximum backoff interval for states `Error` and `Crashing` is calculated based on the `replicator.max_history` option. See *Replicator configuration section* for additional information.

Normal vs Continuous Replications

Normal (non-continuous) replications once started will be allowed to run to completion. That behavior is to preserve their semantics of replicating a snapshot of the source database to the target. For example if new documents are added to the source after the replication are started, those updates should not show up on the target database. Stopping and restring a normal replication would violate that constraint.

Warning: When there is a mix of continuous and normal replications, once normal replication are scheduled to run, they might temporarily starve continuous replication jobs.

However, normal replications will still be stopped and rescheduled if an operator reduces the value for the maximum number of replications. This is so that if an operator decides replications are overwhelming a node that it has the ability to recover. Any stopped replications will be resubmitted to the queue to be rescheduled.

2.2.5 Compatibility Mode

Previous version of CouchDB replicator wrote state updates back to replication documents. In cases where user code programmatically read those states, there is compatibility mode enabled via a configuration setting:

```
[replicator]
update_docs = true
```

In this mode replicator will continue to write state updates to the documents.

To effectively disable the scheduling behavior, which periodically stop and starts jobs, set `max_jobs` configuration setting to a large number. For example:

```
[replicator]
max_jobs = 9999999
```

See *Replicator configuration section* for other replicator configuration options.

2.2.6 Canceling replications

To cancel a replication simply `DELETE` the document which triggered the replication. To update a replication, for example, change the number of worker or the source, simply update the document with new data. If there is extra application-specific data in the replication documents, that data is ignored by the replicator.

2.2.7 Server restart

When CouchDB is restarted, it checks its `_replicator` databases and restarts replications described by documents if they are not already in in a `completed` or `failed` state. If they are, they are ignored.

2.2.8 Clustering

In a cluster, replication jobs are balanced evenly among all the nodes nodes such that a replication job runs on only one node at a time.

Every time there is a cluster membership change, that is when nodes are added or removed, as it happens in a rolling reboot, replicator application will notice the change, rescan all the document and running replication, and re-evaluate their cluster placement in light of the new set of live nodes. This mechanism also provides replication fail-over in case a node fails. Replication jobs started from replication documents (but not those started from `_replicate` HTTP endpoint) will automatically migrate one of the live nodes.

2.2.9 Additional Replicator Databases

Imagine replicator database (`_replicator`) has these two documents which represent pull replications from servers A and B:

```
{
    "_id": "rep_from_A",
    "source": "http://user:password@aserver.com:5984/foo",
    "target": {
        "url": "http://localhost:5984/foo_a",
        "auth": {
            "basic": {
                "username": "adm",
                "password": "pass"
            }
        }
    },
    "continuous": true
}
```

```
{
    "_id": "rep_from_B",
    "source": "http://user:password@bserver.com:5984/foo",
    "target": {
        "url": "http://localhost:5984/foo_b",
        "auth": {
            "basic": {
                "username": "adm",
                "password": "pass"
            }
        }
    },
    "continuous": true
}
```

Now without stopping and restarting CouchDB, add another replicator database. For example `another/_replicator`:

```
$ curl -X PUT http://adm:pass@localhost:5984/another%2F_replicator/
{"ok":true}
```

Note: A / (%2F) character in a database name, when used in a URL, should be escaped.

Then add a replication document to the new replicator database:

```
{
    "_id": "rep_from_X",
    "source": "http://user:password@xserver.com:5984/foo",
    "target": "http://adm:pass@localhost:5984/foo_x",
    "continuous": true
}
```

From now on, there are three replications active in the system: two replications from A and B, and a new one from X.

Then remove the additional replicator database:

```
$ curl -X DELETE http://adm:pass@localhost:5984/another%2F_replicator/
{"ok":true}
```

After this operation, replication pulling from server X will be stopped and the replications in the `_replicator` database (pulling from servers A and B) will continue.

2.2.10 Fair Share Job Scheduling

When multiple `_replicator` databases are used, and the total number of jobs on any node is greater than `max_jobs`, replication jobs will be scheduled such that each of the `_replicator` databases by default get an equal chance of running their jobs.

This is accomplished by assigning a number of "shares" to each `_replicator` database and then automatically adjusting the proportion of running jobs to match each database's proportion of shares. By default, each `_replicator` database is assigned 100 shares. It is possible to alter the share assignments for each individual `_replicator` database in the *[replicator.shares]* configuration section.

The fair share behavior is perhaps easier described with a set of examples. Each example assumes the default of `max_jobs = 500`, and two replicator databases: `_replicator` and `another/_replicator`.

Example 1: If `_replicator` has 1000 jobs and `another/_replicator` has 10, the scheduler will run about 490 jobs from `_replicator` and 10 jobs from `another/_replicator`.

Example 2: If `_replicator` has 200 jobs and `another/_replicator` also has 200 jobs, all 400 jobs will get to run as the sum of all the jobs is less than the `max_jobs` limit.

Example 3: If both replicator databases have 1000 jobs each, the scheduler will run about 250 jobs from each database on average.

Example 4: If both replicator databases have 1000 jobs each, but `_replicator` was assigned 400 shares, then on average the scheduler would run about 400 jobs from `_replicator` and 100 jobs from `_another/replicator`.

The proportions described in the examples are approximate and might oscillate a bit, and also might take anywhere from tens of minutes to an hour to converge.

2.2.11 Replicating the replicator database

Imagine you have in server C a replicator database with the two following pull replication documents in it:

```
{
     "_id": "rep_from_A",
     "source": "http://user:password@aserver.com:5984/foo",
     "target": "http://adm:pass@localhost:5984/foo_a",
     "continuous": true
}
```

```
{
     "_id": "rep_from_B",
     "source": "http://user:password@bserver.com:5984/foo",
     "target": "http://adm:pass@localhost:5984/foo_b",
     "continuous": true
}
```

Now you would like to have the same pull replications going on in server D, that is, you would like to have server D pull replicating from servers A and B. You have two options:

- Explicitly add two documents to server's D replicator database
- Replicate server's C replicator database into server's D replicator database

Both alternatives accomplish exactly the same goal.

2.2.12 Delegations

Replication documents can have a custom `user_ctx` property. This property defines the user context under which a replication runs. For the old way of triggering a replication (POSTing to `/_replicate/`), this property is not needed. That's because information about the authenticated user is readily available during the replication, which is not persistent in that case. Now, with the replicator database, the problem is that information about which user is starting a particular replication is only present when the replication document is written. The information in the replication document and the replication itself are persistent, however. This implementation detail implies that in the case of a non-admin user, a `user_ctx` property containing the user's name and a subset of their roles must be defined in the replication document. This is enforced by the document update validation function present in the default design document of the replicator database. The validation function also ensures that non-admin users are unable to set the value of the user context's `name` property to anything other than their own user name. The same principle applies for roles.

For admins, the `user_ctx` property is optional, and if it's missing it defaults to a user context with name `null` and an empty list of roles, which means design documents won't be written to local targets. If writing design documents to local targets is desired, the role `_admin` must be present in the user context's list of roles.

Also, for admins the `user_ctx` property can be used to trigger a replication on behalf of another user. This is the user context that will be passed to local target database document validation functions.

Note: The `user_ctx` property only has effect for local endpoints.

Example delegated replication document:

```
{
    "_id": "my_rep",
    "source": "http://user:password@bserver.com:5984/foo",
    "target": "http://adm:pass@localhost:5984/bar",
    "continuous": true,
    "user_ctx": {
```

(continues on next page)

(continued from previous page)

```
        "name": "joe",
        "roles": ["erlanger", "researcher"]
    }
}
```

As stated before, the `user_ctx` property is optional for admins, while being mandatory for regular (non-admin) users. When the roles property of `user_ctx` is missing, it defaults to the empty list `[]`.

2.2.13 Selector Objects

Including a Selector Object in the replication document enables you to use a query expression to determine if a document should be included in the replication.

The selector specifies fields in the document, and provides an expression to evaluate with the field content or other data. If the expression resolves to `true`, the document is replicated.

The selector object must:

- Be structured as valid JSON.
- Contain a valid query expression.

The syntax for a selector is the same as the *selectorsyntax* used for *_find*.

Using a selector is significantly more efficient than using a JavaScript filter function, and is the recommended option if filtering on document attributes only.

2.2.14 Specifying Usernames and Passwords

There are multiple ways to specify usernames and passwords for replication endpoints:

- In an `{"auth": {"basic": ...}}` object:

  New in version 3.2.0.

  ```
  {
      "target": {
          "url": "http://someurl.com/mydb",
          "auth": {
              "basic": {
                  "username": "$username",
                  "password": "$password"
               }
          }
      },
      ...
  }
  ```

  This is the prefererred format as it allows including characters like `@`, `:` and others in the username and password fields.

- In the userinfo part of the endpoint URL. This allows for a more compact endpoint represention however, it prevents using characters like `@` and `:` in usernames or passwords:

  ```
  {
      "target": "http://adm:pass@localhost:5984/bar"
      ...
  }
  ```

Specifying credentials in the userinfo part of the URL is deprecated as per RFC3986. CouchDB still supports this way of specifying credentials and doesn't yet have a target release when support will be removed.

- In an `"Authorization: Basic $b64encoded_username_and_password"` header:

```
{
    "target": {
        "url": "http://someurl.com/mydb",
            "headers": {
                "Authorization": "Basic dXNlcjpwYXNz"
            }
        },
    ...
}
```

This method has the downside of the going through the extra step of base64 encoding. In addition, it could give the impression that it encrypts or hides the credentials so it could encourage invadvertent sharing and leaking credentials.

When credentials are provided in multiple forms, they are selected in the following order:

- `"auth": {"basic": {...}}` object
- URL userinfo
- `"Authorization: Basic ..."` header.

First, the `auth` object is checked, and if credentials are defined there, they are used. If they are not, then URL userinfo is checked. If credentials are found there, then those credentials are used, otherwise basic auth header is used.

2.2.15 Replicate Winning Revisions Only

Use the `winning_revs_only: true` option to replicate "winning" document revisions only. These are the revisions that would be returned by the `GET db/doc` API endpoint by default, or appear in the `_changes` feed with the default parameters.

```
POST http://couchdb:5984/_replicate HTTP/1.1
Accept: application/json
Content-Type: application/json

{
    "winning_revs_only" : true
    "source" : "http://source:5984/recipes",
    "target" : "http://target:5984/recipes",
}
```

Replication with this mode discards conflicting revisions, so it could be one way to remove conflicts through replication.

Replication IDs and checkpoint IDs, generated by `winning_revs_only: true` replications will be different than those generated by default, so it is possible to first replicate the winning revisions, then later, to "backfill" the rest of the revisions with a regular replication job.

`winning_revs_only: true` option can be combined with filters or other options like `continuous: true` or `create_target: true`.

2.3 Replication and conflict model

Let's take the following example to illustrate replication and conflict handling.

- Alice has a document containing Bob's business card;
- She synchronizes it between her desktop PC and her laptop;
- On the desktop PC, she updates Bob's E-mail address; Without syncing again, she updates Bob's mobile number on the laptop;
- Then she replicates the two to each other again.

So on the desktop the document has Bob's new E-mail address and his old mobile number, and on the laptop it has his old E-mail address and his new mobile number.

The question is, what happens to these conflicting updated documents?

2.3.1 CouchDB replication

CouchDB works with JSON documents inside databases. Replication of databases takes place over HTTP, and can be either a "pull" or a "push", but is unidirectional. So the easiest way to perform a full sync is to do a "push" followed by a "pull" (or vice versa).

So, Alice creates v1 and sync it. She updates to v2a on one side and v2b on the other, and then replicates. What happens?

The answer is simple: both versions exist on both sides!

```
  DESKTOP                          LAPTOP
+---------+
| /db/bob |                                      INITIAL
|   v1    |                                      CREATION
+---------+

+---------+                      +---------+
| /db/bob |  ----------------->  | /db/bob |     PUSH
|   v1    |                      |   v1    |
+---------+                      +---------+

+---------+                      +---------+  INDEPENDENT
| /db/bob |                      | /db/bob |     LOCAL
|   v2a   |                      |   v2b   |     EDITS
+---------+                      +---------+

+---------+                      +---------+
| /db/bob |  ----------------->  | /db/bob |     PUSH
|   v2a   |                      |   v2a   |
+---------+                      |   v2b   |
                                 +---------+

+---------+                      +---------+
| /db/bob |  <-----------------  | /db/bob |     PULL
|   v2a   |                      |   v2a   |
|   v2b   |                      |   v2b   |
+---------+                      +---------+
```

After all, this is not a file system, so there's no restriction that only one document can exist with the name /db/bob. These are just "conflicting" revisions under the same name.

Because the changes are always replicated, the data is safe. Both machines have identical copies of both documents, so failure of a hard drive on either side won't lose any of the changes.

Another thing to notice is that peers do not have to be configured or tracked. You can do regular replications to peers, or you can do one-off, ad-hoc pushes or pulls. After the replication has taken place, there is no record kept of which peer any particular document or revision came from.

So the question now is: what happens when you try to read /db/bob? By default, CouchDB picks one arbitrary revision as the "winner", using a deterministic algorithm so that the same choice will be made on all peers. The same happens with views: the deterministically-chosen winner is the only revision fed into your map function.

Let's say that the winner is v2a. On the desktop, if Alice reads the document she'll see v2a, which is what she saved there. But on the laptop, after replication, she'll also see only v2a. It could look as if the changes she made there have been lost - but of course they have not, they have just been hidden away as a conflicting revision. But eventually she'll need these changes merged into Bob's business card, otherwise they will effectively have been lost.

Any sensible business-card application will, at minimum, have to present the conflicting versions to Alice and allow her to create a new version incorporating information from them all. Ideally it would merge the updates itself.

2.3.2 Conflict avoidance

When working on a single node, CouchDB will avoid creating conflicting revisions by returning a 409 Conflict error. This is because, when you PUT a new version of a document, you must give the `_rev` of the previous version. If that `_rev` has already been superseded, the update is rejected with a 409 Conflict response.

So imagine two users on the same node are fetching Bob's business card, updating it concurrently, and writing it back:

```
USER1    ----------->  GET /db/bob
         <-----------  {"_rev":"1-aaa", ...}

USER2    ----------->  GET /db/bob
         <-----------  {"_rev":"1-aaa", ...}

USER1    ----------->  PUT /db/bob?rev=1-aaa
         <-----------  {"_rev":"2-bbb", ...}

USER2    ----------->  PUT /db/bob?rev=1-aaa
         <-----------  409 Conflict  (not saved)
```

User2's changes are rejected, so it's up to the app to fetch /db/bob again, and either:

1. apply the same changes as were applied to the earlier revision, and submit a new PUT
2. redisplay the document so the user has to edit it again
3. just overwrite it with the document being saved before (which is not advisable, as user1's changes will be silently lost)

So when working in this mode, your application still has to be able to handle these conflicts and have a suitable retry strategy, but these conflicts never end up inside the database itself.

2.3.3 Revision tree

When you update a document in CouchDB, it keeps a list of the previous revisions. In the case where conflicting updates are introduced, this history branches into a tree, where the current conflicting revisions for this document form the tips (leaf nodes) of this tree:

```
  ,--> r2a
r1 --> r2b
  `--> r2c
```

Each branch can then extend its history - for example if you read revision r2b and then PUT with ?rev=r2b then you will make a new revision along that particular branch.

```
  ,--> r2a -> r3a -> r4a
r1 --> r2b -> r3b
  `--> r2c -> r3c
```

Here, (r4a, r3b, r3c) are the set of conflicting revisions. The way you resolve a conflict is to delete the leaf nodes along the other branches. So when you combine (r4a+r3b+r3c) into a single merged document, you would replace r4a and delete r3b and r3c.

```
  ,--> r2a -> r3a -> r4a -> r5a
r1 --> r2b -> r3b -> (r4b deleted)
  `--> r2c -> r3c -> (r4c deleted)
```

Note that r4b and r4c still exist as leaf nodes in the history tree, but as deleted docs. You can retrieve them but they will be marked `"_deleted":true`.

When you compact a database, the bodies of all the non-leaf documents are discarded. However, the list of historical _revs is retained, for the benefit of later conflict resolution in case you meet any old replicas of the database at some time in future. There is "revision pruning" to stop this getting arbitrarily large.

2.3.4 Working with conflicting documents

The basic *GET /{db}/{docid}* operation will not show you any information about conflicts. You see only the deterministically-chosen winner, and get no indication as to whether other conflicting revisions exist or not:

```
{
    "_id":"test",
    "_rev":"2-b91bb807b4685080c6a651115ff558f5",
    "hello":"bar"
}
```

If you do `GET /db/test?conflicts=true`, and the document is in a conflict state, then you will get the winner plus a _conflicts member containing an array of the revs of the other, conflicting revision(s). You can then fetch them individually using subsequent `GET /db/test?rev=xxxx` operations:

```
{
    "_id":"test",
    "_rev":"2-b91bb807b4685080c6a651115ff558f5",
    "hello":"bar",
    "_conflicts":[
        "2-65db2a11b5172bf928e3bcf59f728970",
        "2-5bc3c6319edf62d4c624277fdd0ae191"
    ]
}
```

If you do `GET /db/test?open_revs=all` then you will get all the leaf nodes of the revision tree. This will give you all the current conflicts, but will also give you leaf nodes which have been deleted (i.e. parts of the conflict history which have since been resolved). You can remove these by filtering out documents with `"_deleted":true`:

```
[
    {"ok":{"_id":"test","_rev":"2-5bc3c6319edf62d4c624277fdd0ae191","hello":"foo"}},
    {"ok":{"_id":"test","_rev":"2-65db2a11b5172bf928e3bcf59f728970","hello":"baz"}},
    {"ok":{"_id":"test","_rev":"2-b91bb807b4685080c6a651115ff558f5","hello":"bar"}}
]
```

The `"ok"` tag is an artifact of `open_revs`, which also lets you list explicit revisions as a JSON array, e.g. `open_revs=[rev1,rev2,rev3]`. In this form, it would be possible to request a revision which is now missing, because the database has been compacted.

Note: The order of revisions returned by `open_revs=all` is **NOT** related to the deterministic "winning" algorithm. In the above example, the winning revision is 2-b91b… and happens to be returned last, but in other cases it can be returned in a different position.

Once you have retrieved all the conflicting revisions, your application can then choose to display them all to the user. Or it could attempt to merge them, write back the merged version, and delete the conflicting versions - that is, to resolve the conflict permanently.

As described above, you need to update one revision and delete all the conflicting revisions explicitly. This can be done using a single *POST* to `_bulk_docs`, setting `"_deleted":true` on those revisions you wish to delete.

2.3.5 Multiple document API

Finding conflicted documents with Mango

New in version 2.2.0.

CouchDB's *Mango system* allows easy querying of documents with conflicts, returning the full body of each document as well.

Here's how to use it to find all conflicts in a database:

```
$ curl -X POST http://adm:pass@127.0.0.1:5984/dbname/_find \
    -d '{"selector": {"_conflicts": { "$exists": true}}, "conflicts": true}' \
    -Hcontent-type:application/json
```

```
{"docs": [
{"_id":"doc","_rev":"1-3975759ccff3842adf690a5c10caee42","a":2,"_conflicts":["1-
↪23202479633c2b380f79507a776743d5"]}
],
"bookmark":
↪"g1AAAABheJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYozA1kgKQ6YVA5QkBFMgKSVDHWNjI0MjEzMLc2MjZONkowtDNLML
↪"}
```

The `bookmark` value can be used to navigate through additional pages of results if necessary. Mango by default only returns 25 results per request.

If you expect to run this query often, be sure to create a Mango secondary index to speed the query:

```
$ curl -X POST http://adm:pass@127.0.0.1:5984/dbname/_index \
    -d '{"index":{"fields": ["_conflicts"]}}' \
    -Hcontent-type:application/json
```

Of course, the selector can be enhanced to filter documents on additional keys in the document. Be sure to add those keys to your secondary index as well, or a full database scan will be triggered.

Finding conflicted documents using the _all_docs index

You can fetch multiple documents at once using `include_docs=true` on a view. However, a `conflicts=true` request is ignored; the "doc" part of the value never includes a `_conflicts` member. Hence you would need to do another query to determine for each document whether it is in a conflicting state:

```
$ curl 'http://adm:pass@127.0.0.1:5984/conflict_test/_all_docs?include_docs=true&
↪conflicts=true'
```

```
{
    "total_rows":1,
    "offset":0,
    "rows":[
        {
            "id":"test",
            "key":"test",
            "value":{"rev":"2-b91bb807b4685080c6a651115ff558f5"},
            "doc":{
                "_id":"test",
                "_rev":"2-b91bb807b4685080c6a651115ff558f5",
                "hello":"bar"
            }
        }
    ]
}
```

```
$ curl 'http://adm:pass@127.0.0.1:5984/conflict_test/test?conflicts=true'
```

```
{
    "_id":"test",
    "_rev":"2-b91bb807b4685080c6a651115ff558f5",
    "hello":"bar",
    "_conflicts":[
        "2-65db2a11b5172bf928e3bcf59f728970",
        "2-5bc3c6319edf62d4c624277fdd0ae191"
    ]
}
```

2.3.6 View map functions

Views only get the winning revision of a document. However they do also get a `_conflicts` member if there are any conflicting revisions. This means you can write a view whose job is specifically to locate documents with conflicts. Here is a simple map function which achieves this:

```
function(doc) {
    if (doc._conflicts) {
        emit(null, [doc._rev].concat(doc._conflicts));
    }
}
```

which gives the following output:

```
{
    "total_rows":1,
    "offset":0,
    "rows":[
```

(continues on next page)

(continued from previous page)

```
        {
            "id":"test",
            "key":null,
            "value":[
                "2-b91bb807b4685080c6a651115ff558f5",
                "2-65db2a11b5172bf928e3bcf59f728970",
                "2-5bc3c6319edf62d4c624277fdd0ae191"
            ]
        }
    ]
}
```

If you do this, you can have a separate "sweep" process which periodically scans your database, looks for documents which have conflicts, fetches the conflicting revisions, and resolves them.

Whilst this keeps the main application simple, the problem with this approach is that there will be a window between a conflict being introduced and it being resolved. From a user's viewpoint, this may appear that the document they just saved successfully may suddenly lose their changes, only to be resurrected some time later. This may or may not be acceptable.

Also, it's easy to forget to start the sweeper, or not to implement it properly, and this will introduce odd behaviour which will be hard to track down.

CouchDB's "winning" revision algorithm may mean that information drops out of a view until a conflict has been resolved. Consider Bob's business card again; suppose Alice has a view which emits mobile numbers, so that her telephony application can display the caller's name based on caller ID. If there are conflicting documents with Bob's old and new mobile numbers, and they happen to be resolved in favour of Bob's old number, then the view won't be able to recognise his new one. In this particular case, the application might have preferred to put information from both the conflicting documents into the view, but this currently isn't possible.

Suggested algorithm to fetch a document with conflict resolution:

1. Get document via `GET docid?conflicts=true` request
2. For each member in the `_conflicts` array call `GET docid?rev=xxx`. If any errors occur at this stage, restart from step 1. (There could be a race where someone else has already resolved this conflict and deleted that rev)
3. Perform application-specific merging
4. Write `_bulk_docs` with an update to the first rev and deletes of the other revs.

This could either be done on every read (in which case you could replace all calls to GET in your application with calls to a library which does the above), or as part of your sweeper code.

And here is an example of this in Ruby using the low-level RestClient:

```
require "rubygems"
require "rest_client"
require "json"

DB = "http://adm:pass@127.0.0.1:5984/db"

# Write multiple documents
def writem(docs, new_edits)
  JSON.parse(
    RestClient.post(
      "#{DB}/_bulk_docs",
      {:docs => docs, :new_edits => new_edits}.to_json,
      {content_type: :json, accept: :json}
    )
```

(continues on next page)

(continued from previous page)

```
  )
end

# Write one document, return the rev
def write1(doc, id = nil, rev = nil)
  doc["_id"] = id if id
  doc["_rev"] = rev if rev

  if rev
    writem([doc], false)
  else
    writem([doc], true).first["rev"]
  end
end

# Read a document, return *all* revs
def read1(id)
  retries = 0
  loop do
    # FIXME: escape id
    res = [JSON.parse(RestClient.get("#{DB}/#{id}?conflicts=true"))]

    if revs = res.first.delete("_conflicts")
      begin
        revs.each do |rev|
          res << JSON.parse(RestClient.get("#{DB}/#{id}?rev=#{rev}"))
        end

      rescue
        retries += 1
        raise if retries >= 5
        next
      end
    end

    return res
  end
end

# Create DB
RestClient.delete(DB) rescue nil
RestClient.put(DB, {}.to_json)

# Write a document
rev1 = write1({"hello" => "xxx"}, "test")
p(read1("test"))

# Make three conflicting versions
(1..3).each do |num|
  write1({"hello" => "foo"}, "test", rev1 + num.to_s)
  write1({"hello" => "bar"}, "test", rev1 + num.to_s)
  write1({"hello" => "baz"}, "test", rev1 + num.to_s)
end

res = read1("test")
p(res)
```

(continues on next page)

(continued from previous page)

```
# Now let's replace these three with one
res.first["hello"] = "foo+bar+baz"
res.each_with_index do |r, i|
  unless i == 0
    r.replace({"_id" => r["_id"], "_rev" => r["_rev"], "_deleted" => true})
  end
end

writem(res, true)
p(read1("test"))
```

An application written this way never has to deal with a `PUT 409`, and is automatically multi-master capable.

You can see that it's straightforward enough when you know what you're doing. It's just that CouchDB doesn't currently provide a convenient HTTP API for "fetch all conflicting revisions", nor "PUT to supersede these N revisions", so you need to wrap these yourself. At the time of writing, there are no known client-side libraries which provide support for this.

2.3.7 Merging and revision history

Actually performing the merge is an application-specific function. It depends on the structure of your data. Sometimes it will be easy: e.g. if a document contains a list which is only ever appended to, then you can perform a union of the two list versions.

Some merge strategies look at the changes made to an object, compared to its previous version. This is how Git's merge function works.

For example, to merge Bob's business card versions v2a and v2b, you could look at the differences between v1 and v2b, and then apply these changes to v2a as well.

With CouchDB, you can sometimes get hold of old revisions of a document. For example, if you fetch `/db/bob?rev=v2b&revs_info=true` you'll get a list of the previous revision ids which ended up with revision v2b. Doing the same for v2a you can find their common ancestor revision. However if the database has been compacted, the content of that document revision will have been lost. `revs_info` will still show that v1 was an ancestor, but report it as "missing":

```
BEFORE COMPACTION           AFTER COMPACTION

     ,-> v2a                     v2a
   v1
     `-> v2b                     v2b
```

So if you want to work with diffs, the recommended way is to store those diffs within the new revision itself. That is: when you replace v1 with v2a, include an extra field or attachment in v2a which says which fields were changed from v1 to v2a. This unfortunately does mean additional book-keeping for your application.

2.3.8 Comparison with other replicating data stores

The same issues arise with other replicating systems, so it can be instructive to look at these and see how they compare with CouchDB. Please feel free to add other examples.

Unison

Unison is a bi-directional file synchronisation tool. In this case, the business card would be a file, say *bob.vcf*.

When you run unison, changes propagate both ways. If a file has changed on one side but not the other, the new replaces the old. Unison maintains a local state file so that it knows whether a file has changed since the last successful replication.

In our example it has changed on both sides. Only one file called *bob.vcf* can exist within the file system. Unison solves the problem by simply ducking out: the user can choose to replace the remote version with the local version, or vice versa (both of which would lose data), but the default action is to leave both sides unchanged.

From Alice's point of view, at least this is a simple solution. Whenever she's on the desktop she'll see the version she last edited on the desktop, and whenever she's on the laptop she'll see the version she last edited there.

But because no replication has actually taken place, the data is not protected. If her laptop hard drive dies, she'll lose all her changes made on the laptop; ditto if her desktop hard drive dies.

It's up to her to copy across one of the versions manually (under a different filename), merge the two, and then finally push the merged version to the other side.

Note also that the original file (version v1) has been lost at this point. So it's not going to be known from inspection alone whether v2a or v2b has the most up-to-date E-mail address for Bob, or which version has the most up-to-date mobile number. Alice has to remember which one she entered last.

Git

Git is a well-known distributed source control system. Like Unison, Git deals with files. However, Git considers the state of a whole set of files as a single object, the "tree". Whenever you save an update, you create a "commit" which points to both the updated tree and the previous commit(s), which in turn point to the previous tree(s). You therefore have a full history of all the states of the files. This history forms a branch, and a pointer is kept to the tip of the branch, from which you can work backwards to any previous state. The "pointer" is an SHA1 hash of the tip commit.

If you are replicating with one or more peers, a separate branch is made for each of those peers. For example, you might have:

```
main                -- my local branch
remotes/foo/main    -- branch on peer 'foo'
remotes/bar/main    -- branch on peer 'bar'
```

In the regular workflow, replication is a "pull", importing changes from a remote peer into the local repository. A "pull" does two things: first "fetch" the state of the peer into the remote tracking branch for that peer; and then attempt to "merge" those changes into the local branch.

Now let's consider the business card. Alice has created a Git repo containing `bob.vcf`, and cloned it across to the other machine. The branches look like this, where `AAAAAAAA` is the SHA1 of the commit:

```
---------- desktop ----------          ---------- laptop ----------
main: AAAAAAAA                       main: AAAAAAAA
remotes/laptop/main: AAAAAAAA        remotes/desktop/main: AAAAAAAA
```

Now she makes a change on the desktop, and commits it into the desktop repo; then she makes a different change on the laptop, and commits it into the laptop repo:

```
---------- desktop ----------          ---------- laptop ----------
main: BBBBBBBB                        main: CCCCCCCC
remotes/laptop/main: AAAAAAAA         remotes/desktop/main: AAAAAAAA
```

Now on the desktop she does `git pull laptop`. First, the remote objects are copied across into the local repo and the remote tracking branch is updated:

```
---------- desktop ----------          ---------- laptop ----------
main: BBBBBBBB                        main: CCCCCCCC
remotes/laptop/main: CCCCCCCC         remotes/desktop/main: AAAAAAAA
```

Note: The repo still contains AAAAAAAA because commits BBBBBBBB and CCCCCCCC point to it.

Then Git will attempt to merge the changes in. Knowing that the parent commit to `CCCCCCCC` is `AAAAAAAA`, it takes a diff between `AAAAAAAA` and `CCCCCCCC` and tries to apply it to `BBBBBBBB`.

If this is successful, then you'll get a new version with a merge commit:

```
---------- desktop ----------          ---------- laptop ----------
main: DDDDDDDD                        main: CCCCCCCC
remotes/laptop/main: CCCCCCCC         remotes/desktop/main: AAAAAAAA
```

Then Alice has to logon to the laptop and run `git pull desktop`. A similar process occurs. The remote tracking branch is updated:

```
---------- desktop ----------          ---------- laptop ----------
main: DDDDDDDD                        main: CCCCCCCC
remotes/laptop/main: CCCCCCCC         remotes/desktop/main: DDDDDDDD
```

Then a merge takes place. This is a special case: `CCCCCCCC` is one of the parent commits of `DDDDDDDD`, so the laptop can *fast forward* update from `CCCCCCCC` to `DDDDDDDD` directly without having to do any complex merging. This leaves the final state as:

```
---------- desktop ----------          ---------- laptop ----------
main: DDDDDDDD                        main: DDDDDDDD
remotes/laptop/main: CCCCCCCC         remotes/desktop/main: DDDDDDDD
```

Now this is all and good, but you may wonder how this is relevant when thinking about CouchDB.

First, note what happens in the case when the merge algorithm fails. The changes are still propagated from the remote repo into the local one, and are available in the remote tracking branch. So, unlike Unison, you know the data is protected. It's just that the local working copy may fail to update, or may diverge from the remote version. It's up to you to create and commit the combined version yourself, but you are guaranteed to have all the history you might need to do this.

Note that while it is possible to build new merge algorithms into Git, the standard ones are focused on line-based changes to source code. They don't work well for XML or JSON if it's presented without any line breaks.

The other interesting consideration is multiple peers. In this case you have multiple remote tracking branches, some of which may match your local branch, some of which may be behind you, and some of which may be ahead of you (i.e. contain changes that you haven't yet merged):

```
main: AAAAAAAA
remotes/foo/main: BBBBBBBB
remotes/bar/main: CCCCCCCC
remotes/baz/main: AAAAAAAA
```

Note that each peer is explicitly tracked, and therefore has to be explicitly created. If a peer becomes stale or is no longer needed, it's up to you to remove it from your configuration and delete the remote tracking branch. This is

different from CouchDB, which doesn't keep any peer state in the database.

Another difference between CouchDB and Git is that it maintains all history back to time zero - Git compaction keeps diffs between all those versions in order to reduce size, but CouchDB discards them. If you are constantly updating a document, the size of a Git repo would grow forever. It is possible (with some effort) to use "history rewriting" to make Git forget commits earlier than a particular one.

What is the CouchDB replication protocol? Is it like Git?

Author
Jason Smith

Date
2011-01-29

Source
StackOverflow

Key points

If you know Git, then you know how Couch replication works. Replicating is *very* similar to pushing or pulling with distributed source managers like Git.

CouchDB replication does not have its own protocol. A replicator simply connects to two DBs as a client, then reads from one and writes to the other. Push replication is reading the local data and updating the remote DB; pull replication is vice versa.

- **Fun fact 1**: The replicator is actually an independent Erlang application, in its own process. It connects to both couches, then reads records from one and writes them to the other.
- **Fun fact 2**: CouchDB has no way of knowing who is a normal client and who is a replicator (let alone whether the replication is push or pull). It all looks like client connections. Some of them read records. Some of them write records.

Everything flows from the data model

The replication algorithm is trivial, uninteresting. A trained monkey could design it. It's simple because the cleverness is the data model, which has these useful characteristics:

1. Every record in CouchDB is completely independent of all others. That sucks if you want to do a JOIN or a transaction, but it's awesome if you want to write a replicator. Just figure out how to replicate one record, and then repeat that for each record.
2. Like Git, records have a linked-list revision history. A record's revision ID is the checksum of its own data. Subsequent revision IDs are checksums of: the new data, plus the revision ID of the previous.
3. In addition to application data (`{"name": "Jason", "awesome": true}`), every record stores the evolutionary time line of all previous revision IDs leading up to itself.
   - Exercise: Take a moment of quiet reflection. Consider any two different records, A and B. If A's revision ID appears in B's time line, then B definitely evolved from A. Now consider Git's fast-forward merges. Do you hear that? That is the sound of your mind being blown.
4. Git isn't really a linear list. It has forks, when one parent has multiple children. CouchDB has that too.
   - Exercise: Compare two different records, A and B. A's revision ID does not appear in B's time line; however, one revision ID, C, is in both A's and B's time line. Thus A didn't evolve from B. B didn't evolve from A. But rather, A and B have a common ancestor C. In Git, that is a "fork." In CouchDB, it's a "conflict."
   - In Git, if both children go on to develop their time lines independently, that's cool. Forks totally support that.
   - In CouchDB, if both children go on to develop their time lines independently, that cool too. Conflicts totally support that.

- **Fun fact 3**: CouchDB "conflicts" do not correspond to Git "conflicts." A Couch conflict is a divergent revision history, what Git calls a "fork." For this reason the CouchDB community pronounces "conflict" with a silent *n*: "co-flicked."

5. Git also has merges, when one child has multiple parents. CouchDB *sort* of has that too.

   - **In the data model, there is no merge.** The client simply marks one time line as deleted and continues to work with the only extant time line.
   - **In the application, it feels like a merge.** Typically, the client merges the *data* from each time line in an application-specific way. Then it writes the new data to the time line. In Git, this is like copying and pasting the changes from branch A into branch B, then committing to branch B and deleting branch A. The data was merged, but there was no *git merge*.
   - These behaviors are different because, in Git, the time line itself is important; but in CouchDB, the data is important and the time line is incidental—it's just there to support replication. That is one reason why CouchDB's built-in revisioning is inappropriate for storing revision data like a wiki page.

Final notes

At least one sentence in this writeup (possibly this one) is complete BS.

2.4 CouchDB Replication Protocol

Version
3

The *CouchDB Replication Protocol* is a protocol for synchronising JSON documents between 2 peers over HTTP/1.1 by using the public *CouchDB REST API* and is based on the Apache CouchDB MVCC Data model.

2.4.1 Preface

Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in **RFC 2119**.

Goals

The primary goal of this specification is to describe the *CouchDB Replication Protocol* under the hood.

The secondary goal is to provide enough detailed information about the protocol to make it easy to build tools on any language and platform that can synchronize data with CouchDB.

Definitions

JSON:
JSON (JavaScript Object Notation) is a text format for the serialization of structured data. It is described in ECMA-262 and **RFC 4627**.

URI:
A URI is defined by **RFC 3986**. It can be a URL as defined in **RFC 1738**.

ID:
An identifier (could be a UUID) as described in **RFC 4122**.

Revision:
A MVCC token value of following pattern: `N-sig` where `N` is ALWAYS a positive integer and `sig` is the Document signature (custom). Don't mix it up with the revision in version control systems!

Leaf Revision:
The last Document Revision in a series of changes. Documents may have multiple Leaf Revisions (aka Conflict Revisions) due to concurrent updates.

Document:
A document is a JSON object with an ID and Revision defined in `_id` and `_rev` fields respectively. A Document's ID MUST be unique within the Database where it is stored.

Database:
A collection of Documents with a unique URI.

Changes Feed:
A stream of Document-changing events (create, update, delete) for the specified Database.

Sequence ID:
An ID provided by the Changes Feed. It MUST be incremental, but MAY NOT always be an integer.

Source:
Database from where the Documents are replicated.

Target:
Database where the Documents are replicated to.

Replication:
The one-way directed synchronization process of Source and Target endpoints.

Checkpoint:
Intermediate Recorded Sequence ID used for Replication recovery.

Replicator:
A service or an application which initiates and runs Replication.

Filter Function:
A special function of any programming language that is used to filter Documents during Replication (see *Filter Functions*)

Filter Function Name:
An ID of a Filter Function that may be used as a symbolic reference (aka callback function) to apply the related Filter Function to Replication.

Filtered Replication:
Replication of Documents from Source to Target using a Filter Function.

Full Replication:
Replication of all Documents from Source to Target.

Push Replication:
Replication process where Source is a local endpoint and Target is remote.

Pull Replication:
Replication process where Source is a remote endpoint and Target is local.

Continuous Replication:
Replication that "never stops": after processing all events from the Changes Feed, the Replicator doesn't close the connection, but awaits new change events from the Source. The connection is kept alive by periodic heartbeats.

Replication Log:
A special Document that holds Replication history (recorded Checkpoints and a few more statistics) between Source and Target.

Replication ID:
A unique value that unambiguously identifies the Replication Log.

2.4.2 Replication Protocol Algorithm

The *CouchDB Replication Protocol* is not *magical*, but an agreement on usage of the public *CouchDB HTTP REST API* to enable Documents to be replicated from Source to Target.

The reference implementation, written in Erlang, is provided by the couch_replicator module in Apache CouchDB.

It is RECOMMENDED that one follow this algorithm specification, use the same HTTP endpoints, and run requests with the same parameters to provide a completely compatible implementation. Custom Replicator implementations MAY use different HTTP API endpoints and request parameters depending on their local specifics and they MAY implement only part of the Replication Protocol to run only Push or Pull Replication. However, while such solutions could also run the Replication process, they loose compatibility with the CouchDB Replicator.

Verify Peers

```
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
' Verify Peers:                                                          '
'                                                                        '
'                404 Not Found   +--------------------------------+      '
'       +------------------------ |     Check Source Existence     |      '
'       |                        +--------------------------------+      '
'       |                        |          HEAD /source          |      '
'       |                        +--------------------------------+      '
'       |                          |                                     '
'       |                          | 200 OK                              '
'       |                          v                                     '
'       |                        +--------------------------------+      '
'       |                        |     Check Target Existence     | ----+ '
'       |                        +--------------------------------+     | '
'       |                        |         HEAD /target           |     | '
'       |                        +--------------------------------+     | '
'       |                          |                                    | '
'       |                          | 404 Not Found                      | '
'       v                          v                                    | '
'   +-------+    No               +--------------------------------+     | '
'   | Abort | <----------------- |         Create Target?         |     | '
'   +-------+                    +--------------------------------+     | '
'       ^                          |                                    | '
'       |                          | Yes                                | '
'       |                          v                                    | '
'       |        Failure         +--------------------------------+     | '
'       +------------------------ |          Create Target         |     | '
'                                +--------------------------------+     | '
'                                |           PUT /target          |     | '
'                                +--------------------------------+     | '
'                                  |                                    | '
'                                  | 201 Created                200 OK | '
'                                  |                                    | '
+ - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - - - | - +
                                   |                                    |
+ - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - - - | - +
' Get Peers Information:           |                                    | '
'                                  +------------------------------------+ '
'                                  |                                      '
'                                  v                                      '
'                                +--------------------------------+      '
'                                |     Get Source Information     |      '
```

(continues on next page)

(continued from previous page)

```
'                                +--------------------------------+          '
'                                                                            '
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

The Replicator MUST ensure that both Source and Target exist by using *HEAD /{db}* requests.

Check Source Existence

Request:

```
HEAD /source HTTP/1.1
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 05 Oct 2013 08:50:39 GMT
Server: CouchDB (Erlang/OTP)
```

Check Target Existence

Request:

```
HEAD /target HTTP/1.1
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 05 Oct 2013 08:51:11 GMT
Server: CouchDB (Erlang/OTP)
```

Create Target?

In case of a non-existent Target, the Replicator MAY make a *PUT /{db}* request to create the Target:

Request:

```
PUT /target HTTP/1.1
Accept: application/json
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 201 Created
Content-Length: 12
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
Date: Sat, 05 Oct 2013 08:58:41 GMT
Server: CouchDB (Erlang/OTP)

{
    "ok": true
}
```

However, the Replicator's PUT request MAY NOT succeeded due to insufficient privileges (which are granted by the provided credential) and so receive a 401 Unauthorized or a 403 Forbidden error. Such errors SHOULD be expected and well handled:

```
HTTP/1.1 500 Internal Server Error
Cache-Control: must-revalidate
Content-Length: 108
Content-Type: application/json
Date: Fri, 09 May 2014 13:50:32 GMT
Server: CouchDB (Erlang OTP)

{
    "error": "unauthorized",
    "reason": "unauthorized to access or create database http://
↪localhost:5984/target"
}
```

Abort

In case of a non-existent Source or Target, Replication SHOULD be aborted with an HTTP error response:

```
HTTP/1.1 500 Internal Server Error
Cache-Control: must-revalidate
Content-Length: 56
Content-Type: application/json
Date: Sat, 05 Oct 2013 08:55:29 GMT
Server: CouchDB (Erlang OTP)

{
    "error": "db_not_found",
    "reason": "could not open source"
}
```

Get Peers Information

```
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -+
' Verify Peers:                                                        '
'                         +------------------------+                  '
'                         | Check Target Existence |                  '
'                         +------------------------+                  '
'                                     |                                 '
'                                     | 200 OK                          '
'                                     |                                 '
+ - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - -+
                                      |
+ - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - -+
' Get Peers Information:               |                                 '
```

(continues on next page)

(continued from previous page)

```
'                                    v                            '
'                        +-------------------------+              '
'                        | Get Source Information |              '
'                        +-------------------------+              '
'                        |      GET /source       |              '
'                        +-------------------------+              '
'                                    |                            '
'                                    | 200 OK                     '
'                                    v                            '
'                        +-------------------------+              '
'                        | Get Target Information |              '
'                        +-------------------------+              '
'                        |      GET /target       |              '
'                        +-------------------------+              '
'                                    |                            '
'                                    | 200 OK                     '
'                                    |                            '
+ - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - -+
                                     |
+ - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - -+
' Find Common Ancestry:               |                            '
'                                    |                            '
'                                    v                            '
'                        +-------------------------+              '
'                        | Generate Replication ID |              '
'                        +-------------------------+              '
'                                                                 '
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -+
```

The Replicator retrieves basic information both from Source and Target using *GET /{db}* requests. The GET response MUST contain JSON objects with the following mandatory fields:

- **instance_start_time** (*string*): Always `"0"`. (Returned for legacy reasons.)
- **update_seq** (*number* / *string*): The current database Sequence ID.

Any other fields are optional. The information that the Replicator needs is the `update_seq` field: this value will be used to define a *temporary* (because Database data is subject to change) upper bound for changes feed listening and statistic calculating to show proper Replication progress.

Get Source Information

Request:

```
GET /source HTTP/1.1
Accept: application/json
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 256
Content-Type: application/json
Date: Tue, 08 Oct 2013 07:53:08 GMT
Server: CouchDB (Erlang OTP)
```

(continues on next page)

(continued from previous page)

```
{
    "committed_update_seq": 61772,
    "compact_running": false,
    "db_name": "source",
    "disk_format_version": 6,
    "doc_count": 41961,
    "doc_del_count": 3807,
    "instance_start_time": "0",
    "purge_seq": 0,
    "sizes": {
      "active": 70781613961,
      "disk": 79132913799,
      "external": 72345632950
    },
    "update_seq": 61772
}
```

Get Target Information

Request:

```
GET /target/ HTTP/1.1
Accept: application/json
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 363
Content-Type: application/json
Date: Tue, 08 Oct 2013 12:37:01 GMT
Server: CouchDB (Erlang/OTP)

{
    "compact_running": false,
    "db_name": "target",
    "disk_format_version": 5,
    "doc_count": 1832,
    "doc_del_count": 1,
    "instance_start_time": "0",
    "purge_seq": 0,
    "sizes": {
      "active": 50829452,
      "disk": 77001455,
      "external": 60326450
    },
    "update_seq": "1841-g1AAAADveJzLYWBgYMlgTmGQT0lKzi9KdUhJMtbLSs1LLUst0k"
}
```

Find Common Ancestry

```
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
' Get Peers Information:                                                        '
'                                                                              '
'                              +-------------------------------------------+ '
'                              |           Get Target Information          | '
'                              +-------------------------------------------+ '
'                                |                                             '
+ - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - - - - - - - - +
                                  |
+ - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - - - - - - - - +
' Find Common Ancestry:           v                                             '
'                              +-------------------------------------------+ '
'                              |          Generate Replication ID          | '
'                              +-------------------------------------------+ '
'                                |                                             '
'                                |                                             '
'                                v                                             '
'                              +-------------------------------------------+ '
'                              |      Get Replication Log from Source      | '
'                              +-------------------------------------------+ '
'                              |     GET /source/_local/replication-id     | '
'                              +-------------------------------------------+ '
'                                |                                             '
'                                | 200 OK                                      '
'                                | 404 Not Found                               '
'                                v                                             '
'                              +-------------------------------------------+ '
'                              |      Get Replication Log from Target      | '
'                              +-------------------------------------------+ '
'                              |     GET /target/_local/replication-id     | '
'                              +-------------------------------------------+ '
'                                |                                             '
'                                | 200 OK                                      '
'                                | 404 Not Found                               '
'                                v                                             '
'                              +-------------------------------------------+ '
'                              |          Compare Replication Logs         | '
'                              +-------------------------------------------+ '
'                                |                                             '
'                                | Use latest common sequence as start point '
'                                |                                             '
+ - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - - - - - - - - +
                                  |
                                  |
+ - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - - - - - - - - +
' Locate Changed Documents:       |                                             '
'                                |                                             '
'                                v                                             '
'                              +-------------------------------------------+ '
'                              |        Listen Source Changes Feed         | '
'                              +-------------------------------------------+ '
'                                                                              '
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

Generate Replication ID

Before Replication is started, the Replicator MUST generate a Replication ID. This value is used to track Replication History, resume and continue previously interrupted Replication process.

The Replication ID generation algorithm is implementation specific. Whatever algorithm is used it MUST uniquely identify the Replication process. CouchDB's Replicator, for example, uses the following factors in generating a Replication ID:

- Persistent Peer UUID value. For CouchDB, the local *Server UUID* is used
- Source and Target URI and if Source or Target are local or remote Databases
- If Target needed to be created
- If Replication is Continuous
- Any custom headers
- *Filter function* code if used
- Changes Feed query parameters, if any

Note: See couch_replicator_ids.erl for an example of a Replication ID generation implementation.

Retrieve Replication Logs from Source and Target

Once the Replication ID has been generated, the Replicator SHOULD retrieve the Replication Log from both Source and Target using *GET /{db}/_local/{docid}*:

Request:

```
GET /source/_local/b3e44b920ee2951cb2e123b63044427a HTTP/1.1
Accept: application/json
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 1019
Content-Type: application/json
Date: Thu, 10 Oct 2013 06:18:56 GMT
ETag: "0-8"
Server: CouchDB (Erlang OTP)

{
    "_id": "_local/b3e44b920ee2951cb2e123b63044427a",
    "_rev": "0-8",
    "history": [
        {
            "doc_write_failures": 0,
            "docs_read": 2,
            "docs_written": 2,
            "end_last_seq": 5,
            "end_time": "Thu, 10 Oct 2013 05:56:38 GMT",
            "missing_checked": 2,
            "missing_found": 2,
```

(continues on next page)

(continued from previous page)

```
            "recorded_seq": 5,
            "session_id": "d5a34cbbdafa70e0db5cb57d02a6b955",
            "start_last_seq": 3,
            "start_time": "Thu, 10 Oct 2013 05:56:38 GMT"
        },
        {
            "doc_write_failures": 0,
            "docs_read": 1,
            "docs_written": 1,
            "end_last_seq": 3,
            "end_time": "Thu, 10 Oct 2013 05:56:12 GMT",
            "missing_checked": 1,
            "missing_found": 1,
            "recorded_seq": 3,
            "session_id": "11a79cdae1719c362e9857cd1ddff09d",
            "start_last_seq": 2,
            "start_time": "Thu, 10 Oct 2013 05:56:12 GMT"
        },
        {
            "doc_write_failures": 0,
            "docs_read": 2,
            "docs_written": 2,
            "end_last_seq": 2,
            "end_time": "Thu, 10 Oct 2013 05:56:04 GMT",
            "missing_checked": 2,
            "missing_found": 2,
            "recorded_seq": 2,
            "session_id": "77cdf93cde05f15fcb710f320c37c155",
            "start_last_seq": 0,
            "start_time": "Thu, 10 Oct 2013 05:56:04 GMT"
        }
    ],
    "replication_id_version": 3,
    "session_id": "d5a34cbbdafa70e0db5cb57d02a6b955",
    "source_last_seq": 5
}
```

The Replication Log SHOULD contain the following fields:

- **history** (*array* of *object*): Replication history. **Required**
  - **doc_write_failures** (*number*): Number of failed writes
  - **docs_read** (*number*): Number of read documents
  - **docs_written** (*number*): Number of written documents
  - **end_last_seq** (*number*): Last processed Update Sequence ID
  - **end_time** (*string*): Replication completion timestamp in **RFC 5322** format
  - **missing_checked** (*number*): Number of checked revisions on Source
  - **missing_found** (*number*): Number of missing revisions found on Target
  - **recorded_seq** (*number*): Recorded intermediate Checkpoint. **Required**
  - **session_id** (*string*): Unique session ID. Commonly, a random UUID value is used. **Required**
  - **start_last_seq** (*number*): Start update Sequence ID
  - **start_time** (*string*): Replication start timestamp in **RFC 5322** format

- **replication_id_version** (*number*): Replication protocol version. Defines Replication ID calculation algorithm, HTTP API calls and the others routines. **Required**
- **session_id** (*string*): Unique ID of the last session. Shortcut to the `session_id` field of the latest `history` object. **Required**
- **source_last_seq** (*number*): Last processed Checkpoint. Shortcut to the `recorded_seq` field of the latest `history` object. **Required**

This request MAY fall with a 404 Not Found response:

Request:

```
GET /source/_local/b6cef528f67aa1a8a014dd1144b10e09 HTTP/1.1
Accept: application/json
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 404 Object Not Found
Cache-Control: must-revalidate
Content-Length: 41
Content-Type: application/json
Date: Tue, 08 Oct 2013 13:31:10 GMT
Server: CouchDB (Erlang OTP)

{
    "error": "not_found",
    "reason": "missing"
}
```

That's OK. This means that there is no information about the current Replication so it must not have been run previously and as such the Replicator MUST run a Full Replication.

Compare Replication Logs

If the Replication Logs are successfully retrieved from both Source and Target then the Replicator MUST determine their common ancestry by following the next algorithm:

- Compare `session_id` values for the chronological last session - if they match both Source and Target have a common Replication history and it seems to be valid. Use `source_last_seq` value for the startup Checkpoint
- In case of mismatch, iterate over the `history` collection to search for the latest (chronologically) common `session_id` for Source and Target. Use value of `recorded_seq` field as startup Checkpoint

If Source and Target has no common ancestry, the Replicator MUST run Full Replication.

Locate Changed Documents

```
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
' Find Common Ancestry:                                                       '
'                                                                             '
'             +-----------------------------+                                 '
'             |   Compare Replication Logs  |                                 '
'             +-----------------------------+                                 '
'                                           |                                 '
'                                           |                                 '
```

(continues on next page)

(continued from previous page)

```
+ - - - - - - - - - - - - - - - - - - - -  |  - - - - - - - - - - - - - - - +
                                          |
+ - - - - - - - - - - - - - - - - - - - -  |  - - - - - - - - - - - - - - - +
' Locate Changed Documents:                |                               '
'                                         |                               '
'                                         |                               '
'                                         v                               '
'            +-------------------------------+                            '
'   +------> |     Listen to Changes Feed    | -----+                     '
'   |        +-------------------------------+      |                     '
'   |        |     GET  /source/_changes     |      |                     '
'   |        |     POST /source/_changes     |      |                     '
'   |        +-------------------------------+      |                     '
'   |                                       |        |                     '
'   |                                       |        |                     '
'   |                There are new changes  |        | No more changes     '
'   |                                       |        |                     '
'   |                                       v        v                     '
'   |        +-------------------------------+    +----------------------+ '
'   |        |     Read Batch of Changes     |    | Replication Completed | '
'   |        +-------------------------------+    +----------------------+ '
'   |                                       |                               '
'   | No                                    |                               '
'   |                                       v                               '
'   |        +-------------------------------+                               '
'   |        |  Compare Documents Revisions  |                               '
'   |        +-------------------------------+                               '
'   |        |    POST /target/_revs_diff    |                               '
'   |        +-------------------------------+                               '
'   |                                       |                               '
'   |                                 200 OK |                               '
'   |                                       v                               '
'   |        +-------------------------------+                               '
'   +------- |     Any Differences Found?    |                               '
'            +-------------------------------+                               '
'                                         |                               '
'                                     Yes |                               '
'                                         |                               '
+ - - - - - - - - - - - - - - - - - - - -  |  - - - - - - - - - - - - - - - +
                                          |
+ - - - - - - - - - - - - - - - - - - - -  |  - - - - - - - - - - - - - - - +
' Replicate Changes:                       |                               '
'                                         v                               '
'            +-------------------------------+                               '
'            |  Fetch Next Changed Document  |                               '
'            +-------------------------------+                               '
'                                                                            '
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

Listen to Changes Feed

When the start up Checkpoint has been defined, the Replicator SHOULD read the Source's *Changes Feed* by using a `GET /{db}/_changes` request. This request MUST be made with the following query parameters:

- `feed` parameter defines the Changes Feed response style: for Continuous Replication the `continuous` value SHOULD be used, otherwise - `normal`.
- `style=all_docs` query parameter tells the Source that it MUST include all Revision leaves for each document's event in output.
- For Continuous Replication the `heartbeat` parameter defines the heartbeat period in *milliseconds*. The RECOMMENDED value by default is `10000` (10 seconds).
- If a startup Checkpoint was found during the Replication Logs comparison, the `since` query parameter MUST be passed with this value. In case of Full Replication it MAY be `0` (number zero) or be omitted.

Additionally, the `filter` query parameter MAY be specified to enable a *filter function* on Source side. Other custom parameters MAY also be provided.

Read Batch of Changes

Reading the whole feed in a single shot may not be an optimal use of resources. It is RECOMMENDED to process the feed in small chunks. However, there is no specific recommendation on chunk size since it is heavily dependent on available resources: large chunks requires more memory while they reduce I/O operations and vice versa.

Note, that Changes Feed output format is different for a request with *feed=normal* and with *feed=continuous* query parameter.

Normal Feed:

Request:

```
GET /source/_changes?feed=normal&style=all_docs&heartbeat=10000 HTTP/1.1
Accept: application/json
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Fri, 09 May 2014 16:20:41 GMT
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked

{"results":[
{"seq":14,"id":"f957f41e","changes":[{"rev":"3-46a3"}],"deleted":true}
{"seq":29,"id":"ddf339dd","changes":[{"rev":"10-304b"}]}
{"seq":37,"id":"d3cc62f5","changes":[{"rev":"2-eec2"}],"deleted":true}
{"seq":39,"id":"f13bd08b","changes":[{"rev":"1-b35d"}]}
{"seq":41,"id":"e0a99867","changes":[{"rev":"2-c1c6"}]}
{"seq":42,"id":"a75bdfc5","changes":[{"rev":"1-967a"}]}
{"seq":43,"id":"a5f467a0","changes":[{"rev":"1-5575"}]}
{"seq":45,"id":"470c3004","changes":[{"rev":"11-c292"}]}
{"seq":46,"id":"b1cb8508","changes":[{"rev":"10-ABC"}]}
{"seq":47,"id":"49ec0489","changes":[{"rev":"157-b01f"},{"rev":"123-6f7c"}]}
{"seq":49,"id":"dad10379","changes":[{"rev":"1-9346"},{"rev":"6-5b8a"}]}
{"seq":50,"id":"73464877","changes":[{"rev":"1-9f08"}]}
```

(continues on next page)

(continued from previous page)

```
{"seq":51,"id":"7ae19302","changes":[{"rev":"1-57bf"}]}
{"seq":63,"id":"6a7a6c86","changes":[{"rev":"5-acf6"}],"deleted":true}
{"seq":64,"id":"dfb9850a","changes":[{"rev":"1-102f"}]}
{"seq":65,"id":"c532afa7","changes":[{"rev":"1-6491"}]}
{"seq":66,"id":"af8a9508","changes":[{"rev":"1-3db2"}]}
{"seq":67,"id":"caa3dded","changes":[{"rev":"1-6491"}]}
{"seq":68,"id":"79f3b4e9","changes":[{"rev":"1-102f"}]}
{"seq":69,"id":"1d89d16f","changes":[{"rev":"1-3db2"}]}
{"seq":71,"id":"abae7348","changes":[{"rev":"2-7051"}]}
{"seq":77,"id":"6c25534f","changes":[{"rev":"9-CDE"},{"rev":"3-00e7"},{"rev
↪":"1-ABC"}]}
{"seq":78,"id":"SpaghettiWithMeatballs","changes":[{"rev":"22-5f95"}]}
],
"last_seq":78}
```

Continuous Feed:

Request:

```
GET /source/_changes?feed=continuous&style=all_docs&heartbeat=10000 HTTP/1.1
Accept: application/json
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Fri, 09 May 2014 16:22:22 GMT
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked

{"seq":14,"id":"f957f41e","changes":[{"rev":"3-46a3"}],"deleted":true}
{"seq":29,"id":"ddf339dd","changes":[{"rev":"10-304b"}]}
{"seq":37,"id":"d3cc62f5","changes":[{"rev":"2-eec2"}],"deleted":true}
{"seq":39,"id":"f13bd08b","changes":[{"rev":"1-b35d"}]}
{"seq":41,"id":"e0a99867","changes":[{"rev":"2-c1c6"}]}
{"seq":42,"id":"a75bdfc5","changes":[{"rev":"1-967a"}]}
{"seq":43,"id":"a5f467a0","changes":[{"rev":"1-5575"}]}
{"seq":45,"id":"470c3004","changes":[{"rev":"11-c292"}]}
{"seq":46,"id":"b1cb8508","changes":[{"rev":"10-ABC"}]}
{"seq":47,"id":"49ec0489","changes":[{"rev":"157-b01f"},{"rev":"123-6f7c"}]}
{"seq":49,"id":"dad10379","changes":[{"rev":"1-9346"},{"rev":"6-5b8a"}]}
{"seq":50,"id":"73464877","changes":[{"rev":"1-9f08"}]}
{"seq":51,"id":"7ae19302","changes":[{"rev":"1-57bf"}]}
{"seq":63,"id":"6a7a6c86","changes":[{"rev":"5-acf6"}],"deleted":true}
{"seq":64,"id":"dfb9850a","changes":[{"rev":"1-102f"}]}
{"seq":65,"id":"c532afa7","changes":[{"rev":"1-6491"}]}
{"seq":66,"id":"af8a9508","changes":[{"rev":"1-3db2"}]}
{"seq":67,"id":"caa3dded","changes":[{"rev":"1-6491"}]}
{"seq":68,"id":"79f3b4e9","changes":[{"rev":"1-102f"}]}
{"seq":69,"id":"1d89d16f","changes":[{"rev":"1-3db2"}]}
{"seq":71,"id":"abae7348","changes":[{"rev":"2-7051"}]}
{"seq":75,"id":"SpaghettiWithMeatballs","changes":[{"rev":"21-5949"}]}
{"seq":77,"id":"6c255","changes":[{"rev":"9-CDE"},{"rev":"3-00e7"},{"rev":
↪"1-ABC"}]}
```

(continues on next page)

(continued from previous page)

```
{"seq":78,"id":"SpaghettiWithMeatballs","changes":[{"rev":"22-5f95"}]}
```

For both Changes Feed formats record-per-line style is preserved to simplify iterative fetching and decoding JSON objects with less memory footprint.

Calculate Revision Difference

After reading the batch of changes from the Changes Feed, the Replicator forms a JSON mapping object for Document ID and related leaf Revisions and sends the result to Target via a *POST /{db}/_revs_diff* request:

Request:

```
POST /target/_revs_diff HTTP/1.1
Accept: application/json
Content-Length: 287
Content-Type: application/json
Host: localhost:5984
User-Agent: CouchDB

{
    "baz": [
        "2-7051cbe5c8faecd085a3fa619e6e6337"
    ],
    "foo": [
        "3-6a540f3d701ac518d3b9733d673c5484"
    ],
    "bar": [
        "1-d4e501ab47de6b2000fc8a02f84a0c77",
        "1-967a00dff5e02add41819138abb3284d"
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 88
Content-Type: application/json
Date: Fri, 25 Oct 2013 14:44:41 GMT
Server: CouchDB (Erlang/OTP)

{
    "baz": {
        "missing": [
            "2-7051cbe5c8faecd085a3fa619e6e6337"
        ]
    },
    "bar": {
        "missing": [
            "1-d4e501ab47de6b2000fc8a02f84a0c77"
        ]
    }
}
```

In the response the Replicator receives a Document ID – Revisions mapping, but only for Revisions that do not exist in Target and are REQUIRED to be transferred from Source.

If all Revisions in the request match the current state of the Documents then the response will contain an empty JSON object:

Request

```
POST /target/_revs_diff HTTP/1.1
Accept: application/json
Content-Length: 160
Content-Type: application/json
Host: localhost:5984
User-Agent: CouchDB

{
    "foo": [
        "3-6a540f3d701ac518d3b9733d673c5484"
    ],
    "bar": [
        "1-967a00dff5e02add41819138abb3284d"
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 2
Content-Type: application/json
Date: Fri, 25 Oct 2013 14:45:00 GMT
Server: CouchDB (Erlang/OTP)

{}
```

Replication Completed

When there are no more changes left to process and no more Documents left to replicate, the Replicator finishes the Replication process. If Replication wasn't Continuous, the Replicator MAY return a response to client with statistics about the process.

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 414
Content-Type: application/json
Date: Fri, 09 May 2014 15:14:19 GMT
Server: CouchDB (Erlang OTP)

{
    "history": [
        {
            "doc_write_failures": 2,
            "docs_read": 2,
            "docs_written": 0,
            "end_last_seq": 2939,
            "end_time": "Fri, 09 May 2014 15:14:19 GMT",
            "missing_checked": 1835,
            "missing_found": 2,
            "recorded_seq": 2939,
            "session_id": "05918159f64842f1fe73e9e2157b2112",
```

(continues on next page)

(continued from previous page)

```
            "start_last_seq": 0,
            "start_time": "Fri, 09 May 2014 15:14:18 GMT"
        }
    ],
    "ok": true,
    "replication_id_version": 3,
    "session_id": "05918159f64842f1fe73e9e2157b2112",
    "source_last_seq": 2939
}
```

Replicate Changes

```
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
' Locate Changed Documents:                                                    '
'                                                                             '
'               +-------------------------------------+                       '
'               |      Any Differences Found?         |                       '
'               +-------------------------------------+                       '
'                                                     |                       '
'                                                     |                       '
'                                                     |                       '
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - +
                                                      |
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - +
' Replicate Changes:                                  |                        '
'                                                     v                        '
'               +-------------------------------------+                       '
'   +---------> |     Fetch Next Changed Document     | <---------------------+   '
'   |           +-------------------------------------+                       |   '
'   |           |          GET /source/docid          |                       |   '
'   |           +-------------------------------------+                       |   '
'   |             |                                                           |   '
'   |             |                                                           |   '
'   |             |                                           201 Created     |   '
'   |             | 200 OK                                    401 Unauthorized |   '
'   |             |                                           403 Forbidden   |   '
'   |             |                                                           |   '
'   |             v                                                           |   '
'   |           +-------------------------------------+                       |   '
'   |   +------ |  Document Has Changed Attachments?  |                       |   '
'   |   |       +-------------------------------------+                       |   '
'   |   |         |                                                           |   '
'   |   |         |                                                           |   '
'   |   |         | Yes                                                       |   '
'   |   |         |                                                           |   '
'   |   |         v                                                           |   '
'   |   |       +------------------------+   Yes    +---------------------------+ '
'   |   | No    |  Are They Big Enough?  | -------> | Update Document on Target | '
'   |   |       +------------------------+          +---------------------------+ '
'   |   |         |                                 |     PUT /target/docid     | '
'   |   |         |                                 +---------------------------+ '
'   |   |         |                                                               '
'   |   |         | No                                                            '
'   |   |         |                                                               '
'   |   |         v                                                               '
```

(continues on next page)

(continued from previous page)

```
'   |   |       +-------------------------------------+                           '
'   |   +-----> |     Put Document Into the Stack     |                           '
'   |           +-------------------------------------+                           '
'   |             |                                                               '
'   |             |                                                               '
'   |             v                                                               '
'   |     No    +-------------------------------------+                           '
'   +---------- |           Stack is Full?            |                           '
'   |           +-------------------------------------+                           '
'   |             |                                                               '
'   |             | Yes                                                           '
'   |             |                                                               '
'   |             v                                                               '
'   |           +-------------------------------------+                           '
'   |           | Upload Stack of Documents to Target |                           '
'   |           +-------------------------------------+                           '
'   |           |       POST /target/_bulk_docs       |                           '
'   |           +-------------------------------------+                           '
'   |             |                                                               '
'   |             | 201 Created                                                   '
'   |             v                                                               '
'   |           +-------------------------------------+                           '
'   |           |          Ensure in Commit           |                           '
'   |           +-------------------------------------+                           '
'   |           |  POST /target/_ensure_full_commit   |                           '
'   |           +-------------------------------------+                           '
'   |             |                                                               '
'   |             | 201 Created                                                   '
'   |             v                                                               '
'   |           +-------------------------------------+                           '
'   |           |    Record Replication Checkpoint    |                           '
'   |           +-------------------------------------+                           '
'   |           |  PUT /source/_local/replication-id  |                           '
'   |           |  PUT /target/_local/replication-id  |                           '
'   |           +-------------------------------------+                           '
'   |             |                                                               '
'   |             | 201 Created                                                   '
'   |             v                                                               '
'   |     No    +-------------------------------------+                           '
'   +---------- | All Documents from Batch Processed? |                           '
'               +-------------------------------------+                           '
'                                                   |                             '
'                                               Yes |                             '
'                                                   |                             '
+ - - - - - - - - - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - +
                                                    |
+ - - - - - - - - - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - +
' Locate Changed Documents:                          |                             '
'                                                   v                             '
'               +-------------------------------------+                           '
'               |       Listen to Changes Feed        |                           '
'               +-------------------------------------+                           '
'                                                                                 '
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

Fetch Changed Documents

At this step the Replicator MUST fetch all Document Leaf Revisions from Source that are missed at Target. This operation is effective if Replication WILL use previously calculated Revision differences since they define missing Documents and their Revisions.

To fetch the Document the Replicator will make a *GET /{db}/{docid}* request with the following query parameters:

- `revs=true`: Instructs the Source to include the list of all known revisions into the Document in the `_revisions` field. This information is needed to synchronize the Document's ancestors history between Source and Target
- The `open_revs` query parameter contains a JSON array with a list of Leaf Revisions that are needed to be fetched. If the specified Revision exists then the Document MUST be returned for this Revision. Otherwise, Source MUST return an object with the single field `missing` with the missed Revision as the value. In case the Document contains attachments, Source MUST return information only for those ones that had been changed (added or updated) since the specified Revision values. If an attachment was deleted, the Document MUST NOT have stub information for it
- `latest=true`: Ensures, that Source will return the latest Document Revision regardless of which one was specified in the `open_revs` query parameter. This parameter solves a race condition problem where the requested Document may be changed in between this step and handling related events on the Changes Feed

In the response Source SHOULD return *multipart/mixed* or respond instead with *application/json* unless the Accept header specifies a different mime type. The *multipart/mixed* content type allows handling the response data as a stream, since there could be multiple documents (one per each Leaf Revision) plus several attachments. These attachments are mostly binary and JSON has no way to handle such data except as base64 encoded strings which are very ineffective for transfer and processing operations.

With a *multipart/mixed* response the Replicator handles multiple Document Leaf Revisions and their attachments one by one as raw data without any additional encoding applied. There is also one agreement to make data processing more effective: the Document ALWAYS goes before its attachments, so the Replicator has no need to process all the data to map related Documents-Attachments and may handle it as stream with lesser memory footprint.

Request:

```
GET /source/SpaghettiWithMeatballs?revs=true&open_revs=[%225-00ecbbc%22,
↪%221-917fa23%22,%223-6bcedf1%22]&latest=true HTTP/1.1
Accept: multipart/mixed
Host: localhost:5984
User-Agent: CouchDB
```

Response:

```
HTTP/1.1 200 OK
Content-Type: multipart/mixed; boundary="7b1596fc4940bc1be725ad67f11ec1c4"
Date: Thu, 07 Nov 2013 15:10:16 GMT
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked

--7b1596fc4940bc1be725ad67f11ec1c4
Content-Type: application/json

{
    "_id": "SpaghettiWithMeatballs",
    "_rev": "1-917fa23",
    "_revisions": {
        "ids": [
            "917fa23"
```

(continues on next page)

(continued from previous page)

```
        ],
        "start": 1
    },
    "description": "An Italian-American delicious dish",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
--7b1596fc4940bc1be725ad67f11ec1c4
Content-Type: multipart/related; boundary="a81a77b0ca68389dda3243a43ca946f2"

--a81a77b0ca68389dda3243a43ca946f2
Content-Type: application/json

{
    "_attachments": {
      "recipe.txt": {
          "content_type": "text/plain",
          "digest": "md5-R5CrCb6fX10Y46AqtNn0oQ==",
          "follows": true,
          "length": 87,
          "revpos": 7
      }
    },
    "_id": "SpaghettiWithMeatballs",
    "_rev": "7-474f12e",
    "_revisions": {
        "ids": [
            "474f12e",
            "5949cfc",
            "00ecbbc",
            "fc997b6",
            "3552c87",
            "404838b",
            "5defd9d",
            "dc1e4be"
        ],
        "start": 7
    },
    "description": "An Italian-American delicious dish",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs",
        "love"
    ],
    "name": "Spaghetti with meatballs"
}
--a81a77b0ca68389dda3243a43ca946f2
Content-Disposition: attachment; filename="recipe.txt"
Content-Type: text/plain
Content-Length: 87
```

(continues on next page)

(continued from previous page)

```
1. Cook spaghetti
2. Cook meetballs
3. Mix them
4. Add tomato sauce
5. ...
6. PROFIT!

--a81a77b0ca68389dda3243a43ca946f2--
--7b1596fc4940bc1be725ad67f11ec1c4
Content-Type: application/json; error="true"

{"missing":"3-6bcedf1"}
--7b1596fc4940bc1be725ad67f11ec1c4--
```

After receiving the response, the Replicator puts all the received data into a local stack for further bulk upload to utilize network bandwidth effectively. The local stack size could be limited by number of Documents or bytes of handled JSON data. When the stack is full the Replicator uploads all the handled Document in bulk mode to the Target. While bulk operations are highly RECOMMENDED to be used, in certain cases the Replicator MAY upload Documents to Target one by one.

Note: Alternative Replicator implementations MAY use alternative ways to retrieve Documents from Source. For instance, PouchDB doesn't use the Multipart API and fetches only the latest Document Revision with inline attachments as a single JSON object. While this is still valid CouchDB HTTP API usage, such solutions MAY require a different API implementation for non-CouchDB Peers.

Upload Batch of Changed Documents

To upload multiple Documents in a single shot the Replicator sends a *POST /{db}/_bulk_docs* request to Target with payload containing a JSON object with the following mandatory fields:

- **docs** (*array* of *objects*): List of Document objects to update on Target. These Documents MUST contain the `_revisions` field that holds a list of the full Revision history to let Target create Leaf Revisions that correctly preserve ancestry
- **new_edits** (*boolean*): Special flag that instructs Target to store Documents with the specified Revision (field `_rev`) value as-is without generating a new revision. Always `false`

The request also MAY contain *X-Couch-Full-Commit* that used to control CouchDB <3.0 behavior when delayed commits were enabled. Other Peers MAY ignore this header or use it to control similar local feature.

Request:

```
POST /target/_bulk_docs HTTP/1.1
Accept: application/json
Content-Length: 826
Content-Type:application/json
Host: localhost:5984
User-Agent: CouchDB
X-Couch-Full-Commit: false

{
    "docs": [
        {
            "_id": "SpaghettiWithMeatballs",
            "_rev": "1-917fa2381192822767f010b95b45325b",
            "_revisions": {
```

(continues on next page)

(continued from previous page)

```
                "ids": [
                    "917fa2381192822767f010b95b45325b"
                ],
                "start": 1
            },
            "description": "An Italian-American delicious dish",
            "ingredients": [
                "spaghetti",
                "tomato sauce",
                "meatballs"
            ],
            "name": "Spaghetti with meatballs"
        },
        {
            "_id": "LambStew",
            "_rev": "1-34c318924a8f327223eed702ddfdc66d",
            "_revisions": {
                "ids": [
                    "34c318924a8f327223eed702ddfdc66d"
                ],
                "start": 1
            },
            "servings": 6,
            "subtitle": "Delicious with scone topping",
            "title": "Lamb Stew"
        },
        {
            "_id": "FishStew",
            "_rev": "1-9c65296036141e575d32ba9c034dd3ee",
            "_revisions": {
                "ids": [
                    "9c65296036141e575d32ba9c034dd3ee"
                ],
                "start": 1
            },
            "servings": 4,
            "subtitle": "Delicious with fresh bread",
            "title": "Fish Stew"
        }
    ],
    "new_edits": false
}
```

In its response Target MUST return a JSON array with a list of Document update statuses. If the Document has been stored successfully, the list item MUST contain the field `ok` with `true` value. Otherwise it MUST contain `error` and `reason` fields with error type and a human-friendly reason description.

Document updating failure isn't fatal as Target MAY reject the update for its own reasons. It's RECOMMENDED to use error type `forbidden` for rejections, but other error types can also be used (like invalid field name etc.). The Replicator SHOULD NOT retry uploading rejected documents unless there are good reasons for doing so (e.g. there is special error type for that).

Note that while a update may fail for one Document in the response, Target can still return a 201 Created response. Same will be true if all updates fail for all uploaded Documents.

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 246
Content-Type: application/json
Date: Sun, 10 Nov 2013 19:02:26 GMT
Server: CouchDB (Erlang/OTP)

[
    {
        "ok": true,
        "id": "SpaghettiWithMeatballs",
        "rev":" 1-917fa2381192822767f010b95b45325b"
    },
    {
        "ok": true,
        "id": "FishStew",
        "rev": "1-9c65296036141e575d32ba9c034dd3ee"
    },
    {
        "error": "forbidden",
        "id": "LambStew",
        "reason": "sorry",
        "rev": "1-34c318924a8f327223eed702ddfdc66d"
    }
]
```

Upload Document with Attachments

There is a special optimization case when then Replicator WILL NOT use bulk upload of changed Documents. This case is applied when Documents contain a lot of attached files or the files are too big to be efficiently encoded with Base64.

For this case the Replicator issues a `/{db}/{docid}?new_edits=false` request with `multipart/related` content type. Such a request allows one to easily stream the Document and all its attachments one by one without any serialization overhead.

Request:

```
PUT /target/SpaghettiWithMeatballs?new_edits=false HTTP/1.1
Accept: application/json
Content-Length: 1030
Content-Type: multipart/related; boundary="864d690aeb91f25d469dec6851fb57f2"
Host: localhost:5984
User-Agent: CouchDB

--2fa48cba80d0cdba7829931fe8acce9d
Content-Type: application/json

{
    "_attachments": {
        "recipe.txt": {
            "content_type": "text/plain",
            "digest": "md5-R5CrCb6fX10Y46AqtNn0oQ==",
            "follows": true,
            "length": 87,
            "revpos": 7
```

(continues on next page)

(continued from previous page)

```
        }
    },
    "_id": "SpaghettiWithMeatballs",
    "_rev": "7-474f12eb068c717243487a9505f6123b",
    "_revisions": {
        "ids": [
            "474f12eb068c717243487a9505f6123b",
            "5949cfcd437e3ee22d2d98a26d1a83bf",
            "00ecbbc54e2a171156ec345b77dfdf59",
            "fc997b62794a6268f2636a4a176efcd6",
            "3552c87351aadc1e4bea2461a1e8113a",
            "404838bc2862ce76c6ebed046f9eb542",
            "5defd9d813628cea6e98196eb0ee8594"
        ],
        "start": 7
    },
    "description": "An Italian-American delicious dish",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs",
        "love"
    ],
    "name": "Spaghetti with meatballs"
}
--2fa48cba80d0cdba7829931fe8acce9d
Content-Disposition: attachment; filename="recipe.txt"
Content-Type: text/plain
Content-Length: 87

1. Cook spaghetti
2. Cook meetballs
3. Mix them
4. Add tomato sauce
5. ...
6. PROFIT!

--2fa48cba80d0cdba7829931fe8acce9d--
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 105
Content-Type: application/json
Date: Fri, 08 Nov 2013 16:35:27 GMT
Server: CouchDB (Erlang/OTP)

{
    "ok": true,
    "id": "SpaghettiWithMeatballs",
    "rev": "7-474f12eb068c717243487a9505f6123b"
}
```

Unlike bulk updating via *POST /{db}/_bulk_docs* endpoint, the response MAY come with a different status code. For instance, in the case when the Document is rejected, Target SHOULD respond with a 403 Forbidden:

Response:

```
HTTP/1.1 403 Forbidden
Cache-Control: must-revalidate
Content-Length: 39
Content-Type: application/json
Date: Fri, 08 Nov 2013 16:35:27 GMT
Server: CouchDB (Erlang/OTP)

{
    "error": "forbidden",
    "reason": "sorry"
}
```

Replicator SHOULD NOT retry requests in case of a 401 Unauthorized, 403 Forbidden, 409 Conflict or 412 Precondition Failed since repeating the request couldn't solve the issue with user credentials or uploaded data.

Ensure In Commit

Once a batch of changes has been successfully uploaded to Target, the Replicator issues a *POST /{db}/_ensure_full_commit* request to ensure that every transferred bit is laid down on disk or other *persistent* storage place. Target MUST return 201 Created response with a JSON object containing the following mandatory fields:

- **instance_start_time** (*string*): Timestamp of when the database was opened, expressed in *microseconds* since the epoch
- **ok** (*boolean*): Operation status. Constantly `true`

Request:

```
POST /target/_ensure_full_commit HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 53
Content-Type: application/json
Date: Web, 06 Nov 2013 18:20:43 GMT
Server: CouchDB (Erlang/OTP)

{
    "instance_start_time": "0",
    "ok": true
}
```

Record Replication Checkpoint

Since batches of changes were uploaded and committed successfully, the Replicator updates the Replication Log both on Source and Target recording the current Replication state. This operation is REQUIRED so that in the case of Replication failure the replication can resume from last point of success, not from the very beginning.

Replicator updates Replication Log on Source:

Request:

```
PUT /source/_local/afa899a9e59589c3d4ce5668e3218aef HTTP/1.1
Accept: application/json
Content-Length: 591
Content-Type: application/json
Host: localhost:5984
User-Agent: CouchDB

{
    "_id": "_local/afa899a9e59589c3d4ce5668e3218aef",
    "_rev": "0-1",
    "_revisions": {
        "ids": [
            "31f36e40158e717fbe9842e227b389df"
        ],
        "start": 1
    },
    "history": [
        {
            "doc_write_failures": 0,
            "docs_read": 6,
            "docs_written": 6,
            "end_last_seq": 26,
            "end_time": "Thu, 07 Nov 2013 09:42:17 GMT",
            "missing_checked": 6,
            "missing_found": 6,
            "recorded_seq": 26,
            "session_id": "04bf15bf1d9fa8ac1abc67d0c3e04f07",
            "start_last_seq": 0,
            "start_time": "Thu, 07 Nov 2013 09:41:43 GMT"
        }
    ],
    "replication_id_version": 3,
    "session_id": "04bf15bf1d9fa8ac1abc67d0c3e04f07",
    "source_last_seq": 26
}
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 75
Content-Type: application/json
Date: Thu, 07 Nov 2013 09:42:17 GMT
Server: CouchDB (Erlang/OTP)

{
    "id": "_local/afa899a9e59589c3d4ce5668e3218aef",
    "ok": true,
```

(continues on next page)

(continued from previous page)

```
    "rev": "0-2"
}
```

...and on Target too:

Request:

```
PUT /target/_local/afa899a9e59589c3d4ce5668e3218aef HTTP/1.1
Accept: application/json
Content-Length: 591
Content-Type: application/json
Host: localhost:5984
User-Agent: CouchDB

{
    "_id": "_local/afa899a9e59589c3d4ce5668e3218aef",
    "_rev": "1-31f36e40158e717fbe9842e227b389df",
    "_revisions": {
        "ids": [
            "31f36e40158e717fbe9842e227b389df"
        ],
        "start": 1
    },
    "history": [
        {
            "doc_write_failures": 0,
            "docs_read": 6,
            "docs_written": 6,
            "end_last_seq": 26,
            "end_time": "Thu, 07 Nov 2013 09:42:17 GMT",
            "missing_checked": 6,
            "missing_found": 6,
            "recorded_seq": 26,
            "session_id": "04bf15bf1d9fa8ac1abc67d0c3e04f07",
            "start_last_seq": 0,
            "start_time": "Thu, 07 Nov 2013 09:41:43 GMT"
        }
    ],
    "replication_id_version": 3,
    "session_id": "04bf15bf1d9fa8ac1abc67d0c3e04f07",
    "source_last_seq": 26
}
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 106
Content-Type: application/json
Date: Thu, 07 Nov 2013 09:42:17 GMT
Server: CouchDB (Erlang/OTP)

{
    "id": "_local/afa899a9e59589c3d4ce5668e3218aef",
    "ok": true,
    "rev": "2-9b5d1e36bed6ae08611466e30af1259a"
}
```

Continue Reading Changes

Once a batch of changes had been processed and transferred to Target successfully, the Replicator can continue to listen to the Changes Feed for new changes. If there are no new changes to process the Replication is considered to be done.

For Continuous Replication, the Replicator MUST continue to wait for new changes from Source.

2.4.3 Protocol Robustness

Since the *CouchDB Replication Protocol* works on top of HTTP, which is based on TCP/IP, the Replicator SHOULD expect to be working within an unstable environment with delays, losses and other bad surprises that might eventually occur. The Replicator SHOULD NOT count every HTTP request failure as a *fatal error*. It SHOULD be smart enough to detect timeouts, repeat failed requests, be ready to process incomplete or malformed data and so on. *Data must flow* - that's the rule.

2.4.4 Error Responses

In case something goes wrong the Peer MUST respond with a JSON object with the following REQUIRED fields:

- **error** (*string*): Error type for programs and developers
- **reason** (*string*): Error description for humans

Bad Request

If a request contains malformed data (like invalid JSON) the Peer MUST respond with a HTTP 400 Bad Request and `bad_request` as error type:

```
{
    "error": "bad_request",
    "reason": "invalid json"
}
```

Unauthorized

If a Peer REQUIRES credentials be included with the request and the request does not contain acceptable credentials then the Peer MUST respond with the HTTP 401 Unauthorized and `unauthorized` as error type:

```
{
    "error": "unauthorized",
    "reason": "Name or password is incorrect"
}
```

Forbidden

If a Peer receives valid user credentials, but the requester does not have sufficient permissions to perform the operation then the Peer MUST respond with a HTTP 403 Forbidden and `forbidden` as error type:

```
{
    "error": "forbidden",
    "reason": "You may only update your own user document."
}
```

Resource Not Found

If the requested resource, Database or Document wasn't found on a Peer, the Peer MUST respond with a HTTP 404 Not Found and `not_found` as error type:

```
{
    "error": "not_found",
    "reason": "database \"target\" does not exists"
}
```

Method Not Allowed

If an unsupported method was used then the Peer MUST respond with a HTTP 405 Method Not Allowed and `method_not_allowed` as error type:

```
{
    "error": "method_not_allowed",
    "reason": "Only GET, PUT, DELETE allowed"
}
```

Resource Conflict

A resource conflict error occurs when there are concurrent updates of the same resource by multiple clients. In this case the Peer MUST respond with a HTTP 409 Conflict and `conflict` as error type:

```
{
    "error": "conflict",
    "reason": "document update conflict"
}
```

Precondition Failed

The HTTP 412 Precondition Failed response may be sent in case of an attempt to create a Database (error type `db_exists`) that already exists or some attachment information is missing (error type `missing_stub`). There is no explicit error type restrictions, but it is RECOMMEND to use error types that are previously mentioned:

```
{
    "error": "db_exists",
    "reason": "database \"target\" exists"
}
```

Server Error

Raised in case an error is *fatal* and the Replicator cannot do anything to continue Replication. In this case the Replicator MUST return a HTTP 500 Internal Server Error response with an error description (no restrictions on error type applied):

```
{
    "error": "worker_died",
    "reason": "kaboom!"
}
```

2.4.5 Optimisations

There are RECOMMENDED approaches to optimize the Replication process:

- Keep the number of HTTP requests at a reasonable minimum
- Try to work with a connection pool and make parallel/multiple requests whenever possible
- Don't close sockets after each request: respect the keep-alive option
- Use continuous sessions (cookies, etc.) to reduce authentication overhead
- Try to use bulk requests for every operations with Documents
- Find out optimal batch size for Changes feed processing
- Preserve Replication Logs and resume Replication from the last Checkpoint whenever possible
- Optimize filter functions: let them run as fast as possible
- Get ready for surprises: networks are very unstable environments

2.4.6 API Reference

Common Methods

- `HEAD /{db}` – Check Database existence
- `GET /{db}` – Retrieve Database information
- `GET /{db}/_local/{docid}` – Read the last Checkpoint
- `PUT /{db}/_local/{docid}` – Save a new Checkpoint

For Target

- `PUT /{db}` – Create Target if it not exists and the option was provided
- `POST /{db}/_revs_diff` – Locate Revisions that are not known to Target
- `POST /{db}/_bulk_docs` – Upload Revisions to Target
- `PUT /{db}/{docid}` – Upload a single Document with attachments to Target
- `POST /{db}/_ensure_full_commit` – Ensure that all changes are stored on disk

For Source

- `GET /{db}/_changes` – Fetch changes since the last pull of Source
- `POST /{db}/_changes` – Fetch changes for specified Document IDs since the last pull of Source
- `GET /{db}/{docid}` – Retrieve a single Document from Source with attachments

2.4.7 Reference

- Refuge RCouch wiki
- CouchBase Lite IOS wiki

CHAPTER

THREE

DESIGN DOCUMENTS

CouchDB supports special documents within databases known as "design documents". These documents, mostly driven by JavaScript you write, are used to build indexes, validate document updates, format query results, and filter replications.

3.1 Design Documents

In this section we'll show how to write design documents, using the built-in *JavaScript Query Server*.

But before we start to write our first document, let's take a look at the list of common objects that will be used during our code journey - we'll be using them extensively within each function:

- *Database information object*
- *Request object*
- *Response object*
- *UserCtx object*
- *Database Security object*
- *Guide to JavaScript Query Server*

3.1.1 Creation and Structure

Design documents contain functions such as view and update functions. These functions are executed when requested.

Design documents are denoted by an id field with the format `_design/{name}`. Their structure follows the example below.

Example:

```
{
    "_id": "_design/example",
    "views": {
        "view-number-one": {
            "map": "function (doc) {/* function code here - see below */}"
        },
        "view-number-two": {
            "map": "function (doc) {/* function code here - see below */}",
            "reduce": "function (keys, values, rereduce) {/* function code here - see below */}"
        }
    },
    "updates": {
        "updatefun1": "function(doc,req) {/* function code here - see below */}",
```

(continues on next page)

(continued from previous page)

```
        "updatefun2": "function(doc,req) {/* function code here - see below */}"
    },
    "filters": {
        "filterfunction1": "function(doc, req){ /* function code here - see below */ }
↪"
    },
    "validate_doc_update": "function(newDoc, oldDoc, userCtx, secObj) { /* function␣
↪code here - see below */ }",
    "language": "javascript"
}
```

As you can see, a design document can include multiple functions of the same type. The example defines two views, both of which have a map function and one of which has a reduce function. It also defines two update functions and one filter function. The Validate Document Update function is a special case, as each design document cannot contain more than one of those.

3.1.2 View Functions

Views are the primary tool used for querying and reporting on CouchDB databases.

Map Functions

mapfun(*doc*)

Arguments

- **doc** – The document that is being processed

Map functions accept a single document as the argument and (optionally) *emit()* key/value pairs that are stored in a view.

```
function (doc) {
  if (doc.type === 'post' && doc.tags && Array.isArray(doc.tags)) {
    doc.tags.forEach(function (tag) {
      emit(tag.toLowerCase(), 1);
    });
  }
}
```

In this example a key/value pair is emitted for each value in the *tags* array of a document with a *type* of "post". Note that *emit()* may be called many times for a single document, so the same document may be available by several different keys.

Also keep in mind that each document is *sealed* to prevent the situation where one map function changes document state and another receives a modified version.

For efficiency reasons, documents are passed to a group of map functions - each document is processed by a group of map functions from all views of the related design document. This means that if you trigger an index update for one view in the design document, all others will get updated too.

Since version *1.1.0*, *map* supports *CommonJS* modules and the *require()* function.

Reduce and Rereduce Functions

redfun(*keys*, *values*[, *rereduce*])

Arguments

- **keys** – Array of pairs of key-docid for related map function results. Always `null` if rereduce is running (has `true` value).
- **values** – Array of map function result values.
- **rereduce** – Boolean flag to indicate a rereduce run.

Returns
Reduces *values*

Reduce functions take two required arguments of keys and values lists - the result of the related map function - and an optional third value which indicates if *rereduce* mode is active or not. *Rereduce* is used for additional reduce values list, so when it is `true` there is no information about related *keys* (first argument is `null`).

Note that if the result of a *reduce* function is longer than the initial values list then a Query Server error will be raised. However, this behavior can be disabled by setting `reduce_limit` config option to `false`:

```
[query_server_config]
reduce_limit = false
```

While disabling `reduce_limit` might be useful for debug proposes, remember that the main task of reduce functions is to *reduce* the mapped result, not to make it bigger. Generally, your reduce function should converge rapidly to a single value - which could be an array or similar object.

Built-in Reduce Functions

Additionally, CouchDB has a set of built-in reduce functions. These are implemented in Erlang and run inside CouchDB, so they are much faster than the equivalent JavaScript functions.

_approx_count_distinct

New in version 2.2.

Approximates the number of distinct keys in a view index using a variant of the HyperLogLog algorithm. This algorithm enables an efficient, parallelizable computation of cardinality using fixed memory resources. CouchDB has configured the underlying data structure to have a relative error of ~2%.

As this reducer ignores the emitted values entirely, an invocation with `group=true` will simply return a value of 1 for every distinct key in the view. In the case of array keys, querying the view with a `group_level` specified will return the number of distinct keys that share the common group prefix in each row. The algorithm is also cognizant of the `startkey` and `endkey` boundaries and will return the number of distinct keys within the specified key range.

A final note regarding Unicode collation: this reduce function uses the binary representation of each key in the index directly as input to the HyperLogLog filter. As such, it will (incorrectly) consider keys that are not byte identical but that compare equal according to the Unicode collation rules to be distinct keys, and thus has the potential to overestimate the cardinality of the key space if a large number of such keys exist.

_count

Counts the number of values in the index with a given key. This could be implemented in JavaScript as:

```
// could be replaced by _count
function(keys, values, rereduce) {
    if (rereduce) {
        return sum(values);
    } else {
        return values.length;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

_stats

Computes the following quantities for numeric values associated with each key: `sum`, `min`, `max`, `count`, and `sumsqr`. The behavior of the `_stats` function varies depending on the output of the map function. The simplest case is when the map phase emits a single numeric value for each key. In this case the `_stats` function is equivalent to the following JavaScript:

```
// could be replaced by _stats
function(keys, values, rereduce) {
    if (rereduce) {
        return {
            'sum': values.reduce(function(a, b) { return a + b.sum }, 0),
            'min': values.reduce(function(a, b) { return Math.min(a, b.min) },␣
↪Infinity),
            'max': values.reduce(function(a, b) { return Math.max(a, b.max) }, -
↪Infinity),
            'count': values.reduce(function(a, b) { return a + b.count }, 0),
            'sumsqr': values.reduce(function(a, b) { return a + b.sumsqr }, 0)
        }
    } else {
        return {
            'sum': sum(values),
            'min': Math.min.apply(null, values),
            'max': Math.max.apply(null, values),
            'count': values.length,
            'sumsqr': (function() {
            var sumsqr = 0;

            values.forEach(function (value) {
                sumsqr += value * value;
            });

            return sumsqr;
            })(),
        }
    }
}
```

The `_stats` function will also work with "pre-aggregated" values from a map phase. A map function that emits an object containing `sum`, `min`, `max`, `count`, and `sumsqr` keys and numeric values for each can use the `_stats` function to combine these results with the data from other documents. The emitted object may contain other keys (these are ignored by the reducer), and it is also possible to mix raw numeric values and pre-aggregated objects in a single view and obtain the correct aggregated statistics.

Finally, `_stats` can operate on key-value pairs where each value is an array comprised of numbers or pre-aggregated objects. In this case **every** value emitted from the map function must be an array, and the arrays must all be the same length, as `_stats` will compute the statistical quantities above *independently* for each element in the array. Users who want to compute statistics on multiple values from a single document should either `emit` each value into the index separately, or compute the statistics for the set of values using the JavaScript example above and emit a pre-aggregated object.

_sum

In its simplest variation, `_sum` sums the numeric values associated with each key, as in the following JavaScript:

```
// could be replaced by _sum
function(keys, values) {
    return sum(values);
}
```

As with `_stats`, the `_sum` function offers a number of extended capabilities. The `_sum` function requires that map values be numbers, arrays of numbers, or objects. When presented with array output from a map function, `_sum` will compute the sum for every element of the array. A bare numeric value will be treated as an array with a single element, and arrays with fewer elements will be treated as if they contained zeroes for every additional element in the longest emitted array. As an example, consider the following map output:

```
{"total_rows":5, "offset":0, "rows": [
    {"id":"id1", "key":"abc", "value": 2},
    {"id":"id2", "key":"abc", "value": [3,5,7]},
    {"id":"id2", "key":"def", "value": [0,0,0,42]},
    {"id":"id2", "key":"ghi", "value": 1},
    {"id":"id1", "key":"ghi", "value": 3}
]}
```

The `_sum` for this output without any grouping would be:

```
{"rows": [
    {"key":null, "value": [9,5,7,42]}
]}
```

while the grouped output would be

```
{"rows": [
    {"key":"abc", "value": [5,5,7]},
    {"key":"def", "value": [0,0,0,42]},
    {"key":"ghi", "value": 4
]}
```

This is in contrast to the behavior of the `_stats` function which requires that all emitted values be arrays of identical length if any array is emitted.

It is also possible to have `_sum` recursively descend through an emitted object and compute the sums for every field in the object. Objects *cannot* be mixed with other data structures. Objects can be arbitrarily nested, provided that the values for all fields are themselves numbers, arrays of numbers, or objects.

Note: Why don't reduce functions support CommonJS modules?

While *map* functions have limited access to stored modules through `require()`, there is no such feature for *reduce* functions. The reason lies deep inside the way *map* and *reduce* functions are processed by the Query Server. Let's take a look at *map* functions first:

1. CouchDB sends all *map* functions in a processed design document to the Query Server.
2. the Query Server handles them one by one, compiles and puts them onto an internal stack.
3. after all *map* functions have been processed, CouchDB will send the remaining documents for indexing, one by one.
4. the Query Server receives the document object and applies it to every function from the stack. The emitted results are then joined into a single array and sent back to CouchDB.

Now let's see how *reduce* functions are handled:

1. CouchDB sends *as a single command* the list of available *reduce* functions with the result list of key-value pairs that were previously returned from the *map* functions.

2. the Query Server compiles the reduce functions and applies them to the key-value lists. The reduced result is sent back to CouchDB.

As you may note, *reduce* functions are applied in a single shot to the map results while *map* functions are applied to documents one by one. This means that it's possible for *map* functions to precompile CommonJS libraries and use them during the entire view processing, but for *reduce* functions they would be compiled again and again for each view result reduction, which would lead to performance degradation.

3.1.3 Show Functions

Warning: Show functions are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

showfun(*doc*, *req*)

Arguments

- **doc** – The document that is being processed; may be omitted.
- **req** – *Request object*.

Returns
Response object

Return type
object or string

Show functions are used to represent documents in various formats, commonly as HTML pages with nice formatting. They can also be used to run server-side functions without requiring a pre-existing document.

Basic example of show function could be:

```
function(doc, req){
    if (doc) {
        return "Hello from " + doc._id + "!";
    } else {
        return "Hello, world!";
    }
}
```

Also, there is more simple way to return json encoded data:

```
function(doc, req){
    return {
        'json': {
            'id': doc['_id'],
            'rev': doc['_rev']
        }
    }
}
```

and even files (this one is CouchDB logo):

```
function(doc, req){
    return {
        'headers': {
            'Content-Type' : 'image/png',
        },
        'base64': ''.concat(
```

(continues on next page)

(continued from previous page)

```
            'iVBORw0KGgoAAAANSUhEUgAAABAAAAAQCAMAAAAoLQ9TAAAAsV',
            'BMVEUAAAD///////////////////////5ur3rEBn///////////////wDBL/',
            'AADuBAe9EB3IEBz/7+//X1/qBQn2AgP/f3/ilpzsDxfpChDtDhXeCA76AQH/v7',
            '/84eLyWV/uc3bJPEf/Dw/uw8bRWmP1h4zxSlD6YGHuQ0f6g4XyQkXvCA36MDH6',
            'wMH/z8/yAwX64ODeh47BHiv/Ly/20dLQLTj98PDXWmP/Pz//39/wGyJ7Iy9JAA',
            'AADHRSTlMAbw8vf08/bz+Pv19jK/W3AAAAg0lEQVR4Xp3LRQ4DQRBD0QqTm4Y5',
            'zMxw/4OleiJlHeUtv2X6RbNO1Uqj9g0RMCuQO0vBIg4vMFeOpCWIWmDOw82fZx',
            'vaND1c8OG4vrdOqD8YwgpDYDxRgkSm5rwu0nQVBJuMg++pLXZyr5jnc1BaH4GT',
            'LvEliY253nA3pVhQqdPt0f/erJkMGMB8xucAAAAASUVORK5CYII=')
    }
}
```

But what if you need to represent data in different formats via a single function? Functions *registerType()* and *provides()* are your the best friends in that question:

```
function(doc, req){
    provides('json', function(){
        return {'json': doc}
    });
    provides('html', function(){
        return '<pre>' + toJSON(doc) + '</pre>'
    })
    provides('xml', function(){
        return {
            'headers': {'Content-Type': 'application/xml'},
            'body' : ''.concat(
                '<?xml version="1.0" encoding="utf-8"?>\n',
                '<doc>',
                (function(){
                    escape = function(s){
                        return s.replace(/&quot;/g, '"')
                                .replace(/&gt;/g, '>')
                                .replace(/&lt;/g, '<')
                                .replace(/&amp;/g, '&');
                    };
                    var content = '';
                    for(var key in doc){
                        if(!doc.hasOwnProperty(key)) continue;
                        var value = escape(toJSON(doc[key]));
                        var key = escape(key);
                        content += ''.concat(
                            '<' + key + '>',
                            value
                            '</' + key + '>'
                        )
                    }
                    return content;
                })(),
                '</doc>'
            )
        }
    })
    registerType('text-json', 'text/json')
    provides('text-json', function(){
        return toJSON(doc);
    })
}
```

This function may return *html*, *json* , *xml* or our custom *text json* format representation of same document object with same processing rules. Probably, the *xml* provider in our function needs more care to handle nested objects correctly, and keys with invalid characters, but you've got the idea!

See also:

CouchDB Guide:

- Show Functions

3.1.4 List Functions

Warning: List functions are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

`listfun`(*head*, *req*)

Arguments

- **head** – *View Head Information*
- **req** – *Request object*.

Returns
Last chunk.

Return type
string

While *Show Functions* are used to customize document presentation, *List Functions* are used for the same purpose, but on *View Functions* results.

The following list function formats the view and represents it as a very simple HTML page:

```
function(head, req){
    start({
        'headers': {
            'Content-Type': 'text/html'
        }
    });
    send('<html><body><table>');
    send('<tr><th>ID</th><th>Key</th><th>Value</th></tr>');
    while(row = getRow()){
        send(''.concat(
            '<tr>',
            '<td>' + toJSON(row.id) + '</td>',
            '<td>' + toJSON(row.key) + '</td>',
            '<td>' + toJSON(row.value) + '</td>',
            '</tr>'
        ));
    }
    send('</table></body></html>');
}
```

Templates and styles could obviously be used to present data in a nicer fashion, but this is an excellent starting point. Note that you may also use `registerType()` and `provides()` functions in a similar way as for *Show Functions*! However, note that `provides()` expects the return value to be a string when used inside a list function, so you'll need to use `start()` to set any custom headers and stringify your JSON before returning it.

See also:

CouchDB Guide:

- Transforming Views with List Functions

3.1.5 Update Functions

updatefun(*doc*, *req*)

Arguments

- **doc** – The document that is being processed.
- **req** – *Request object*

Returns

Two-element array: the first element is the (updated or new) document, which is committed to the database. If the first element is `null` no document will be committed to the database. If you are updating an existing document, it should already have an `_id` set, and if you are creating a new document, make sure to set its `_id` to something, either generated based on the input or the `req.uuid` provided. The second element is the response that will be sent back to the caller.

Update handlers are functions that clients can request to invoke server-side logic that will create or update a document. This feature allows a range of use cases such as providing a server-side last modified timestamp, updating individual fields in a document without first getting the latest revision, etc.

When the request to an update handler includes a document ID in the URL, the server will provide the function with the most recent version of that document. You can provide any other values needed by the update handler function via the `POST/PUT` entity body or query string parameters of the request.

A basic example that demonstrates all use-cases of update handlers:

```
function(doc, req){
    if (!doc){
        if ('id' in req && req['id']){
            // create new document
            return [{'_id': req['id']}, 'New World']
        }
        // change nothing in database
        return [null, 'Empty World']
    }
    doc['world'] = 'hello';
    doc['edited_by'] = req['userCtx']['name']
    return [doc, 'Edited World!']
}
```

3.1.6 Filter Functions

filterfun(*doc*, *req*)

Arguments

- **doc** – The document that is being processed
- **req** – *Request object*

Returns

Boolean value: `true` means that *doc* passes the filter rules, `false` means that it does not.

Filter functions mostly act like *Show Functions* and *List Functions*: they format, or *filter* the *changes feed*.

Classic Filters

By default the changes feed emits all database documents changes. But if you're waiting for some special changes, processing all documents is inefficient.

Filters are special design document functions that allow the changes feed to emit only specific documents that pass filter rules.

Let's assume that our database is a mailbox and we need to handle only new mail events (documents with the status *new*). Our filter function would look like this:

```
function(doc, req){
    // we need only `mail` documents
    if (doc.type != 'mail'){
        return false;
    }
    // we're interested only in `new` ones
    if (doc.status != 'new'){
        return false;
    }
    return true; // passed!
}
```

Filter functions must return `true` if a document passed all the rules. Now, if you apply this function to the changes feed it will emit only changes about "new mails":

```
GET /somedatabase/_changes?filter=mailbox/new_mail HTTP/1.1
```


```
{"results":[
{"seq":"1-g1AAAAF9eJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBMZc4EC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HqQ_
↪kQG3qgSQqnoCqvJYgCRDA5ACKpxPWOUCiMr9hFUegKi8T1jlA4hKkDuzAC2yZRo","id":
↪"df8eca9da37dade42ee4d7aa3401f1dd","changes":[{"rev":"1-
↪c2e0085a21d34fa1cecb6dc26a4ae657"}]},
{"seq":"9-
↪g1AAAAIreJyVkEsKwjAURUMrqCOXoCuQ5MU0OrI70XyppcaRY92J7kR3ojupaSPUUgqWwAu85By4t0AITbJYo5k7aUNSAnyJ_
↪SGFf4gEkvOyLPMsFtHRL8ZKaC1M0v3eq5ALP-X2a0G1xYKhgnONpmenjT04o_v5tOJ3LV5itTES_
↪uP3FX9ppcAACaVsQAo38hNd_eVFt8ZklVljPqSPYLoH06PJhG0Cxq7-yhQcz-B4_
↪fQCjFuqBjjewVF3E9cORoExSrpU_gHBTo5m","id":"df8eca9da37dade42ee4d7aa34024714",
↪"changes":[{"rev":"1-29d748a6e87b43db967fe338bcb08d74"}]},
],
"last_seq":"10-
↪g1AAAAIreJyVkEsKwjAURR9tQR25BF2B5GMaHdmdaNIk1FLjyLHuRHeiO9Gd1LQRaimFlsALvOQcuLcAgGkWKpjbs9I4wYSvkD
↪BALkoyzLPQhGc3GKSCqWEjrvfexVy6abc_SxQWwzRVHCuYHaxSpuj1aqfTyp-3-
↪IlSrdakmH8oeKvrRSIkJhSNiKFjdyEm7uc6N6YTKo3iI_pw5se3vRsMiETE23WgzJ5x8s73n-
↪9EMYNTUc4Pt5RdxPVDkYJYxR3qfwLwW6OZw"}
```


Note that the value of `last_seq` is *10-..*, but we received only two records. Seems like any other changes were for documents that haven't passed our filter.

We probably need to filter the changes feed of our mailbox by more than a single status value. We're also interested in statuses like "spam" to update spam-filter heuristic rules, "outgoing" to let a mail daemon actually send mails, and so on. Creating a lot of similar functions that actually do similar work isn't good idea - so we need a dynamic filter.

You may have noticed that filter functions take a second argument named *request*. This allows the creation of dynamic filters based on query parameters, *user context* and more.

The dynamic version of our filter looks like this:

```
function(doc, req){
    // we need only `mail` documents
    if (doc.type != 'mail'){
        return false;
    }
    // we're interested only in requested status
    if (doc.status != req.query.status){
        return false;
    }
    return true; // passed!
}
```

and now we have passed the *status* query parameter in the request to let our filter match only the required documents:

```
GET /somedatabase/_changes?filter=mailbox/by_status&status=new HTTP/1.1
```


```
{"results":[
{"seq":"1-g1AAAAF9eJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBMZc4EC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HqQ_
↪kQG3qgSQqnoCqvJYgCRDA5ACKpxPWOUCiMr9hFUegKi8T1jlA4hKkDuzAC2yZRo","id":
↪"df8eca9da37dade42ee4d7aa3401f1dd","changes":[{"rev":"1-
↪c2e0085a21d34fa1cecb6dc26a4ae657"}]},
{"seq":"9-
↪g1AAAAIreJyVkEsKwjAURUMrqCOXoCuQ5MU0OrI70XyppcaRY92J7kR3ojupaSPUUgqWwAu85By4t0AITbJYo5k7aUNSAnyJ_
↪SGFf4gEkvOyLPMsFtHRL8ZKaC1M0v3eq5ALP-X2a0G1xYKhgnONpmenjT04o_v5tOJ3LV5itTES_
↪uP3FX9ppcAACaVsQAo38hNd_eVFt8ZklVljPqSPYLoH06PJhG0Cxq7-yhQcz-B4_
↪fQCjFuqBjjewVF3E9cORoExSrpU_gHBTo5m","id":"df8eca9da37dade42ee4d7aa34024714",
↪"changes":[{"rev":"1-29d748a6e87b43db967fe338bcb08d74"}]},
],
"last_seq":"10-
↪g1AAAAIreJyVkEsKwjAURR9tQR25BF2B5GMaHdmdaNIk1FLjyLHuRHeiO9Gd1LQRaimFlsALvOQcuLcAgGkWKpjbs9I4wYSvkD
↪BALkoyzLPQhGc3GKSCqWEjrvfexVy6abc_SxQWwzRVHCuYHaxSpuj1aqfTyp-3-
↪IlSrdakmH8oeKvrRSIkJhSNiKFjdyEm7uc6N6YTKo3iI_pw5se3vRsMiETE23WgzJ5x8s73n-
↪9EMYNTUc4Pt5RdxPVDkYJYxR3qfwLwW6OZw"}
```


and we can easily change filter behavior with:

```
GET /somedatabase/_changes?filter=mailbox/by_status&status=spam HTTP/1.1
```


```
{"results":[
{"seq":"6-g1AAAAIreJyVkM0JwjAYQD9bQT05gk4gaWIaPdlNNL_UUuPJs26im-gmuklMjVClFFoCXyDJe_
↪BSAsA4jxVM7VHpJEswWyC_ktJfRBzEzDlX5DGPDv5gJLlSXKfN560KMfdTbL4W-
↪FgM1oQzpmByskqbvdWqnc8qfvvHCyTXWuBu_K7iz38VCOOUENqjwg79hIvfvOhamQahROoVYn3-
↪I5huwXSvm5BJsTbLTk3B8QiO58-_YMoMkT0cr-BwdRElmFKSNKniDcAcjmM","id":
↪"8960e91220798fc9f9d29d24ed612e0d","changes":[{"rev":"3-
↪cc6ff71af716ddc2ba114967025c0ee0"}]},
],
"last_seq":"10-
↪g1AAAAIreJyVkEsKwjAURR9tQR25BF2B5GMaHdmdaNIk1FLjyLHuRHeiO9Gd1LQRaimFlsALvOQcuLcAgGkWKpjbs9I4wYSvkD
↪BALkoyzLPQhGc3GKSCqWEjrvfexVy6abc_SxQWwzRVHCuYHaxSpuj1aqfTyp-3-
↪IlSrdakmH8oeKvrRSIkJhSNiKFjdyEm7uc6N6YTKo3iI_pw5se3vRsMiETE23WgzJ5x8s73n-
↪9EMYNTUc4Pt5RdxPVDkYJYxR3qfwLwW6OZw"}
```


Combining filters with a *continuous* feed allows creating powerful event-driven systems.

View Filters

View filters are the same as classic filters above, with one small difference: they use the *map* instead of the *filter* function of a view, to filter the changes feed. Each time a key-value pair is emitted from the *map* function, a change is returned. This allows avoiding filter functions that mostly do the same work as views.

To use them just pass *filter=_view* and *view=designdoc/viewname* as request parameters to the *changes feed*:

```
GET /somedatabase/_changes?filter=_view&view=dname/viewname  HTTP/1.1
```

Note: Since view filters use *map* functions as filters, they can't show any dynamic behavior since *request object* is not available.

See also:

CouchDB Guide:

- Guide to filter change notification

3.1.7 Validate Document Update Functions

validatefun(*newDoc*, *oldDoc*, *userCtx*, *secObj*)

Arguments

- **newDoc** – New version of document that will be stored.
- **oldDoc** – Previous version of document that is already stored.
- **userCtx** – *User Context Object*
- **secObj** – *Security Object*

Throws
`forbidden` error to gracefully prevent document storing.

Throws
`unauthorized` error to prevent storage and allow the user to re-auth.

A design document may contain a function named *validate_doc_update* which can be used to prevent invalid or unauthorized document update requests from being stored. The function is passed the new document from the update request, the current document stored in the database, a *User Context Object* containing information about the user writing the document (if present), and a *Security Object* with lists of database security roles.

Validation functions typically examine the structure of the new document to ensure that required fields are present and to verify that the requesting user should be allowed to make changes to the document properties. For example, an application may require that a user must be authenticated in order to create a new document or that specific document fields be present when a document is updated. The validation function can abort the pending document write by throwing one of two error objects:

```
// user is not authorized to make the change but may re-authenticate
throw({ unauthorized: 'Error message here.' });

// change is not allowed
throw({ forbidden: 'Error message here.' });
```

Document validation is optional, and each design document in the database may have at most one validation function. When a write request is received for a given database, the validation function in each design document in that database is called in an unspecified order. If any of the validation functions throw an error, the write will not succeed.

Example: The `_design/_auth` ddoc from *_users* database uses a validation function to ensure that documents contain some required fields and are only modified by a user with the `_admin` role:

```
function(newDoc, oldDoc, userCtx, secObj) {
    if (newDoc._deleted === true) {
        // allow deletes by admins and matching users
        // without checking the other fields
        if ((userCtx.roles.indexOf('_admin') !== -1) ||
            (userCtx.name == oldDoc.name)) {
            return;
        } else {
            throw({forbidden: 'Only admins may delete other user docs.'});
        }
    }

    if ((oldDoc && oldDoc.type !== 'user') || newDoc.type !== 'user') {
        throw({forbidden : 'doc.type must be user'});
    } // we only allow user docs for now

    if (!newDoc.name) {
        throw({forbidden: 'doc.name is required'});
    }

    if (!newDoc.roles) {
        throw({forbidden: 'doc.roles must exist'});
    }

    if (!isArray(newDoc.roles)) {
        throw({forbidden: 'doc.roles must be an array'});
    }

    if (newDoc._id !== ('org.couchdb.user:' + newDoc.name)) {
        throw({
            forbidden: 'Doc ID must be of the form org.couchdb.user:name'
        });
    }

    if (oldDoc) { // validate all updates
        if (oldDoc.name !== newDoc.name) {
            throw({forbidden: 'Usernames can not be changed.'});
        }
    }

    if (newDoc.password_sha && !newDoc.salt) {
        throw({
            forbidden: 'Users with password_sha must have a salt.' +
                'See /_utils/script/couch.js for example code.'
        });
    }

    var is_server_or_database_admin = function(userCtx, secObj) {
        // see if the user is a server admin
        if(userCtx.roles.indexOf('_admin') !== -1) {
            return true; // a server admin
        }

        // see if the user a database admin specified by name
        if(secObj && secObj.admins && secObj.admins.names) {
```

(continues on next page)

(continued from previous page)

```
            if(secObj.admins.names.indexOf(userCtx.name) !== -1) {
                return true; // database admin
            }
        }

        // see if the user a database admin specified by role
        if(secObj && secObj.admins && secObj.admins.roles) {
            var db_roles = secObj.admins.roles;
            for(var idx = 0; idx < userCtx.roles.length; idx++) {
                var user_role = userCtx.roles[idx];
                if(db_roles.indexOf(user_role) !== -1) {
                    return true; // role matches!
                }
            }
        }

        return false; // default to no admin
    }

    if (!is_server_or_database_admin(userCtx, secObj)) {
        if (oldDoc) { // validate non-admin updates
            if (userCtx.name !== newDoc.name) {
                throw({
                    forbidden: 'You may only update your own user document.'
                });
            }
            // validate role updates
            var oldRoles = oldDoc.roles.sort();
            var newRoles = newDoc.roles.sort();

            if (oldRoles.length !== newRoles.length) {
                throw({forbidden: 'Only _admin may edit roles'});
            }

            for (var i = 0; i < oldRoles.length; i++) {
                if (oldRoles[i] !== newRoles[i]) {
                    throw({forbidden: 'Only _admin may edit roles'});
                }
            }
        } else if (newDoc.roles.length > 0) {
            throw({forbidden: 'Only _admin may set roles'});
        }
    }

    // no system roles in users db
    for (var i = 0; i < newDoc.roles.length; i++) {
        if (newDoc.roles[i][0] === '_') {
            throw({
                forbidden:
                'No system roles (starting with underscore) in users db.'
            });
        }
    }

    // no system names as names
    if (newDoc.name[0] === '_') {
```

(continues on next page)

(continued from previous page)

```
        throw({forbidden: 'Username may not start with underscore.'});
    }

    var badUserNameChars = [':'];

    for (var i = 0; i < badUserNameChars.length; i++) {
        if (newDoc.name.indexOf(badUserNameChars[i]) >= 0) {
            throw({forbidden: 'Character `' + badUserNameChars[i] +
                    '` is not allowed in usernames.'});
        }
    }
}
```

Note: The `return` statement is used only for function, it has no impact on the validation process.

See also:

CouchDB Guide:

- Validation Functions

3.2 Guide to Views

Views are the primary tool used for querying and reporting on CouchDB documents. There you'll learn how they work and how to use them to build effective applications with CouchDB.

3.2.1 Introduction to Views

Views are useful for many purposes:

- Filtering the documents in your database to find those relevant to a particular process.
- Extracting data from your documents and presenting it in a specific order.
- Building efficient indexes to find documents by any value or structure that resides in them.
- Use these indexes to represent relationships among documents.
- Finally, with views you can make all sorts of calculations on the data in your documents. For example, if documents represent your company's financial transactions, a view can answer the question of what the spending was in the last week, month, or year.

What Is a View?

Let's go through the different use cases. First is extracting data that you might need for a special purpose in a specific order. For a front page, we want a list of blog post titles sorted by date. We'll work with a set of example documents as we walk through how views work:

```
{
    "_id":"biking",
    "_rev":"AE19EBC7654",

    "title":"Biking",
    "body":"My biggest hobby is mountainbiking. The other day...",
    "date":"2009/01/30 18:04:11"
}
```

```
{
    "_id":"bought-a-cat",
    "_rev":"4A3BBEE711",

    "title":"Bought a Cat",
    "body":"I went to the pet store earlier and brought home a little kitty...",
    "date":"2009/02/17 21:13:39"
}
```

```
{
    "_id":"hello-world",
    "_rev":"43FBA4E7AB",

    "title":"Hello World",
    "body":"Well hello and welcome to my new blog...",
    "date":"2009/01/15 15:52:20"
}
```

Three will do for the example. Note that the documents are sorted by "_id", which is how they are stored in the database. Now we define a view. Bear with us without an explanation while we show you some code:

```
function(doc) {
    if(doc.date && doc.title) {
        emit(doc.date, doc.title);
    }
}
```

This is a *map function*, and it is written in JavaScript. If you are not familiar with JavaScript but have used C or any other C-like language such as Java, PHP, or C#, this should look familiar. It is a simple function definition.

You provide CouchDB with view functions as strings stored inside the `views` field of a design document. To create this view you can use this command:

```
curl -X PUT http://admin:password@127.0.0.1:5984/db/_design/my_ddoc
     -d '{"views":{"my_filter":{"map":
         "function(doc) { if(doc.date && doc.title) { emit(doc.date, doc.title); }}"}}
↪}'
```

You don't run the JavaScript function yourself. Instead, when you *query your view*, CouchDB takes the source code and runs it for you on every document in the database your view was defined in. You *query your view* to retrieve the *view result* using the following command:

```
curl -X GET http://admin:password@127.0.0.1:5984/db/_design/my_ddoc/_view/my_filter
```

All map functions have a single parameter doc. This is a single document in the database. Our map function checks whether our document has a `date` and a `title` attribute — luckily, all of our documents have them — and then calls the built-in *`emit()`* function with these two attributes as arguments.

The *`emit()`* function always takes two arguments: the first is `key`, and the second is `value`. The `emit(key, value)` function creates an entry in our *view result*. One more thing: the `emit()` function can be called multiple times in the map function to create multiple entries in the view results from a single document, but we are not doing that yet.

CouchDB takes whatever you pass into the emit() function and puts it into a list (see Table 1, "View results" below). Each row in that list includes the *key* and *value*. More importantly, the list is sorted by key (by `doc.date` in our case). The most important feature of a view result is that it is sorted by *key*. We will come back to that over and over again to do neat things. Stay tuned.

Table 1. View results:

Key	Value
"2009/01/15 15:52:20"	"Hello World"
"2009/01/30 18:04:11"	"Biking"
"2009/02/17 21:13:39"	"Bought a Cat"

When you query your view, CouchDB takes the source code and runs it for you on every document in the database. If you have a lot of documents, that takes quite a bit of time and you might wonder if it is not horribly inefficient to do this. Yes, it would be, but CouchDB is designed to avoid any extra costs: it only runs through all documents once, when you first query your view. If a document is changed, the map function is only run once, to recompute the keys and values for that single document.

The view result is stored in a B-tree, just like the structure that is responsible for holding your documents. View B-trees are stored in their own file, so that for high-performance CouchDB usage, you can keep views on their own disk. The B-tree provides very fast lookups of rows by key, as well as efficient streaming of rows in a key range. In our example, a single view can answer all questions that involve time: "Give me all the blog posts from last week" or "last month" or "this year." Pretty neat.

When we query our view, we get back a list of all documents sorted by date. Each row also includes the post title so we can construct links to posts. Table 1 is just a graphical representation of the view result. The actual result is JSON-encoded and contains a little more metadata:

```
{
    "total_rows": 3,
    "offset": 0,
    "rows": [
        {
            "key": "2009/01/15 15:52:20",
            "id": "hello-world",
            "value": "Hello World"
        },

        {
            "key": "2009/01/30 18:04:11",
            "id": "biking",
            "value": "Biking"
        },

        {
            "key": "2009/02/17 21:13:39",
            "id": "bought-a-cat",
            "value": "Bought a Cat"
        }

    ]
}
```

Now, the actual result is not as nicely formatted and doesn't include any superfluous whitespace or newlines, but this is better for you (and us!) to read and understand. Where does that "id" member in the result rows come from? That wasn't there before. That's because we omitted it earlier to avoid confusion. CouchDB automatically includes the document ID of the document that created the entry in the view result. We'll use this as well when constructing links to the blog post pages.

Warning: Do not emit the entire document as the value of your `emit(key, value)` statement unless you're sure you know you want it. This stores an entire additional copy of your document in the view's secondary index. Views with `emit(key, doc)` take longer to update, longer to write to disk, and consume significantly more disk space. The only advantage is that they are faster to query than using the `?include_docs=true`

parameter when querying a view.

Consider the trade-offs before emitting the entire document. Often it is sufficient to emit only a portion of the document, or just a single key / value pair, in your views.

Efficient Lookups

Let's move on to the second use case for views: "building efficient indexes to find documents by any value or structure that resides in them." We already explained the efficient indexing, but we skipped a few details. This is a good time to finish this discussion as we are looking at map functions that are a little more complex.

First, back to the B-trees! We explained that the B-tree that backs the key-sorted view result is built only once, when you first query a view, and all subsequent queries will just read the B-tree instead of executing the map function for all documents again. What happens, though, when you change a document, add a new one, or delete one? Easy: CouchDB is smart enough to find the rows in the view result that were created by a specific document. It marks them invalid so that they no longer show up in view results. If the document was deleted, we're good — the resulting B-tree reflects the state of the database. If a document got updated, the new document is run through the map function and the resulting new lines are inserted into the B-tree at the correct spots. New documents are handled in the same way. The B-tree is a very efficient data structure for our needs, and the crash-only design of CouchDB databases is carried over to the view indexes as well.

To add one more point to the efficiency discussion: usually multiple documents are updated between view queries. The mechanism explained in the previous paragraph gets applied to all changes in the database since the last time the view was queried in a batch operation, which makes things even faster and is generally a better use of your resources.

Find One

On to more complex map functions. We said "find documents by any value or structure that resides in them." We already explained how to extract a value by which to sort a list of views (our date field). The same mechanism is used for fast lookups. The URI to query to get a view's result is `/database/_design/designdocname/_view/viewname`. This gives you a list of all rows in the view. We have only three documents, so things are small, but with thousands of documents, this can get long. You can add view parameters to the URI to constrain the result set. Say we know the date of a blog post. To find a single document, we would use `/blog/_design/docs/_view/by_date?key="2009/01/30 18:04:11"` to get the "Biking" blog post. Remember that you can place whatever you like in the key parameter to the emit() function. Whatever you put in there, we can now use to look up exactly — and fast.

Note that in the case where multiple rows have the same key (perhaps we design a view where the key is the name of the post's author), key queries can return more than one row.

Find Many

We talked about "getting all posts for last month." If it's February now, this is as easy as:

```
/blog/_design/docs/_view/by_date?startkey="2010/01/01 00:00:00"&endkey="2010/02/00␣
↪00:00:00"
```

The `startkey` and `endkey` parameters specify an inclusive range on which we can search.

To make things a little nicer and to prepare for a future example, we are going to change the format of our date field. Instead of a string, we are going to use an array, where individual members are part of a timestamp in decreasing significance. This sounds fancy, but it is rather easy. Instead of:

```
{
    "date": "2009/01/31 00:00:00"
}
```

we use:

```
{
    "date": [2009, 1, 31, 0, 0, 0]
}
```

Our map function does not have to change for this, but our view result looks a little different:

Table 2. New view results:

Key	Value
[2009, 1, 15, 15, 52, 20]	"Hello World"
[2009, 2, 17, 21, 13, 39]	"Biking"
[2009, 1, 30, 18, 4, 11]	"Bought a Cat"

And our queries change to:

```
/blog/_design/docs/_view/by_date?startkey=[2010, 1, 1, 0, 0, 0]&endkey=[2010, 2, 1, 0,
↪ 0, 0]
```

For all you care, this is just a change in syntax, not meaning. But it shows you the power of views. Not only can you construct an index with scalar values like strings and integers, you can also use JSON structures as keys for your views. Say we tag our documents with a list of tags and want to see all tags, but we don't care for documents that have not been tagged.

```
{
    ...
    tags: ["cool", "freak", "plankton"],
    ...
}
```

```
{
    ...
    tags: [],
    ...
}
```

```
function(doc) {
    if(doc.tags.length > 0) {
        for(var idx in doc.tags) {
            emit(doc.tags[idx], null);
        }
    }
}
```

This shows a few new things. You can have conditions on structure (`if(doc.tags.length > 0)`) instead of just values. This is also an example of how a map function calls *emit()* multiple times per document. And finally, you can pass null instead of a value to the value parameter. The same is true for the key parameter. We'll see in a bit how that is useful.

Reversed Results

To retrieve view results in reverse order, use the `descending=true` query parameter. If you are using a `startkey` parameter, you will find that CouchDB returns different rows or no rows at all. What's up with that?

It's pretty easy to understand when you see how view query options work under the hood. A view is stored in a tree structure for fast lookups. Whenever you query a view, this is how CouchDB operates:

1. Starts reading at the top, or at the position that `startkey` specifies, if present.
2. Returns one row at a time until the end or until it hits `endkey`, if present.

If you specify `descending=true`, the reading direction is reversed, not the sort order of the rows in the view. In addition, the same two-step procedure is followed.

Say you have a view result that looks like this:

Key	Value
0	"foo"
1	"bar"
2	"baz"

Here are potential query options: `?startkey=1&descending=true`. What will CouchDB do? See #1 above: it jumps to `startkey`, which is the row with the key 1, and starts reading backward until it hits the end of the view. So the particular result would be:

Key	Value
1	"bar"
0	"foo"

This is very likely not what you want. To get the rows with the indexes 1 and 2 in reverse order, you need to switch the `startkey` to `endkey`: `endkey=1&descending=true`:

Key	Value
2	"baz"
1	"bar"

Now that looks a lot better. CouchDB started reading at the bottom of the view and went backward until it hit `endkey`.

The View to Get Comments for Posts

We use an array key here to support the `group_level` reduce query parameter. CouchDB's views are stored in the B-tree file structure. Because of the way B-trees are structured, we can cache the intermediate reduce results in the non-leaf nodes of the tree, so reduce queries can be computed along arbitrary key ranges in logarithmic time. See Figure 1, "Comments map function".

In the blog app, we use `group_level` reduce queries to compute the count of comments both on a per-post and total basis, achieved by querying the same view index with different methods. With some array keys, and assuming each key has the value 1:

```
["a","b","c"]
["a","b","e"]
["a","c","m"]
["b","a","c"]
["b","a","g"]
```

the reduce view:

```
function(keys, values, rereduce) {
    return sum(values)
}
```

or:

```
_sum
```

which is a built-in CouchDB reduce function (the others are `_count` and `_stats`). `_sum` here returns the total number of rows between the start and end key. So with `startkey=["a","b"]&endkey=["b"]` (which includes the first three of the above keys) the result would equal `3`. The effect is to count rows. If you'd like to count rows without depending on the row value, you can switch on the `rereduce` parameter:

```
function(keys, values, rereduce) {
    if (rereduce) {
        return sum(values);
    } else {
        return values.length;
    }
}
```

Note: The JavaScript function above could be effectively replaced by the built-in `_count`.

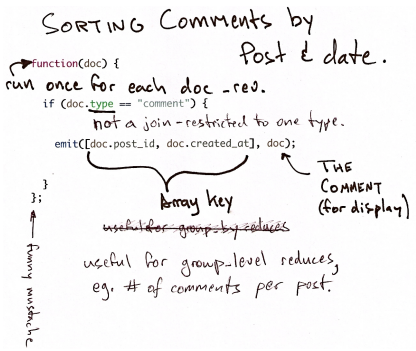

Fig. 1: Figure 1. Comments map function

This is the reduce view used by the example app to count comments, while utilizing the map to output the comments, which are more useful than just `1` over and over. It pays to spend some time playing around with map and reduce functions. Fauxton is OK for this, but it doesn't give full access to all the query parameters. Writing your own test code for views in your language of choice is a great way to explore the nuances and capabilities of CouchDB's incremental MapReduce system.

Anyway, with a `group_level` query, you're basically running a series of reduce range queries: one for each group that shows up at the level you query. Let's reprint the key list from earlier, grouped at level `1`:

```
["a"]    3
["b"]    2
```

And at `group_level=2`:

```
["a","b"]    2
["a","c"]    1
["b","a"]    2
```

Using the parameter `group=true` makes it behave as though it were `group_level=999`, so in the case of our current example, it would give the number 1 for each key, as there are no exactly duplicated keys.

Reduce/Rereduce

We briefly talked about the `rereduce` parameter to the reduce function. We'll explain what's up with it in this section. By now, you should have learned that your view result is stored in B-tree index structure for efficiency. The existence and use of the `rereduce` parameter is tightly coupled to how the B-tree index works.

Consider the map result are:

```
"afrikaans", 1
"afrikaans", 1
"chinese", 1
"chinese", 1
"chinese", 1
"chinese", 1
"french", 1
"italian", 1
"italian", 1
"spanish", 1
"vietnamese", 1
"vietnamese", 1
```

Example 1. Example view result (mmm, food)

When we want to find out how many dishes there are per origin, we can reuse the simple reduce function shown earlier:

```
function(keys, values, rereduce) {
    return sum(values);
}
```

Figure 2, "The B-tree index" shows a simplified version of what the B-tree index looks like. We abbreviated the key strings.

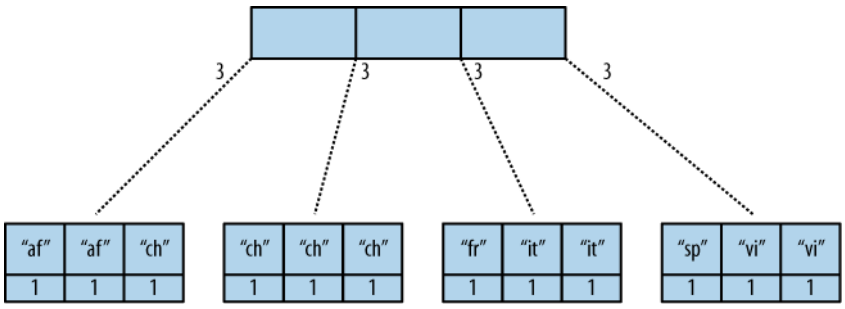

Fig. 2: Figure 2. The B-tree index

The view result is what computer science grads call a "pre-order" walk through the tree. We look at each element in each node starting from the left. Whenever we see that there is a subnode to descend into, we descend and start reading the elements in that subnode. When we have walked through the entire tree, we're done.

You can see that CouchDB stores both keys and values inside each leaf node. In our case, it is simply always 1, but you might have a value where you count other results and then all rows have a different value. What's important is that CouchDB runs all elements that are within a node into the reduce function (setting the `rereduce` parameter to false) and stores the result inside the parent node along with the edge to the subnode. In our case, each edge has a 3 representing the reduce value for the node it points to.

Note: In reality, nodes have more than 1,600 elements in them. CouchDB computes the result for all the elements in multiple iterations over the elements in a single node, not all at once (which would be disastrous for memory consumption).

Now let's see what happens when we run a query. We want to know how many "chinese" entries we have. The query option is simple: `?key="chinese"`. See Figure 3, "The B-tree index reduce result".

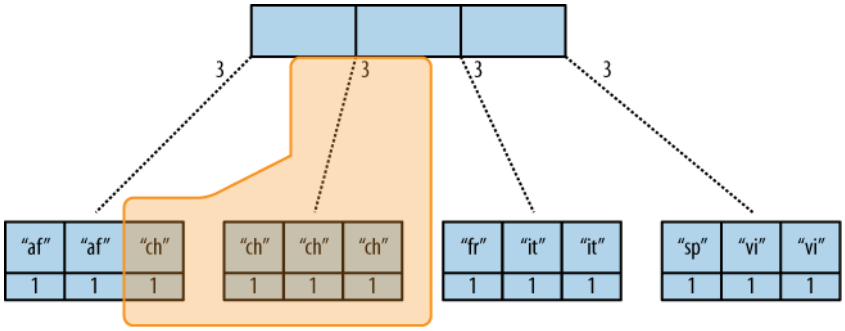

Fig. 3: Figure 3. The B-tree index reduce result

CouchDB detects that all values in the subnode include the "chinese" key. It concludes that it can take just the 3 values associated with that node to compute the final result. It then finds the node left to it and sees that it's a node with keys outside the requested range (`key=` requests a range where the beginning and the end are the same value). It concludes that it has to use the "chinese" element's value and the other node's value and run them through the reduce function with the `rereduce` parameter set to true.

The reduce function effectively calculates 3 + 1 at query time and returns the desired result. The next example shows some pseudocode that shows the last invocation of the reduce function with actual values:

```
function(null, [3, 1], true) {
    return sum([3, 1]);
}
```

Now, we said your reduce function must actually reduce your values. If you see the B-tree, it should become obvious what happens when you don't reduce your values. Consider the following map result and reduce function. This time we want to get a list of all the unique labels in our view:

```
"abc", "afrikaans"
"cef", "afrikaans"
"fhi", "chinese"
"hkl", "chinese"
"ino", "chinese"
"lqr", "chinese"
"mtu", "french"
"owx", "italian"
"qza", "italian"
"tdx", "spanish"
"xfg", "vietnamese"
"zul", "vietnamese"
```

We don't care for the key here and only list all the labels we have. Our reduce function removes duplicates:

```
function(keys, values, rereduce) {
    var unique_labels = {};
    values.forEach(function(label) {
        if(!unique_labels[label]) {
            unique_labels[label] = true;
```

(continues on next page)

(continued from previous page)

```
        }
    });

    return unique_labels;
}
```

This translates to Figure 4, "An overflowing reduce index".

We hope you get the picture. The way the B-tree storage works means that if you don't actually reduce your data in the reduce function, you end up having CouchDB copy huge amounts of data around that grow linearly, if not faster, with the number of rows in your view.

CouchDB will be able to compute the final result, but only for views with a few rows. Anything larger will experience a ridiculously slow view build time. To help with that, CouchDB since version 0.10.0 will throw an error if your reduce function does not reduce its input values.

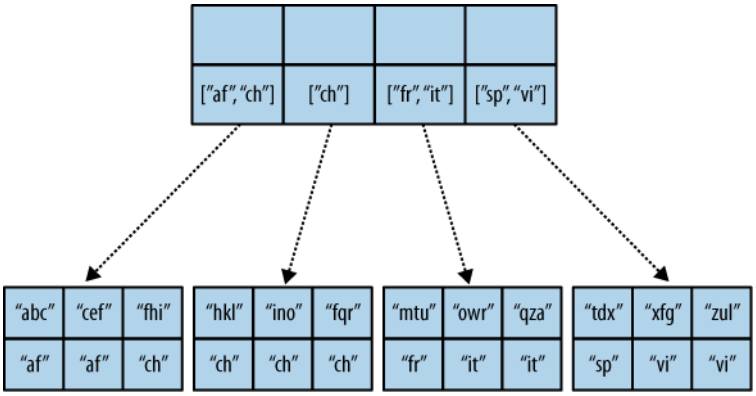

Fig. 4: Figure 4. An overflowing reduce index

One vs. Multiple Design Documents

A common question is: when should I split multiple views into multiple design documents, or keep them together?

Each view you create corresponds to one B-tree. All views in a single design document will live in the same set of index files on disk (one file per database shard; in 2.0+ by default, 8 files per node).

The most practical consideration for separating views into separate documents is how often you change those views. Views that change often, and are in the same design document as other views, will invalidate those other views' indexes when the design document is written, forcing them all to rebuild from scratch. Obviously you will want to avoid this in production!

However, when you have multiple views with the same map function in the same design document, CouchDB will optimize and only calculate that map function once. This lets you have two views with different *reduce* functions (say, one with `_sum` and one with `_stats`) but build only a single copy of the mapped index. It also saves disk space and the time to write multiple copies to disk.

Another benefit of having multiple views in the same design document is that the index files can keep a single index of backwards references from docids to rows. CouchDB needs these "back refs" to invalidate rows in a view when a document is deleted (otherwise, a delete would force a total rebuild!)

One other consideration is that each separate design document will spawn another (set of) `couchjs` processes to generate the view, one per shard. Depending on the number of cores on your server(s), this may be efficient (using all of the idle cores you have) or inefficient (overloading the CPU on your servers). The exact situation will depend on your deployment architecture.

So, should you use one or multiple design documents? The choice is yours.

Lessons Learned

- If you don't use the key field in the map function, you are probably doing it wrong.
- If you are trying to make a list of values unique in the reduce functions, you are probably doing it wrong.
- If you don't reduce your values to a single scalar value or a small fixed-sized object or array with a fixed number of scalar values of small sizes, you are probably doing it wrong.

Wrapping Up

Map functions are side effect–free functions that take a document as argument and *emit* key/value pairs. CouchDB stores the emitted rows by constructing a sorted B-tree index, so row lookups by key, as well as streaming operations across a range of rows, can be accomplished in a small memory and processing footprint, while writes avoid seeks. Generating a view takes `O(N)`, where `N` is the total number of rows in the view. However, querying a view is very quick, as the B-tree remains shallow even when it contains many, many keys.

Reduce functions operate on the sorted rows emitted by map view functions. CouchDB's reduce functionality takes advantage of one of the fundamental properties of B-tree indexes: for every leaf node (a sorted row), there is a chain of internal nodes reaching back to the root. Each leaf node in the B-tree carries a few rows (on the order of tens, depending on row size), and each internal node may link to a few leaf nodes or other internal nodes.

The reduce function is run on every node in the tree in order to calculate the final reduce value. The end result is a reduce function that can be incrementally updated upon changes to the map function, while recalculating the reduction values for a minimum number of nodes. The initial reduction is calculated once per each node (inner and leaf) in the tree.

When run on leaf nodes (which contain actual map rows), the reduce function's third parameter, `rereduce`, is false. The arguments in this case are the keys and values as output by the map function. The function has a single returned reduction value, which is stored on the inner node that a working set of leaf nodes have in common, and is used as a cache in future reduce calculations.

When the reduce function is run on inner nodes, the `rereduce` flag is `true`. This allows the function to account for the fact that it will be receiving its own prior output. When `rereduce` is true, the values passed to the function are intermediate reduction values as cached from previous calculations. When the tree is more than two levels deep, the *rereduce* phase is repeated, consuming chunks of the previous level's output until the final reduce value is calculated at the root node.

A common mistake new CouchDB users make is attempting to construct complex aggregate values with a reduce function. Full reductions should result in a scalar value, like 5, and not, for instance, a JSON hash with a set of unique keys and the count of each. The problem with this approach is that you'll end up with a very large final value. The number of unique keys can be nearly as large as the number of total keys, even for a large set. It is fine to combine a few scalar calculations into one reduce function; for instance, to find the total, average, and standard deviation of a set of numbers in a single function.

If you're interested in pushing the edge of CouchDB's incremental reduce functionality, have a look at Google's paper on Sawzall, which gives examples of some of the more exotic reductions that can be accomplished in a system with similar constraints.

3.2.2 Views Collation

Basics

View functions specify a key and a value to be returned for each row. CouchDB collates the view rows by this key. In the following example, the `LastName` property serves as the key, thus the result will be sorted by `LastName`:

```
function(doc) {
    if (doc.Type == "customer") {
        emit(doc.LastName, {FirstName: doc.FirstName, Address: doc.Address});
```

(continues on next page)

(continued from previous page)

```
    }
}
```

CouchDB allows arbitrary JSON structures to be used as keys. You can use JSON arrays as keys for fine-grained control over sorting and grouping.

Examples

The following clever trick would return both customer and order documents. The key is composed of a customer `_id` and a sorting token. Because the key for order documents begins with the `_id` of a customer document, all the orders will be sorted by customer. Because the sorting token for customers is lower than the token for orders, the customer document will come before the associated orders. The values 0 and 1 for the sorting token are arbitrary.

```
function(doc) {
    if (doc.Type == "customer") {
        emit([doc._id, 0], null);
    } else if (doc.Type == "order") {
        emit([doc.customer_id, 1], null);
    }
}
```

To list a specific customer with `_id` XYZ, and all of that customer's orders, limit the startkey and endkey ranges to cover only documents for that customer's `_id`:

```
startkey=["XYZ"]&endkey=["XYZ", {}]
```

It is not recommended to emit the document itself in the view. Instead, to include the bodies of the documents when requesting the view, request the view with `?include_docs=true`.

Sorting by Dates

It maybe be convenient to store date attributes in a human readable format (i.e. as a *string*), but still sort by date. This can be done by converting the date to a *number* in the *`emit()`* function. For example, given a document with a created_at attribute of `'Wed Jul 23 16:29:21 +0100 2013'`, the following emit function would sort by date:

```
emit(Date.parse(doc.created_at).getTime(), null);
```

Alternatively, if you use a date format which sorts lexicographically, such as `"2013/06/09 13:52:11 +0000"` you can just

```
emit(doc.created_at, null);
```

and avoid the conversion. As a bonus, this date format is compatible with the JavaScript date parser, so you can use `new Date(doc.created_at)` in your client side JavaScript to make date sorting easy in the browser.

String Ranges

If you need start and end keys that encompass every string with a given prefix, it is better to use a high value Unicode character, than to use a `'ZZZZ'` suffix.

That is, rather than:

```
startkey="abc"&endkey="abcZZZZZZZZZ"
```

You should use:

```
startkey="abc"&endkey="abc\uffff0"
```

Collation Specification

This section is based on the view_collation function in view_collation.js:

```
// special values sort before all other types
null
false
true

// then numbers
1
2
3.0
4

// then text, case sensitive
"a"
"A"
"aa"
"b"
"B"
"ba"
"bb"

// then arrays. compared element by element until different.
// Longer arrays sort after their prefixes
["a"]
["b"]
["b","c"]
["b","c", "a"]
["b","d"]
["b","d", "e"]

// then object, compares each key value in the list until different.
// larger objects sort after their subset objects.
{a:1}
{a:2}
{b:1}
{b:2}
{b:2, a:1} // Member order does matter for collation.
           // CouchDB preserves member order
           // but doesn't require that clients will.
           // this test might fail if used with a js engine
           // that doesn't preserve order
{b:2, c:2}
```

Comparison of strings is done using ICU which implements the Unicode Collation Algorithm, giving a dictionary sorting of keys. This can give surprising results if you were expecting ASCII ordering. Note that:

- All symbols sort before numbers and letters (even the "high" symbols like tilde, `0x7e`)
- Differing sequences of letters are compared without regard to case, so `a < aa` but also `A < aa` and `a < AA`
- Identical sequences of letters are compared with regard to case, with lowercase before uppercase, so `a < A`

You can demonstrate the collation sequence for 7-bit ASCII characters like this:

```
require 'rubygems'
require 'restclient'
require 'json'

DB="http://adm:pass@127.0.0.1:5984/collator"

RestClient.delete DB rescue nil
RestClient.put "#{DB}",""

(32..126).each do |c|
    RestClient.put "#{DB}/#{c.to_s(16)}", {"x"=>c.chr}.to_json
end

RestClient.put "#{DB}/_design/test", <<EOS
{
    "views":{
        "one":{
            "map":"function (doc) { emit(doc.x,null); }"
        }
    }
}
EOS

puts RestClient.get("#{DB}/_design/test/_view/one")
```

This shows the collation sequence to be:

```
` ^ _ - , ; : ! ? . ' " ( ) [ ] { } @ * / \ & # % + < = > | ~ $ 0 1 2 3 4 5 6 7 8 9
a A b B c C d D e E f F g G h H i I j J k K l L m M n N o O p P q Q r R s S t T u U v␣
↪V w W x X y Y z Z
```

Key ranges

Take special care when querying key ranges. For example: the query:

```
startkey="Abc"&endkey="AbcZZZZ"
```

will match "ABC" and "abc1", but not "abc". This is because UCA sorts as:

```
abc < Abc < ABC < abc1 < AbcZZZZZ
```

For most applications, to avoid problems you should lowercase the *startkey*:

```
startkey="abc"&endkey="abcZZZZZZZZ"
```

will match all keys starting with `[aA][bB][cC]`

Complex keys

The query `startkey=["foo"]&endkey=["foo",{}]` will match most array keys with "foo" in the first element, such as `["foo","bar"]` and `["foo",["bar","baz"]]`. However it will not match `["foo",{"an":"object"}]`

_all_docs

The *_all_docs* view is a special case because it uses ASCII collation for doc ids, not UCA:

```
startkey="_design/"&endkey="_design/ZZZZZZZZ"
```

will not find `_design/abc` because *'Z'* comes before *'a'* in the ASCII sequence. A better solution is:

```
startkey="_design/"&endkey="_design0"
```

Raw collation

To squeeze a little more performance out of views, you can specify `"options":{"collation":"raw"}` within the view definition for native Erlang collation, especially if you don't require UCA. This gives a different collation sequence:

```
1
false
null
true
{"a":"a"},
["a"]
"a"
```

Beware that `{}` is no longer a suitable "high" key sentinel value. Use a string like `"\ufff0"` instead.

3.2.3 Joins With Views

Linked Documents

If your *map function* emits an object value which has `{'_id': XXX}` and you *query view* with `include_docs=true` parameter, then CouchDB will fetch the document with id `XXX` rather than the document which was processed to emit the key/value pair.

This means that if one document contains the ids of other documents, it can cause those documents to be fetched in the view too, adjacent to the same key if required.

For example, if you have the following hierarchically-linked documents:

```
[
    { "_id": "11111" },
    { "_id": "22222", "ancestors": ["11111"], "value": "hello" },
    { "_id": "33333", "ancestors": ["22222","11111"], "value": "world" }
]
```

You can emit the values with the ancestor documents adjacent to them in the view like this:

```
function(doc) {
    if (doc.value) {
        emit([doc.value, 0], null);
```

(continues on next page)

(continued from previous page)

```
        if (doc.ancestors) {
            for (var i in doc.ancestors) {
                emit([doc.value, Number(i)+1], {_id: doc.ancestors[i]});
            }
        }
    }
}
```

The result you get is:

```
{
    "total_rows": 5,
    "offset": 0,
    "rows": [
        {
            "id": "22222",
            "key": [
                "hello",
                0
            ],
            "value": null,
            "doc": {
                "_id": "22222",
                "_rev": "1-0eee81fecb5aa4f51e285c621271ff02",
                "ancestors": [
                    "11111"
                ],
                "value": "hello"
            }
        },
        {
            "id": "22222",
            "key": [
                "hello",
                1
            ],
            "value": {
                "_id": "11111"
            },
            "doc": {
                "_id": "11111",
                "_rev": "1-967a00dff5e02add41819138abb3284d"
            }
        },
        {
            "id": "33333",
            "key": [
                "world",
                0
            ],
            "value": null,
            "doc": {
                "_id": "33333",
                "_rev": "1-11e42b44fdb3d3784602eca7c0332a43",
                "ancestors": [
                    "22222",
```

(continues on next page)

(continued from previous page)

```
                    "11111"
                ],
                "value": "world"
            }
        },
        {
            "id": "33333",
            "key": [
                "world",
                1
            ],
            "value": {
                "_id": "22222"
            },
            "doc": {
                "_id": "22222",
                "_rev": "1-0eee81fecb5aa4f51e285c621271ff02",
                "ancestors": [
                    "11111"
                ],
                "value": "hello"
            }
        },
        {
            "id": "33333",
            "key": [
                "world",
                2
            ],
            "value": {
                "_id": "11111"
            },
            "doc": {
                "_id": "11111",
                "_rev": "1-967a00dff5e02add41819138abb3284d"
            }
        }
    ]
}
```

which makes it very cheap to fetch a document plus all its ancestors in one query.

Note that the `"id"` in the row is still that of the originating document. The only difference is that `include_docs` fetches a different doc.

The current revision of the document is resolved at query time, not at the time the view is generated. This means that if a new revision of the linked document is added later, it will appear in view queries even though the view itself hasn't changed. To force a specific revision of a linked document to be used, emit a `"_rev"` property as well as `"_id"`.

Using View Collation

Author
Christopher Lenz

Date
2007-10-05

Source
http://www.cmlenz.net/archives/2007/10/couchdb-joins

Just today, there was a discussion on IRC on how you'd go about modeling a simple blogging system with "post" and "comment" entities, where any blog post might have N comments. If you'd be using an SQL database, you'd obviously have two tables with foreign keys and you'd be using joins. (At least until you needed to add some denormalization).

But what would the "obvious" approach in CouchDB look like?

Approach #1: Comments Inlined

A simple approach would be to have one document per blog post, and store the comments inside that document:

```
{
    "_id": "myslug",
    "_rev": "123456",
    "author": "john",
    "title": "My blog post",
    "content": "Bla bla bla ...",
    "comments": [
        {"author": "jack", "content": "..."},
        {"author": "jane", "content": "..."}
    ]
}
```

Note: Of course the model of an actual blogging system would be more extensive, you'd have tags, timestamps, etc, etc. This is just to demonstrate the basics.

The obvious advantage of this approach is that the data that belongs together is stored in one place. Delete the post, and you automatically delete the corresponding comments, and so on.

You may be thinking that putting the comments inside the blog post document would not allow us to query for the comments themselves, but you'd be wrong. You could trivially write a CouchDB view that would return all comments across all blog posts, keyed by author:

```
function(doc) {
    for (var i in doc.comments) {
        emit(doc.comments[i].author, doc.comments[i].content);
    }
}
```

Now you could list all comments by a particular user by invoking the view and passing it a `?key="username"` query string parameter.

However, this approach has a drawback that can be quite significant for many applications: To add a comment to a post, you need to:

- Fetch the blog post document
- Add the new comment to the JSON structure

- Send the updated document to the server

Now if you have multiple client processes adding comments at roughly the same time, some of them will get a *HTTP 409 Conflict* error on step 3 (that's optimistic concurrency in action). For some applications this makes sense, but in many other apps, you'd want to append new related data regardless of whether other data has been added in the meantime.

The only way to allow non-conflicting addition of related data is by putting that related data into separate documents.

Approach #2: Comments Separate

Using this approach you'd have one document per blog post, and one document per comment. The comment documents would have a "backlink" to the post they belong to.

The blog post document would look similar to the above, minus the comments property. Also, we'd now have a type property on all our documents so that we can tell the difference between posts and comments:

```
{
    "_id": "myslug",
    "_rev": "123456",
    "type": "post",
    "author": "john",
    "title": "My blog post",
    "content": "Bla bla bla ..."
}
```

The comments themselves are stored in separate documents, which also have a type property (this time with the value "comment"), and additionally feature a post property containing the ID of the post document they belong to:

```
{
    "_id": "ABCDEF",
    "_rev": "123456",
    "type": "comment",
    "post": "myslug",
    "author": "jack",
    "content": "..."
}
```

```
{
    "_id": "DEFABC",
    "_rev": "123456",
    "type": "comment",
    "post": "myslug",
    "author": "jane",
    "content": "..."
}
```

To list all comments per blog post, you'd add a simple view, keyed by blog post ID:

```
function(doc) {
    if (doc.type == "comment") {
        emit(doc.post, {author: doc.author, content: doc.content});
    }
}
```

And you'd invoke that view passing it a `?key="post_id"` query string parameter.

Viewing all comments by author is just as easy as before:

```
function(doc) {
    if (doc.type == "comment") {
        emit(doc.author, {post: doc.post, content: doc.content});
    }
}
```

So this is better in some ways, but it also has a disadvantage. Imagine you want to display a blog post with all the associated comments on the same web page. With our first approach, we needed just a single request to the CouchDB server, namely a `GET` request to the document. With this second approach, we need two requests: a `GET` request to the post document, and a `GET` request to the view that returns all comments for the post.

That is okay, but not quite satisfactory. Just imagine you wanted to add threaded comments: you'd now need an additional fetch per comment. What we'd probably want then would be a way to join the blog post and the various comments together to be able to retrieve them with a single HTTP request.

This was when Damien Katz, the author of CouchDB, chimed in to the discussion on IRC to show us the way.

Optimization: Using the Power of View Collation

Obvious to Damien, but not at all obvious to the rest of us: it's fairly simple to make a view that includes both the content of the blog post document, and the content of all the comments associated with that post. The way you do that is by using *complex keys*. Until now we've been using simple string values for the view keys, but in fact they can be arbitrary JSON values, so let's make some use of that:

```
function(doc) {
    if (doc.type == "post") {
        emit([doc._id, 0], null);
    } else if (doc.type == "comment") {
        emit([doc.post, 1], null);
    }
}
```

Okay, this may be confusing at first. Let's take a step back and look at what views in CouchDB are really about.

CouchDB views are basically highly efficient on-disk dictionaries that map keys to values, where the key is automatically indexed and can be used to filter and/or sort the results you get back from your views. When you "invoke" a view, you can say that you're only interested in a subset of the view rows by specifying a `?key=foo` query string parameter. Or you can specify `?startkey=foo` and/or `?endkey=bar` query string parameters to fetch rows over a range of keys. Finally, by adding `?include_docs=true` to the query, the result will include the full body of each emitted document.

It's also important to note that keys are always used for collating (i.e. sorting) the rows. CouchDB has well defined (but as of yet undocumented) rules for comparing arbitrary JSON objects for collation. For example, the JSON value `["foo", 2]` is sorted after (considered "greater than") the values `["foo"]` or `["foo", 1, "bar"]`, but before e.g. `["foo", 2, "bar"]`. This feature enables a whole class of tricks that are rather non-obvious…

See also:

Views Collation

With that in mind, let's return to the view function above. First note that, unlike the previous view functions we've used here, this view handles both "post" and "comment" documents, and both of them end up as rows in the same view. Also, the key in this view is not just a simple string, but an array. The first element in that array is always the ID of the post, regardless of whether we're processing an actual post document, or a comment associated with a post. The second element is 0 for post documents, and 1 for comment documents.

Let's assume we have two blog posts in our database. Without limiting the view results via `key`, `startkey`, or `endkey`, we'd get back something like the following:

```
{
    "total_rows": 5, "offset": 0, "rows": [{
            "id": "myslug",
            "key": ["myslug", 0],
            "value": null
        }, {
            "id": "ABCDEF",
            "key": ["myslug", 1],
            "value": null
        }, {
            "id": "DEFABC",
            "key": ["myslug", 1],
            "value": null
        }, {
            "id": "other_slug",
            "key": ["other_slug", 0],
            "value": null
        }, {
            "id": "CDEFAB",
            "key": ["other_slug", 1],
            "value": null
        },
    ]
}
```

Note: The . . . placeholders here would contain the complete JSON encoding of the corresponding documents

Now, to get a specific blog post and all associated comments, we'd invoke that view with the query string:

```
?startkey=["myslug"]&endkey=["myslug", 2]&include_docs=true
```

We'd get back the first three rows, those that belong to the `myslug` post, but not the others, along with the full bodies of each document. Et voila, we now have the data we need to display a post with all associated comments, retrieved via a single `GET` request.

You may be asking what the 0 and 1 parts of the keys are for. They're simply to ensure that the post document is always sorted before the associated comment documents. So when you get back the results from this view for a specific post, you'll know that the first row contains the data for the blog post itself, and the remaining rows contain the comment data.

One remaining problem with this model is that comments are not ordered, but that's simply because we don't have date/time information associated with them. If we had, we'd add the timestamp as third element of the key array, probably as ISO date/time strings. Now we would continue using the query string `?startkey=["myslug"]&endkey=["myslug", 2]&include_docs=true` to fetch the blog post and all associated comments, only now they'd be in chronological order.

3.2.4 View Cookbook for SQL Jockeys

This is a collection of some common SQL queries and how to get the same result in CouchDB. The key to remember here is that CouchDB does not work like an SQL database at all, and that best practices from the SQL world do not translate well or at all to CouchDB. This document's "cookbook" assumes that you are familiar with the CouchDB basics such as creating and updating databases and documents.

Using Views

How you would do this in SQL:

```
CREATE TABLE
```

or:

```
ALTER TABLE
```

How you can do this in CouchDB?

Using views is a two-step process. First you define a view; then you query it. This is analogous to defining a table structure (with indexes) using `CREATE TABLE` or `ALTER TABLE` and querying it using an SQL query.

Defining a View

Defining a view is done by creating a special document in a CouchDB database. The only real specialness is the `_id` of the document, which starts with `_design/` — for example, _design/application. Other than that, it is just a regular CouchDB document. To make sure CouchDB understands that you are defining a view, you need to prepare the contents of that design document in a special format. Here is an example:

```
{
    "_id": "_design/application",
    "_rev": "1-C1687D17",
    "views": {
        "viewname": {
            "map": "function(doc) { ... }",
            "reduce": "function(keys, values) { ... }"
        }
    }
}
```

We are defining a view *viewname*. The definition of the view consists of two functions: the map function and the reduce function. Specifying a reduce function is optional. We'll look at the nature of the functions later. Note that *viewname* can be whatever you like: `users`, `by-name`, or `by-date` are just some examples.

A single design document can also include multiple view definitions, each identified by a unique name:

```
{
    "_id": "_design/application",
    "_rev": "1-C1687D17",
    "views": {
        "viewname": {
            "map": "function(doc) { ... }",
            "reduce": "function(keys, values) { ... }"
        },
        "anotherview": {
            "map": "function(doc) { ... }",
            "reduce": "function(keys, values) { ... }"
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

Querying a View

The name of the design document and the name of the view are significant for querying the view. To query the view *viewname*, you perform an HTTP `GET` request to the following URI:

```
/database/_design/application/_view/viewname
```

database is the name of the database you created your design document in. Next up is the design document name, and then the view name prefixed with `_view/`. To query *anotherview*, replace *viewname* in that URI with *anotherview*. If you want to query a view in a different design document, adjust the design document name.

MapReduce Functions

MapReduce is a concept that solves problems by applying a two-step process, aptly named the map phase and the reduce phase. The map phase looks at all documents in CouchDB separately one after the other and creates a *map result*. The map result is an ordered list of key/value pairs. Both key and value can be specified by the user writing the map function. A map function may call the built-in `emit(key, value)` function 0 to N times per document, creating a row in the map result per invocation.

CouchDB is smart enough to run a map function only once for every document, even on subsequent queries on a view. Only changes to documents or new documents need to be processed anew.

Map functions

Map functions run in isolation for every document. They can't modify the document, and they can't talk to the outside world—they can't have side effects. This is required so that CouchDB can guarantee correct results without having to recalculate a complete result when only one document gets changed.

The map result looks like this:

```
{"total_rows":3,"offset":0,"rows":[
{"id":"fc2636bf50556346f1ce46b4bc01fe30","key":"Lena","value":5},
{"id":"1fb2449f9b9d4e466dbfa47ebe675063","key":"Lisa","value":4},
{"id":"8ede09f6f6aeb35d948485624b28f149","key":"Sarah","value":6}
]}
```

It is a list of rows sorted by the value of key. The id is added automatically and refers back to the document that created this row. The value is the data you're looking for. For example purposes, it's the girl's age.

The map function that produces this result is:

```
function(doc) {
    if(doc.name && doc.age) {
        emit(doc.name, doc.age);
    }
}
```

It includes the if statement as a sanity check to ensure that we're operating on the right fields and calls the emit function with the name and age as the key and value.

Look Up by Key

How you would do this in SQL:

```
SELECT field FROM table WHERE value="searchterm"
```

How you can do this in CouchDB?

Use case: get a result (which can be a record or set of records) associated with a key ("searchterm").

To look something up quickly, regardless of the storage mechanism, an index is needed. An index is a data structure optimized for quick search and retrieval. CouchDB's map result is stored in such an index, which happens to be a B+ tree.

To look up a value by "searchterm", we need to put all values into the key of a view. All we need is a simple map function:

```
function(doc) {
    if(doc.value) {
        emit(doc.value, null);
    }
}
```

This creates a list of documents that have a value field sorted by the data in the value field. To find all the records that match "searchterm", we query the view and specify the search term as a query parameter:

```
/database/_design/application/_view/viewname?key="searchterm"
```

Consider the documents from the previous section, and say we're indexing on the age field of the documents to find all the five-year-olds:

```
function(doc) {
    if(doc.age && doc.name) {
        emit(doc.age, doc.name);
    }
}
```

Query:

```
/ladies/_design/ladies/_view/age?key=5
```

Result:

```
{"total_rows":3,"offset":1,"rows":[
{"id":"fc2636bf50556346f1ce46b4bc01fe30","key":5,"value":"Lena"}
]}
```

Easy.

Note that you have to emit a value. The view result includes the associated document ID in every row. We can use it to look up more data from the document itself. We can also use the `?include_docs=true` parameter to have CouchDB fetch the individual documents for us.

Look Up by Prefix

How you would do this in SQL:

```
SELECT field FROM table WHERE value LIKE "searchterm%"
```

How you can do this in CouchDB?

Use case: find all documents that have a field value that starts with *searchterm*. For example, say you stored a MIME type (like *text/html* or *image/jpg*) for each document and now you want to find all documents that are images according to the MIME type.

The solution is very similar to the previous example: all we need is a map function that is a little more clever than the first one. But first, an example document:

```
{
    "_id": "Hugh Laurie",
    "_rev": "1-9fded7deef52ac373119d05435581edf",
    "mime-type": "image/jpg",
    "description": "some dude"
}
```

The clue lies in extracting the prefix that we want to search for from our document and putting it into our view index. We use a regular expression to match our prefix:

```
function(doc) {
    if(doc["mime-type"]) {
        // from the start (^) match everything that is not a slash ([^\/]+) until
        // we find a slash (\/). Slashes needs to be escaped with a backslash (\/)
        var prefix = doc["mime-type"].match(/^[^\/]+\//);
        if(prefix) {
          emit(prefix, null);
        }
    }
}
```

We can now query this view with our desired MIME type prefix and not only find all images, but also text, video, and all other formats:

```
/files/_design/finder/_view/by-mime-type?key="image/"
```

Aggregate Functions

How you would do this in SQL:

```
SELECT COUNT(field) FROM table
```

How you can do this in CouchDB?

Use case: calculate a derived value from your data.

We haven't explained reduce functions yet. Reduce functions are similar to aggregate functions in SQL. They compute a value over multiple documents.

To explain the mechanics of reduce functions, we'll create one that doesn't make a whole lot of sense. But this example is easy to understand. We'll explore more useful reductions later.

Reduce functions operate on the output of the map function (also called the map result or intermediate result). The reduce function's job, unsurprisingly, is to reduce the list that the map function produces.

Here's what our summing reduce function looks like:

```
function(keys, values) {
    var sum = 0;
    for(var idx in values) {
        sum = sum + values[idx];
    }
    return sum;
}
```

Here's an alternate, more idiomatic JavaScript version:

```
function(keys, values) {
    var sum = 0;
    values.forEach(function(element) {
        sum = sum + element;
    });
    return sum;
}
```

Note: Don't miss effective built-in *reduce functions* like `_sum` and `_count`

This reduce function takes two arguments: a list of keys and a list of values. For our summing purposes we can ignore the keys-list and consider only the value list. We're looping over the list and add each item to a running total that we're returning at the end of the function.

You'll see one difference between the map and the reduce function. The map function uses `emit()` to create its result, whereas the reduce function returns a value.

For example, from a list of integer values that specify the age, calculate the sum of all years of life for the news headline, *"786 life years present at event."* A little contrived, but very simple and thus good for demonstration purposes. Consider the documents and the map view we used earlier in this document.

The reduce function to calculate the total age of all girls is:

```
function(keys, values) {
    return sum(values);
}
```

Note that, instead of the two earlier versions, we use CouchDB's predefined *`sum()`* function. It does the same thing as the other two, but it is such a common piece of code that CouchDB has it included.

The result for our reduce view now looks like this:

```
{"rows":[
    {"key":null,"value":15}
]}
```

The total sum of all age fields in all our documents is 15. Just what we wanted. The key member of the result object is null, as we can't know anymore which documents took part in the creation of the reduced result. We'll cover more advanced reduce cases later on.

As a rule of thumb, the reduce function should reduce to a single scalar value. That is, an integer; a string; or a small, fixed-size list or object that includes an aggregated value (or values) from the values argument. It should never just return values or similar. CouchDB will give you a warning if you try to use reduce "the wrong way":

```
{
    "error":"reduce_overflow_error",
    "message":"Reduce output must shrink more rapidly: Current output: ..."
}
```

Get Unique Values

How you would do this in SQL:

```
SELECT DISTINCT field FROM table
```

How you can do this in CouchDB?

Getting unique values is not as easy as adding a keyword. But a reduce view and a special query parameter give us the same result. Let's say you want a list of tags that your users have tagged themselves with and no duplicates.

First, let's look at the source documents. We punt on `_id` and `_rev` attributes here:

```
{
    "name":"Chris",
    "tags":["mustache", "music", "couchdb"]
}
```

```
{
    "name":"Noah",
    "tags":["hypertext", "philosophy", "couchdb"]
}
```

```
{
    "name":"Jan",
    "tags":["drums", "bike", "couchdb"]
}
```

Next, we need a list of all tags. A map function will do the trick:

```
function(doc) {
    if(doc.name && doc.tags) {
        doc.tags.forEach(function(tag) {
            emit(tag, null);
        });
    }
}
```

The result will look like this:

```
{"total_rows":9,"offset":0,"rows":[
{"id":"3525ab874bc4965fa3cda7c549e92d30","key":"bike","value":null},
{"id":"3525ab874bc4965fa3cda7c549e92d30","key":"couchdb","value":null},
{"id":"53f82b1f0ff49a08ac79a9dff41d7860","key":"couchdb","value":null},
{"id":"da5ea89448a4506925823f4d985aabbd","key":"couchdb","value":null},
{"id":"3525ab874bc4965fa3cda7c549e92d30","key":"drums","value":null},
{"id":"53f82b1f0ff49a08ac79a9dff41d7860","key":"hypertext","value":null},
{"id":"da5ea89448a4506925823f4d985aabbd","key":"music","value":null},
{"id":"da5ea89448a4506925823f4d985aabbd","key":"mustache","value":null},
{"id":"53f82b1f0ff49a08ac79a9dff41d7860","key":"philosophy","value":null}
]}
```

As promised, these are all the tags, including duplicates. Since each document gets run through the map function in isolation, it cannot know if the same key has been emitted already. At this stage, we need to live with that. To achieve uniqueness, we need a reduce:

```
function(keys, values) {
    return true;
}
```

This reduce doesn't do anything, but it allows us to specify a special query parameter when querying the view:

```
/dudes/_design/dude-data/_view/tags?group=true
```

CouchDB replies:

```
{"rows":[
{"key":"bike","value":true},
{"key":"couchdb","value":true},
{"key":"drums","value":true},
{"key":"hypertext","value":true},
{"key":"music","value":true},
{"key":"mustache","value":true},
{"key":"philosophy","value":true}
]}
```

In this case, we can ignore the value part because it is always true, but the result includes a list of all our tags and no duplicates!

With a small change we can put the reduce to good use, too. Let's see how many of the non-unique tags are there for each tag. To calculate the tag frequency, we just use the summing up we already learned about. In the map function, we emit a 1 instead of null:

```
function(doc) {
    if(doc.name && doc.tags) {
        doc.tags.forEach(function(tag) {
            emit(tag, 1);
        });
    }
}
```

In the reduce function, we return the sum of all values:

```
function(keys, values) {
    return sum(values);
}
```

Now, if we query the view with the `?group=true` parameter, we get back the count for each tag:

```
{"rows":[
{"key":"bike","value":1},
{"key":"couchdb","value":3},
{"key":"drums","value":1},
{"key":"hypertext","value":1},
{"key":"music","value":1},
{"key":"mustache","value":1},
{"key":"philosophy","value":1}
]}
```

Enforcing Uniqueness

How you would do this in SQL:

```
UNIQUE KEY(column)
```

How you can do this in CouchDB?

Use case: your applications require that a certain value exists only once in a database.

This is an easy one: within a CouchDB database, each document must have a unique `_id` field. If you require unique values in a database, just assign them to a document's `_id` field and CouchDB will enforce uniqueness for you.

There's one caveat, though: in the distributed case, when you are running more than one CouchDB node that accepts write requests, uniqueness can be guaranteed only per node or outside of CouchDB. CouchDB will allow two identical IDs to be written to two different nodes. On replication, CouchDB will detect a conflict and flag the document accordingly.

3.2.5 Pagination Recipe

This recipe explains how to paginate over view results. Pagination is a user interface (UI) pattern that allows the display of a large number of rows (*the result set*) without loading all the rows into the UI at once. A fixed-size subset, the *page*, is displayed along with next and previous links or buttons that can move the *viewport* over the result set to an adjacent page.

We assume you're familiar with creating and querying documents and views as well as the multiple view query options.

Example Data

To have some data to work with, we'll create a list of bands, one document per band:

```
{ "name":"Biffy Clyro" }

{ "name":"Foo Fighters" }

{ "name":"Tool" }

{ "name":"Nirvana" }

{ "name":"Helmet" }

{ "name":"Tenacious D" }

{ "name":"Future of the Left" }

{ "name":"A Perfect Circle" }

{ "name":"Silverchair" }

{ "name":"Queens of the Stone Age" }

{ "name":"Kerub" }
```

A View

We need a simple map function that gives us an alphabetical list of band names. This should be easy, but we're adding extra smarts to filter out "The" and "A" in front of band names to put them into the right position:

```
function(doc) {
    if(doc.name) {
        var name = doc.name.replace(/^(A|The) /, "");
        emit(name, null);
    }
}
```

The views result is an alphabetical list of band names. Now say we want to display band names five at a time and have a link pointing to the next five names that make up one page, and a link for the previous five, if we're not on the first page.

We learned how to use the `startkey`, `limit`, and `skip` parameters in earlier documents. We'll use these again here. First, let's have a look at the full result set:

```
{"total_rows":11,"offset":0,"rows":[
    {"id":"a0746072bba60a62b01209f467ca4fe2","key":"Biffy Clyro","value":null},
    {"id":"b47d82284969f10cd1b6ea460ad62d00","key":"Foo Fighters","value":null},
    {"id":"45ccde324611f86ad4932555dea7fce0","key":"Tenacious D","value":null},
    {"id":"d7ab24bb3489a9010c7d1a2087a4a9e4","key":"Future of the Left","value":null},
    {"id":"ad2f85ef87f5a9a65db5b3a75a03cd82","key":"Helmet","value":null},
    {"id":"a2f31cfa68118a6ae9d35444fcb1a3cf","key":"Nirvana","value":null},
    {"id":"67373171d0f626b811bdc34e92e77901","key":"Kerub","value":null},
    {"id":"3e1b84630c384f6aef1a5c50a81e4a34","key":"Perfect Circle","value":null},
    {"id":"84a371a7b8414237fad1b6aaf68cd16a","key":"Queens of the Stone Age","value
↪":null},
    {"id":"dcdaf08242a4be7da1a36e25f4f0b022","key":"Silverchair","value":null},
    {"id":"fd590d4ad53771db47b0406054f02243","key":"Tool","value":null}
]}
```

Setup

The mechanics of paging are very simple:

- Display first page
- If there are more rows to show, show next link
- Draw subsequent page
- If this is not the first page, show a previous link
- If there are more rows to show, show next link

Or in a pseudo-JavaScript snippet:

```
var result = new Result();
var page = result.getPage();

page.display();

if(result.hasPrev()) {
    page.display_link('prev');
}

if(result.hasNext()) {
```

(continues on next page)

(continued from previous page)

```
    page.display_link('next');
}
```

Paging

To get the first five rows from the view result, you use the `?limit=5` query parameter:

```
curl -X GET 'http://adm:pass@127.0.0.1:5984/artists/_design/artists/_view/by-name?
↪limit=5'
```

The result:

```
{"total_rows":11,"offset":0,"rows":[
    {"id":"a0746072bba60a62b01209f467ca4fe2","key":"Biffy Clyro","value":null},
    {"id":"b47d82284969f10cd1b6ea460ad62d00","key":"Foo Fighters","value":null},
    {"id":"45ccde324611f86ad4932555dea7fce0","key":"Tenacious D","value":null},
    {"id":"d7ab24bb3489a9010c7d1a2087a4a9e4","key":"Future of the Left","value":null},
    {"id":"ad2f85ef87f5a9a65db5b3a75a03cd82","key":"Helmet","value":null}
]}
```

By comparing the `total_rows` value to our `limit` value, we can determine if there are more pages to display. We also know by the *offset* member that we are on the first page. We can calculate the value for `skip=` to get the results for the next page:

```
var rows_per_page = 5;
var page = (offset / rows_per_page) + 1; // == 1
var skip = page * rows_per_page; // == 5 for the first page, 10 for the second ...
```

So we query CouchDB with:

```
curl -X GET 'http://adm:pass@127.0.0.1:5984/artists/_design/artists/_view/by-name?
↪limit=5&skip=5'
```

Note we have to use `'` (single quotes) to escape the `&` character that is special to the shell we execute curl in.

The result:

```
{"total_rows":11,"offset":5,"rows":[
    {"id":"a2f31cfa68118a6ae9d35444fcb1a3cf","key":"Nirvana","value":null},
    {"id":"67373171d0f626b811bdc34e92e77901","key":"Kerub","value":null},
    {"id":"3e1b84630c384f6aef1a5c50a81e4a34","key":"Perfect Circle","value":null},
    {"id":"84a371a7b8414237fad1b6aaf68cd16a","key":"Queens of the Stone Age",
    "value":null},
    {"id":"dcdaf08242a4be7da1a36e25f4f0b022","key":"Silverchair","value":null}
]}
```

Implementing the `hasPrev()` and `hasNext()` method is pretty straightforward:

```
function hasPrev()
{
    return page > 1;
}

function hasNext()
{
    var last_page = Math.floor(total_rows / rows_per_page) +
        (total_rows % rows_per_page);
```

(continues on next page)

(continued from previous page)

```
    return page != last_page;
}
```

Paging (Alternate Method)

The method described above performed poorly with large skip values until CouchDB 1.2. Additionally, some use cases may call for the following alternate method even with newer versions of CouchDB. One such case is when duplicate results should be prevented. Using skip alone it is possible for new documents to be inserted during pagination which could change the offset of the start of the subsequent page.

A correct solution is not much harder. Instead of slicing the result set into equally sized pages, we look at 10 rows at a time and use `startkey` to jump to the next 10 rows. We even use skip, but only with the value 1.

Here is how it works:

- Request *rows_per_page + 1* rows from the view
- Display *rows_per_page* rows, *store + 1* row as `next_startkey` and `next_startkey_docid`
- As page information, keep `startkey` and `next_startkey`
- Use the `next_*` values to create the next link, and use the others to create the previous link

The trick to finding the next page is pretty simple. Instead of requesting 10 rows for a page, you request 11 rows, but display only 10 and use the values in the 11th row as the `startkey` for the next page. Populating the link to the previous page is as simple as carrying the current `startkey` over to the next page. If there's no previous `startkey`, we are on the first page. We stop displaying the link to the next page if we get *rows_per_page* or less rows back. This is called linked list pagination, as we go from page to page, or list item to list item, instead of jumping directly to a pre-computed page. There is one caveat, though. Can you spot it?

CouchDB view keys do not have to be unique; you can have multiple index entries read. What if you have more index entries for a key than rows that should be on a page? `startkey` jumps to the first row, and you'd be screwed if CouchDB didn't have an additional parameter for you to use. All view keys with the same value are internally sorted by *docid*, that is, the ID of the document that created that view row. You can use the `startkey_docid` and `endkey_docid` parameters to get subsets of these rows. For pagination, we still don't need `endkey_docid`, but `startkey_docid` is very handy. In addition to `startkey` and `limit`, you also use `startkey_docid` for pagination if, and only if, the extra row you fetch to find the next page has the same key as the current `startkey`.

It is important to note that the `*_docid` parameters only work in addition to the `*key` parameters and are only useful to further narrow down the result set of a view for a single key. They do not work on their own (the one exception being the built-in *_all_docs view* that already sorts by document ID).

The advantage of this approach is that all the key operations can be performed on the super-fast B-tree index behind the view. Looking up a page doesn't include scanning through hundreds and thousands of rows unnecessarily.

Jump to Page

One drawback of the linked list style pagination is that you can't pre-compute the rows for a particular page from the page number and the rows per page. Jumping to a specific page doesn't really work. Our gut reaction, if that concern is raised, is, "Not even Google is doing that!" and we tend to get away with it. Google always pretends on the first page to find 10 more pages of results. Only if you click on the second page (something very few people actually do) might Google display a reduced set of pages. If you page through the results, you get links for the previous and next 10 pages, but no more. Pre-computing the necessary `startkey` and `startkey_docid` for 20 pages is a feasible operation and a pragmatic optimization to know the rows for every page in a result set that is potentially tens of thousands of rows long, or more.

If you really do need to jump to a page over the full range of documents (we have seen applications that require that), you can still maintain an integer value index as the view index and take a hybrid approach at solving pagination.

3.3 Search

Search indexes enable you to query a database by using the Lucene Query Parser Syntax. A search index uses one, or multiple, fields from your documents. You can use a search index to run queries, find documents based on the content they contain, or work with groups, facets, or geographical searches.

Warning: Search cannot function unless it has a functioning, cluster-connected Clouseau instance. See *Search Plugin Installation* for details.

To create a search index, you add a JavaScript function to a design document in the database. An index builds after processing one search request or after the server detects a document update. The `index` function takes the following parameters:

1. Field name - The name of the field you want to use when you query the index. If you set this parameter to `default`, then this field is queried if no field is specified in the query syntax.
2. Data that you want to index, for example, `doc.address.country`.
3. (Optional) The third parameter includes the following fields: `boost`, `facet`, `index`, and `store`. These fields are described in more detail later.

By default, a search index response returns 25 rows. The number of rows that is returned can be changed by using the `limit` parameter. Each response includes a `bookmark` field. You can include the value of the `bookmark` field in later queries to look through the responses.

Example design document that defines a search index:

```
{
    "_id": "_design/search_example",
    "indexes": {
        "animals": {
            "index": "function(doc){ ... }"
        }
    }
}
```

A search index will inherit the partitioning type from the `options.partitioned` field of the design document that contains it.

3.3.1 Index functions

Attempting to index by using a data field that does not exist fails. To avoid this problem, use the appropriate *guard clause*.

Note: Your indexing functions operate in a memory-constrained environment where the document itself forms a part of the memory that is used in that environment. Your code's stack and document must fit inside this memory. In other words, a document must be loaded in order to be indexed. Documents are limited to a maximum size of 64 MB.

Note: Within a search index, do not index the same field name with more than one data type. If the same field name is indexed with different data types in the same search index function, you might get an error when querying the search index that says the field "was indexed without position data." For example, do not include both of these lines in the same search index function, as they index the `myfield` field as two different data types: a string `"this is a string"` and a number `123`.

```
index("myfield", "this is a string");
index("myfield", 123);
```

The function that is contained in the index field is a JavaScript function that is called for each document in the database. The function takes the document as a parameter, extracts some data from it, and then calls the function that is defined in the `index` field to index that data.

The `index` function takes three parameters, where the third parameter is optional.

1. The first parameter is the name of the field you intend to use when querying the index, and which is specified in the Lucene syntax portion of subsequent queries. An example appears in the following query:

   ```
   query=color:red
   ```

   The Lucene field name `color` is the first parameter of the `index` function.

   The `query` parameter can be abbreviated to `q`, so another way of writing the query is as follows:

   ```
   q=color:red
   ```

   If the special value `"default"` is used when you define the name, you do not have to specify a field name at query time. The effect is that the query can be simplified:

   ```
   query=red
   ```

2. The second parameter is the data to be indexed. Keep the following information in mind when you index your data:

   - This data must be only a string, number, or boolean. Other types will cause an error to be thrown by the index function call.
   - If an error is thrown when running your function, for this reason or others, the document will not be added to that search index.

3. The third, optional, parameter is a JavaScript object with the following fields:

   Index function (optional parameter)

   - **boost** - A number that specifies the relevance in search results. Content that is indexed with a boost value greater than 1 is more relevant than content that is indexed without a boost value. Content with a boost value less than one is not so relevant. Value is a positive floating point number. Default is 1 (no boosting).
   - **facet** - Creates a faceted index. See *Faceting*. Values are `true` or `false`. Default is `false`.
   - **index** - Whether the data is indexed, and if so, how. If set to `false`, the data cannot be used for searches, but can still be retrieved from the index if `store` is set to `true`. See *Analyzers*. Values are `true` or `false`. Default is `true`
   - **store** - If `true`, the value is returned in the search result; otherwise, the value is not returned. Values are `true` or `false`. Default is `false`.

   Note: If you do not set the `store` parameter, the index data results for the document are not returned in response to a query.

Example search index function:

```
function(doc) {
    index("default", doc._id);
    if (doc.min_length) {
        index("min_length", doc.min_length, {"store": true});
    }
```

(continues on next page)

(continued from previous page)

```
    if (doc.diet) {
        index("diet", doc.diet, {"store": true});
    }
    if (doc.latin_name) {
        index("latin_name", doc.latin_name, {"store": true});
    }
    if (doc.class) {
        index("class", doc.class, {"store": true});
    }
}
```

Index guard clauses

The `index` function requires the name of the data field to index as the second parameter. However, if that data field does not exist for the document, an error occurs. The solution is to use an appropriate 'guard clause' that checks if the field exists, and contains the expected type of data, *before* any attempt to create the corresponding index.

Example of failing to check whether the index data field exists:

```
if (doc.min_length) {
    index("min_length", doc.min_length, {"store": true});
}
```

You might use the JavaScript `typeof` function to implement the guard clause test. If the field exists *and* has the expected type, the correct type name is returned, so the guard clause test succeeds and it is safe to use the index function. If the field does *not* exist, you would not get back the expected type of the field, therefore you would not attempt to index the field.

JavaScript considers a result to be false if one of the following values is tested:

- 'undefined'
- null
- The number +0
- The number -0
- NaN (not a number)
- "" (the empty string)

Using a guard clause to check whether the required data field exists, and holds a number, before an attempt to index:

```
if (typeof(doc.min_length) === 'number') {
    index("min_length", doc.min_length, {"store": true});
}
```

Use a generic guard clause test to ensure that the type of the candidate data field is defined.

Example of a 'generic' guard clause:

```
if (typeof(doc.min_length) !== 'undefined') {
    // The field exists, and does have a type, so we can proceed to index using it.
    ...
}
```

3.3.2 Analyzers

Analyzers are settings that define how to recognize terms within text. Analyzers can be helpful if you need to *index multiple languages*.

Here's the list of generic analyzers, and their descriptions, that are supported by search:

- `classic` - The standard Lucene analyzer, circa release 3.1.
- `email` - Like the `standard` analyzer, but tries harder to match an email address as a complete token.
- `keyword` - Input is not tokenized at all.
- `simple` - Divides text at non-letters.
- `standard` - The default analyzer. It implements the Word Break rules from the Unicode Text Segmentation algorithm
- `whitespace` - Divides text at white space boundaries.

Example analyzer document:

```
{
    "_id": "_design/analyzer_example",
    "indexes": {
        "INDEX_NAME": {
            "index": "function (doc) { ... }",
            "analyzer": "$ANALYZER_NAME"
        }
    }
}
```

Language-specific analyzers

These analyzers omit common words in the specific language, and many also remove prefixes and suffixes. The name of the language is also the name of the analyzer. See package org.apache.lucene.analysis for more information.

Language	Analyzer
`arabic`	org.apache.lucene.analysis.ar.ArabicAnalyzer
`armenian`	org.apache.lucene.analysis.hy.ArmenianAnalyzer
`basque`	org.apache.lucene.analysis.eu.BasqueAnalyzer
`bulgarian`	org.apache.lucene.analysis.bg.BulgarianAnalyzer
`brazilian`	org.apache.lucene.analysis.br.BrazilianAnalyzer
`catalan`	org.apache.lucene.analysis.ca.CatalanAnalyzer
`cjk`	org.apache.lucene.analysis.cjk.CJKAnalyzer
`chinese`	org.apache.lucene.analysis.cn.smart.SmartChineseAnalyzer
`czech`	org.apache.lucene.analysis.cz.CzechAnalyzer
`danish`	org.apache.lucene.analysis.da.DanishAnalyzer
`dutch`	org.apache.lucene.analysis.nl.DutchAnalyzer
`english`	org.apache.lucene.analysis.en.EnglishAnalyzer
`finnish`	org.apache.lucene.analysis.fi.FinnishAnalyzer
`french`	org.apache.lucene.analysis.fr.FrenchAnalyzer
`german`	org.apache.lucene.analysis.de.GermanAnalyzer
`greek`	org.apache.lucene.analysis.el.GreekAnalyzer
`galician`	org.apache.lucene.analysis.gl.GalicianAnalyzer
`hindi`	org.apache.lucene.analysis.hi.HindiAnalyzer
`hungarian`	org.apache.lucene.analysis.hu.HungarianAnalyzer
`indonesian`	org.apache.lucene.analysis.id.IndonesianAnalyzer

continues on next page

Table 1 – continued from previous page

Language	Analyzer
`irish`	org.apache.lucene.analysis.ga.IrishAnalyzer
`italian`	org.apache.lucene.analysis.it.ItalianAnalyzer
`japanese`	org.apache.lucene.analysis.ja.JapaneseAnalyzer
`japanese`	org.apache.lucene.analysis.ja.JapaneseTokenizer
`latvian`	org.apache.lucene.analysis.lv.LatvianAnalyzer
`norwegian`	org.apache.lucene.analysis.no.NorwegianAnalyzer
`persian`	org.apache.lucene.analysis.fa.PersianAnalyzer
`polish`	org.apache.lucene.analysis.pl.PolishAnalyzer
`portuguese`	org.apache.lucene.analysis.pt.PortugueseAnalyzer
`romanian`	org.apache.lucene.analysis.ro.RomanianAnalyzer
`russian`	org.apache.lucene.analysis.ru.RussianAnalyzer
`spanish`	org.apache.lucene.analysis.es.SpanishAnalyzer
`swedish`	org.apache.lucene.analysis.sv.SwedishAnalyzer
`thai`	org.apache.lucene.analysis.th.ThaiAnalyzer
`turkish`	org.apache.lucene.analysis.tr.TurkishAnalyzer

Note: The `japanese` analyzer, org.apache.lucene.analysis.ja.JapaneseTokenizer, includes DEFAULT_MODE and defaultStopTags.

Note: Language-specific analyzers are optimized for the specified language. You cannot combine a generic analyzer with a language-specific analyzer. Instead, you might use a *per field analyzer* to select different analyzers for different fields within the documents.

Per-field analyzers

The `perfield` analyzer configures multiple analyzers for different fields.

Example of defining different analyzers for different fields:

```
{
    "_id": "_design/analyzer_example",
    "indexes": {
        "INDEX_NAME": {
            "analyzer": {
                "name": "perfield",
                "default": "english",
                "fields": {
                    "spanish": "spanish",
                    "german": "german"
                }
            },
            "index": "function (doc) { ... }"
        }
    }
}
```

Stop words

Stop words are words that do not get indexed. You define them within a design document by turning the analyzer string into an object.

Note: The `keyword`, `simple`, and `whitespace` analyzers do not support stop words.

The default stop words for the `standard` analyzer are included below:

```
"a", "an", "and", "are", "as", "at", "be", "but", "by", "for", "if",
"in", "into", "is", "it", "no", "not", "of", "on", "or", "such",
"that", "the", "their", "then", "there", "these", "they", "this",
"to", "was", "will", "with"
```

Example of defining non-indexed ('stop') words:

```
{
    "_id": "_design/stop_words_example",
    "indexes": {
        "INDEX_NAME": {
            "analyzer": {
                "name": "portuguese",
                "stopwords": [
                    "foo",
                    "bar",
                    "baz"
                ]
            },
            "index": "function (doc) { ... }"
        }
    }
}
```

Testing analyzer tokenization

You can test the results of analyzer tokenization by posting sample data to the `_search_analyze` endpoint.

Example of using HTTP to test the keyword analyzer:

```
POST /_search_analyze HTTP/1.1
Content-Type: application/json
{"analyzer":"keyword", "text":"ablanks@renovations.com"}
```

Example of using the command line to test the keyword analyzer:

```
curl 'https://$HOST:5984/_search_analyze' -H 'Content-Type: application/json'
    -d '{"analyzer":"keyword", "text":"ablanks@renovations.com"}'
```

Result of testing the keyword analyzer:

```
{
    "tokens": [
        "ablanks@renovations.com"
    ]
}
```

Example of using HTTP to test the standard analyzer:

```
POST /_search_analyze HTTP/1.1
Content-Type: application/json
{"analyzer":"standard", "text":"ablanks@renovations.com"}
```

Example of using the command line to test the standard analyzer:

```
curl 'https://$HOST:5984/_search_analyze' -H 'Content-Type: application/json'
    -d '{"analyzer":"standard", "text":"ablanks@renovations.com"}'
```

Result of testing the standard analyzer:

```
{
    "tokens": [
        "ablanks",
        "renovations.com"
    ]
}
```

3.3.3 Queries

After you create a search index, you can query it.

- Issue a partition query using: `GET /$DATABASE/_partition/$PARTITION_KEY/_design/$DDOC/_search/$INDEX_NAME`
- Issue a global query using: `GET /$DATABASE/_design/$DDOC/_search/$INDEX_NAME`

Specify your search by using the `query` parameter.

Example of using HTTP to query a partitioned index:

```
GET /$DATABASE/_partition/$PARTITION_KEY/_design/$DDOC/_search/$INDEX_NAME?include_
↪docs=true&query="*:*"&limit=1 HTTP/1.1
Content-Type: application/json
```

Example of using HTTP to query a global index:

```
GET /$DATABASE/_design/$DDOC/_search/$INDEX_NAME?include_docs=true&query="*:*"&
↪limit=1 HTTP/1.1
Content-Type: application/json
```

Example of using the command line to query a partitioned index:

```
curl https://$HOST:5984/$DATABASE/_partition/$PARTITION_KEY/_design/$DDOC/
_search/$INDEX_NAME?include_docs=true\&query="*:*"\&limit=1 \
```

Example of using the command line to query a global index:

```
curl https://$HOST:5984/$DATABASE/_design/$DDOC/_search/$INDEX_NAME?
include_docs=true\&query="*:*"\&limit=1 \
```

Query Parameters

A full list of query parameters can be found in the *API Reference*.

You must enable *faceting* before you can use the following parameters:

- `counts`
- `drilldown`
- `ranges`

Note: Do not combine the `bookmark` and `stale` options. These options constrain the choice of shard replicas to use for the response. When used together, the options might cause problems when contact is attempted with replicas that are slow or not available.

Relevance

When more than one result might be returned, it is possible for them to be sorted. By default, the sorting order is determined by 'relevance'.

Relevance is measured according to Apache Lucene Scoring. As an example, if you search a simple database for the word `example`, two documents might contain the word. If one document mentions the word `example` 10 times, but the second document mentions it only twice, then the first document is considered to be more 'relevant'.

If you do not provide a `sort` parameter, relevance is used by default. The highest scoring matches are returned first.

If you provide a `sort` parameter, then matches are returned in that order, ignoring relevance.

If you want to use a `sort` parameter, and also include ordering by relevance in your search results, use the special fields `-<score>` or `<score>` within the `sort` parameter.

POSTing search queries

Instead of using the `GET` HTTP method, you can also use `POST`. The main advantage of `POST` queries is that they can have a request body, so you can specify the request as a JSON object. Each parameter in the query string of a `GET` request corresponds to a field in the JSON object in the request body.

Example of using HTTP to POST a search request:

```
POST /db/_design/ddoc/_search/searchname HTTP/1.1
Content-Type: application/json
```

Example of using the command line to POST a search request:

```
curl 'https://$HOST:5984/db/_design/ddoc/_search/searchname' -X POST -H 'Content-
↪Type: application/json' -d @search.json
```

Example JSON document that contains a search request:

```
{
    "q": "index:my query",
    "sort": "foo",
    "limit": 3
}
```

3.3.4 Query syntax

The CouchDB search query syntax is based on the Lucene syntax. Search queries take the form of `name:value` unless the name is omitted, in which case they use the default field, as demonstrated in the following examples:

Example search query expressions:

```
// Birds
class:bird
```

```
// Animals that begin with the letter "l"
l*
```

```
// Carnivorous birds
class:bird AND diet:carnivore
```

```
// Herbivores that start with letter "l"
l* AND diet:herbivore
```

```
// Medium-sized herbivores
min_length:[1 TO 3] AND diet:herbivore
```

```
// Herbivores that are 2m long or less
diet:herbivore AND min_length:[-Infinity TO 2]
```

```
// Mammals that are at least 1.5m long
class:mammal AND min_length:[1.5 TO Infinity]
```

```
// Find "Meles meles"
latin_name:"Meles meles"
```

```
// Mammals who are herbivore or carnivore
diet:(herbivore OR omnivore) AND class:mammal
```

```
// Return all results
*:*
```

Queries over multiple fields can be logically combined, and groups and fields can be further grouped. The available logical operators are case-sensitive and are `AND`, `+`, `OR`, `NOT` and `-`. Range queries can run over strings or numbers.

If you want a fuzzy search, you can run a query with `~` to find terms like the search term. For instance, `look~` finds the terms `book` and `took`.

Note: If the lower and upper bounds of a range query are both strings that contain only numeric digits, the bounds are treated as numbers not as strings. For example, if you search by using the query `mod_date:["20170101" TO "20171231"]`, the results include documents for which `mod_date` is between the numeric values 20170101 and 20171231, not between the strings "20170101" and "20171231".

You can alter the importance of a search term by adding `^` and a positive number. This alteration makes matches containing the term more or less relevant, proportional to the power of the boost value. The default value is 1, which means no increase or decrease in the strength of the match. A decimal value of 0 - 1 reduces importance. making the match strength weaker. A value greater than one increases importance, making the match strength stronger.

Wildcard searches are supported, for both single (`?`) and multiple (`*`) character searches. For example, `dat?` would match `date` and `data`, whereas `dat*` would match `date`, `data`, `database`, and `dates`. Wildcards must come after the search term.

Use `*:*` to return all results.

If the search query does *not* specify the `"group_field"` argument, the response contains a bookmark. If this bookmark is later provided as a URL parameter, the response skips the rows that were seen already, making it quick and easy to get the next set of results.

Note: The response never includes a bookmark if the `"group_field"` parameter is included in the search query. See *group_field parameter*.

Note: The `group_field`, `group_limit`, and `group_sort` options are only available when making global queries.

The following characters require escaping if you want to search on them:

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \ /
```

To escape one of these characters, use a preceding backslash character (\).

The response to a search query contains an `order` field for each of the results. The `order` field is an array where the first element is the field or fields that are specified in the `sort` parameter. See the *sort parameter*. If no `sort` parameter is included in the query, then the `order` field contains the Lucene relevance score. If you use the 'sort by distance' feature as described in *geographical searches*, then the first element is the distance from a point. The distance is measured by using either kilometers or miles.

Note: The second element in the order array can be ignored. It is used for troubleshooting purposes only.

Faceting

CouchDB Search also supports faceted searching, enabling discovery of aggregate information about matches quickly and easily. You can match all documents by using the special `?q=*:*` query syntax, and use the returned facets to refine your query. To indicate that a field must be indexed for faceted queries, set `{"facet": true}` in its options.

Example of search query, specifying that faceted search is enabled:

```
function(doc) {
    index("type", doc.type, {"facet": true});
    index("price", doc.price, {"facet": true});
}
```

To use facets, all the documents in the index must include all the fields that have faceting enabled. If your documents do not include all the fields, you receive a `bad_request` error with the following reason, "The `field_name` does not exist." If each document does not contain all the fields for facets, create separate indexes for each field. If you do not create separate indexes for each field, you must include only documents that contain all the fields. Verify that the fields exist in each document by using a single `if` statement.

Example if statement to verify that the required fields exist in each document:

```
if (typeof doc.town == "string" && typeof doc.name == "string") {
    index("town", doc.town, {facet: true});
    index("name", doc.name, {facet: true});
   }
```

Counts

Note: The `counts` option is only available when making global queries.

The `counts` facet syntax takes a list of fields, and returns the number of query results for each unique value of each named field.

Note: The `count` operation works only if the indexed values are strings. The indexed values cannot be mixed types. For example, if 100 strings are indexed, and one number, then the index cannot be used for `count` operations. You can check the type by using the `typeof` operator, and convert it by using the `parseInt`, `parseFloat`, or `.toString()` functions.

Example of a query using the counts facet syntax:

```
?q=*:*&counts=["type"]
```

Example response after using of the counts facet syntax:

```
{
    "total_rows":100000,
    "bookmark":"g...",
    "rows":[...],
    "counts":{
        "type":{
            "sofa": 10,
            "chair": 100,
            "lamp": 97
        }
    }
}
```

Drilldown

Note: The `drilldown` option is only available when making global queries.

You can restrict results to documents with a dimension equal to the specified label. Restrict the results by adding `drilldown=["dimension","label"]` to a search query. You can include multiple `drilldown` parameters to restrict results along multiple dimensions.

```
GET /things/_design/inventory/_search/fruits?q=*:*&drilldown=["state","old"]&
↪drilldown=["item","apple"]&include_docs=true HTTP/1.1
```

For better language interoperability, you can achieve the same by supplying a list of lists:

```
GET /things/_design/inventory/_search/fruits?q=*:*&drilldown=[["state","old"],["item",
↪"apple"]]&include_docs=true HTTP/1.1
```

You can also supply a list of lists for `drilldown` in bodies of POST requests.

Note that, multiple values for a single key in a `drilldown` means an `OR` relation between them and there is an `AND` relation between multiple keys.

Using a `drilldown` parameter is similar to using `key:value` in the `q` parameter, but the `drilldown` parameter returns values that the analyzer might skip.

For example, if the analyzer did not index a stop word like `"a"`, using `drilldown` returns it when you specify `drilldown=["key","a"]`.

Ranges

Note: The `ranges` option is only available when making global queries.

The `range` facet syntax reuses the standard Lucene syntax for ranges to return counts of results that fit into each specified category. Inclusive range queries are denoted by brackets (`[`, `]`). Exclusive range queries are denoted by curly brackets (`{`, `}`).

Note: The `range` operation works only if the indexed values are numbers. The indexed values cannot be mixed types. For example, if 100 strings are indexed, and one number, then the index cannot be used for `range` operations. You can check the type by using the `typeof` operator, and convert it by using the `parseInt`, `parseFloat`, or `.toString()` functions.

Example of a request that uses faceted search for matching ranges:

```
?q=*:*&ranges={"price":{"cheap":"[0 TO 100]","expensive":"{100 TO Infinity}"}}
```

Example results after a ranges check on a faceted search:

```
{
    "total_rows":100000,
    "bookmark":"g...",
    "rows":[...],
    "ranges": {
        "price": {
            "expensive": 278682,
            "cheap": 257023
        }
    }
}
```

3.3.5 Geographical searches

In addition to searching by the content of textual fields, you can also sort your results by their distance from a geographic coordinate using Lucene's built-in geospatial capabilities.

To sort your results in this way, you must index two numeric fields, representing the longitude and latitude.

Note: You can also sort your results by their distance from a geographic coordinate using Lucene's built-in geospatial capabilities.

You can then query by using the special `<distance...>` sort field, which takes five parameters:

- Longitude field name: The name of your longitude field (`mylon` in the example).
- Latitude field name: The name of your latitude field (`mylat` in the example).
- Longitude of origin: The longitude of the place you want to sort by distance from.
- Latitude of origin: The latitude of the place you want to sort by distance from.
- Units: The units to use: `km` for kilometers or `mi` for miles. The distance is returned in the order field.

You can combine sorting by distance with any other search query, such as range searches on the latitude and longitude, or queries that involve non-geographical information.

That way, you can search in a bounding box, and narrow down the search with extra criteria.

Example geographical data:

```
{
    "name":"Aberdeen, Scotland",
    "lat":57.15,
    "lon":-2.15,
    "type":"city"
}
```

Example of a design document that contains a search index for the geographic data:

```
function(doc) {
    if (doc.type && doc.type == 'city') {
        index('city', doc.name, {'store': true});
        index('lat', doc.lat, {'store': true});
        index('lon', doc.lon, {'store': true});
    }
}
```

An example of using HTTP for a query that sorts cities in the northern hemisphere by their distance to New York:

```
GET /examples/_design/cities-designdoc/_search/cities?q=lat:[0+TO+90]&sort="<distance,
↪lon,lat,-74.0059,40.7127,km>" HTTP/1.1
```

An example of using the command line for a query that sorts cities in the northern hemisphere by their distance to New York:

```
curl 'https://$HOST:5984/examples/_design/cities-designdoc/_search/cities?
↪q=lat:[0+TO+90]&sort="<distance,lon,lat,-74.0059,40.7127,km>"'
```

Example (abbreviated) response, containing a list of northern hemisphere cities sorted by distance to New York:

```
{
    "total_rows": 205,
    "bookmark": "g1A...XIU",
    "rows": [
        {
            "id": "city180",
            "order": [
                8.530665755719783,
                18
            ],
            "fields": {
                "city": "New York, N.Y.",
                "lat": 40.78333333333333,
                "lon": -73.96666666666667
            }
        },
        {
            "id": "city177",
            "order": [
                13.756343205985946,
                17
            ],
```

(continues on next page)

(continued from previous page)

```
            "fields": {
                "city": "Newark, N.J.",
                "lat": 40.733333333333334,
                "lon": -74.16666666666667
            }
        },
        {
            "id": "city178",
            "order": [
                113.53603438866077,
                26
            ],
            "fields": {
                "city": "New Haven, Conn.",
                "lat": 41.31666666666667,
                "lon": -72.91666666666667
            }
        }
    ]
}
```

3.3.6 Highlighting search terms

Sometimes it is useful to get the context in which a search term was mentioned so that you can display more emphasized results to a user.

To get more emphasized results, add the `highlight_fields` parameter to the search query. Specify the field names for which you would like excerpts, with the highlighted search term returned.

By default, the search term is placed in `<em>` tags to highlight it, but the highlight can be overridden by using the `highlights_pre_tag` and `highlights_post_tag` parameters.

The length of the fragments is 100 characters by default. A different length can be requested with the `highlights_size` parameter.

The `highlights_number` parameter controls the number of fragments that are returned, and defaults to 1.

In the response, a `highlights` field is added, with one subfield per field name.

For each field, you receive an array of fragments with the search term highlighted.

Note: For highlighting to work, store the field in the index by using the `store: true` option.

Example of using HTTP to search with highlighting enabled:

```
GET /movies/_design/searches/_search/movies?q=movie_name:Azazel&highlight_fields=[
↪"movie_name"]&highlight_pre_tag="**"&highlight_post_tag="**"&highlights_size=30&
↪highlights_number=2 HTTP/1.1
Authorization: ...
```

Example of using the command line to search with highlighting enabled:

```
curl "https://$HOST:5984/movies/_design/searches/_search/movies?q=movie_name:Azazel&
↪highlight_fields=\[\"movie_name\"\]&highlight_pre_tag=\"**\"&highlight_post_tag=\
↪"**\"&highlights_size=30&highlights_number=2
```

Example of highlighted search results:

```
{
    "highlights": {
        "movie_name": [
            " on the Azazel Orient Express",
            " Azazel manuals, you"
        ]
    }
}
```

3.4 Nouveau

Warning: Nouveau is an experimental feature. Future releases might change how the endpoints work and might invalidate existing indexes.

Nouveau indexes enable you to query a database by using the Lucene Query Parser Syntax. A nouveau index uses one, or multiple, fields from your documents. You can use a nouveau index to run queries to find documents based on the content they contain.

Warning: Nouveau cannot function unless it has a functioning Nouveau server. See *Nouveau Server Installation* for details.

To create a nouveau index, you add a JavaScript function to a design document in the database. An index builds after processing one search request or after the server detects a document update. The `index` function takes the following parameters:

1. Field type - The type of the field, can be `string`, `text`, `double` or `stored`.
2. Field name - The name of the field you want to use when you query the index. If you set this parameter to `default`, then this field is queried if no field is specified in the query syntax.
3. Data that you want to index, for example, `doc.address.country`.
4. (Optional) The third parameter includes the following field: `store`.

By default, a nouveau index response returns 25 rows. The number of hits that are returned can be changed by using the `limit` parameter. Each response includes a `bookmark` field. You can include the value of the `bookmark` field in subsequent queries to fetch results from deeper in the result set.

Example design document that defines a nouveau index:

```
{
    "_id": "_design/nouveau_example",
    "nouveau": {
        "animals": {
            "index": "function(doc){ ... }"
        }
    }
}
```

A nouveau index will inherit the partitioning type from the `options.partitioned` field of the design document that contains it.

3.4.1 Index functions

Attempting to index by using a data field that does not exist fails. To avoid this problem, use the appropriate *guard clause*.

Note: Your indexing functions operate in a memory-constrained environment where the document itself forms a part of the memory that is used in that environment. Your code's stack and document must fit inside this memory. In other words, a document must be loaded in order to be indexed. Documents are limited to a maximum size of 64 MB.

The function that is contained in the index field is a JavaScript function that is called for each document in the database. The function takes the document as a parameter, extracts some data from it, and then calls the function that is defined in the `index` field to index that data.

The `index` function takes four parameters, where the third parameter is optional.

1. The first parameter is the type of the field.
2. The second parameter is the name of the field you intend to use when querying the index, and which is specified in the Lucene syntax portion of subsequent queries. An example appears in the following query:

   ```
   q=color:red
   ```

   The Lucene field name `color` is the first parameter of the `index` function.

   If the special value `"default"` is used when you define the name, you do not have to specify a field name at query time. The effect is that the query can be simplified:

   ```
   q=red
   ```

3. The third parameter is the data to be indexed. Keep the following information in mind when you index your data:
   - This data must be only a string, number, or boolean. Other types will cause an error to be thrown by the index function call.
   - If an error is thrown when running your function, for this reason or others, the document will not be added to that search index.
4. The fourth, optional, parameter is a JavaScript object with the following fields:

   Index function (optional parameter)

   - **store** - If `true`, the value is returned in the search result; otherwise, the value is not returned. Values are `true` or `false`. Default is `false`.

   Note: If you do not set the `store` parameter, the index data results for the document are not returned in response to a query.

Example search index function:

```
function(doc) {
    if (typeof(doc.min_length) == 'number') {
        index("double", "min_length", doc.min_length, {"store": true});
    }
    if (typeof(doc.diet) == 'string') {
        index("string", "diet", doc.diet, {"store": true});
    }
    if (typeof(doc.latin_name) == 'string') {
        index("string", "latin_name", doc.latin_name, {"store": true});
```

(continues on next page)

(continued from previous page)

```
    }
    if (typeof(doc.class) == 'string') {
        index("string", "class", doc.class, {"store": true});
    }
}
```

Index guard clauses

Runtime errors in the index function cause the document not to be indexed at all. The most common runtime errors are described below;

Example of failing to check whether the indexed value exists:

> **Warning:** example of bad code

```
index("double", "min_length", doc.min_length, {"store": true});
```

For documents without a *min_length* value, this index call will pass `undefined` as the value. This will be rejected by nouveau's validation function and the document will not be indexed.

Example of failing to check whether the nested indexed value exists:

> **Warning:** example of bad code

```
if (doc.foo.bar) {
    index("string", "bar", doc.foo.bar, {"store": true});
}
```

This bad example fails in a different way if `doc.foo` doesn't exist; the evaluation of `doc.foo.bar` throws an exception.

```
if (doc.foo && typeof(doc.foo) == 'object' && typeof(doc.foo.bar == 'string')) {
    index("string", "bar", doc.foo.bar, {"store": true});
}
```

This example correctly checks that `doc.foo` is an object and its `bar` entry is a string.

Example of checking the index value exists but disallowing valid false values:

> **Warning:** example of bad code

```
if (doc.min_length) {
  index("double", "min_length", doc.min_length, {"store": true});
}
```

We correct the previous mistake so documents without min_length are indexed (assuming there are other index calls for values that *do* exist) but we've acccidentally prevented the indexing of the `min_length` field if the `doc.min_length` happens to be `0`.

```
if (typeof(doc.min_length == 'number')) {
  index("double", "min_length", doc.min_length, {"store": true});
}
```

This good example ensures we index any document where `min_length` is a number.

3.4.2 Analyzers

Analyzers convert textual input into `tokens` which can be searched on. Analyzers typically have different rules for how they break up input into tokens, they might convert all text to lower case, they might omit whole words (typically words so common they are unlikely to be useful for searching), they might omit parts of words (removing `ing` suffixes in English, for example):

We expose a large number of Lucene's analyzers. We invent one ourselves (`simple_asciifolding`);

- arabic
- armenian
- basque
- bulgarian
- catalan
- chinese
- cjk
- classic
- czech
- danish
- dutch
- email
- english
- finnish
- french
- galician
- german
- hindi
- hungarian
- indonesian
- irish
- italian
- japanese
- keyword
- latvian
- norwegian
- persian
- polish
- portugese
- romanian
- russian
- simple

- simple_asciifolding
- spanish
- standard
- swedish
- thai
- turkish
- whitespace

Example analyzer document:

```
{
    "_id": "_design/analyzer_example",
    "nouveau": {
        "INDEX_NAME": {
            "index": "function (doc) { ... }",
            "default_analyzer": "$ANALYZER_NAME"
        }
    }
}
```

Field analyzers

You may optionally specify a different analyzer for a specific field.

Example of defining different analyzers for different fields:

```
{
    "_id": "_design/analyzer_example",
    "nouveau": {
        "INDEX_NAME": {
            "default_analyzer": "english",
            "field_analyzers": {
                "spanish": "spanish",
                "german": "german"
            },
            "index": "function (doc) { ... }"
        }
    }
}
```

Testing analyzer tokenization

You can test the results of analyzer tokenization by posting sample data to the `_nouveau_analyze` endpoint.

Example of using HTTP to test the keyword analyzer:

```
POST /_nouveau_analyze HTTP/1.1
Content-Type: application/json
{"analyzer":"keyword", "text":"ablanks@renovations.com"}
```

Example of using the command line to test the keyword analyzer:

```
curl 'https://$HOST:5984/_nouveau_analyze' -H 'Content-Type: application/json'
    -d '{"analyzer":"keyword", "text":"ablanks@renovations.com"}'
```

Result of testing the keyword analyzer:

```
{
    "tokens": [
        "ablanks@renovations.com"
    ]
}
```

Example of using HTTP to test the standard analyzer:

```
POST /_nouveau_analyze HTTP/1.1
Content-Type: application/json
{"analyzer":"standard", "text":"ablanks@renovations.com"}
```

Example of using the command line to test the standard analyzer:

```
curl 'https://$HOST:5984/_nouveau_analyze' -H 'Content-Type: application/json'
    -d '{"analyzer":"standard", "text":"ablanks@renovations.com"}'
```

Result of testing the standard analyzer:

```
{
    "tokens": [
        "ablanks",
        "renovations.com"
    ]
}
```

3.4.3 Queries

After you create a search index, you can query it.

- Issue a partition query using: `GET /$DATABASE/_partition/$PARTITION_KEY/_design/$DDOC/_nouveau/$INDEX_NAME`
- Issue a global query using: `GET /$DATABASE/_design/$DDOC/_nouveau/$INDEX_NAME`

Specify your search by using the `q` parameter.

Example of using HTTP to query a partitioned index:

```
GET /$DATABASE/_partition/$PARTITION_KEY/_design/$DDOC/_nouveau/$INDEX_NAME?include_
↪docs=true&q=*:*&limit=1 HTTP/1.1
Content-Type: application/json
```

Example of using HTTP to query a global index:

```
GET /$DATABASE/_design/$DDOC/_nouveau/$INDEX_NAME?include_docs=true&q=*:*&limit=1␣
↪HTTP/1.1
Content-Type: application/json
```

Example of using the command line to query a partitioned index:

```
curl https://$HOST:5984/$DATABASE/_partition/$PARTITION_KEY/_design/$DDOC/
_nouveau/$INDEX_NAME?include_docs=true\&q=*:*\&limit=1 \
```

Example of using the command line to query a global index:

```
curl https://$HOST:5984/$DATABASE/_design/$DDOC/_nouveau/$INDEX_NAME?
include_docs=true\&q=*:*\&limit=1 \
```

Query Parameters

A full list of query parameters can be found in the *API Reference*.

Note: Do not combine the `bookmark` and `update` options. These options constrain the choice of shard replicas to use for the response. When used together, the options might cause problems when contact is attempted with replicas that are slow or not available.

Relevance

When more than one result might be returned, it is possible for them to be sorted. By default, the sorting order is determined by 'relevance'.

Relevance is measured according to Apache Lucene Scoring. As an example, if you search a simple database for the word `example`, two documents might contain the word. If one document mentions the word `example` 10 times, but the second document mentions it only twice, then the first document is considered to be more 'relevant'.

If you do not provide a `sort` parameter, relevance is used by default. The highest scoring matches are returned first.

If you provide a `sort` parameter, then matches are returned in that order, ignoring relevance.

If you want to use a `sort` parameter, and also include ordering by relevance in your search results, use the special fields `-<score>` or `<score>` within the `sort` parameter.

POSTing search queries

Instead of using the `GET` HTTP method, you can also use `POST`. The main advantage of `POST` queries is that they can have a request body, so you can specify the request as a JSON object. Each parameter in the query string of a `GET` request corresponds to a field in the JSON object in the request body.

Example of using HTTP to POST a search request:

```
POST /db/_design/ddoc/_nouveau/searchname HTTP/1.1
Content-Type: application/json
```

Example of using the command line to POST a search request:

```
curl 'https://$HOST:5984/db/_design/ddoc/_nouveau/searchname' -X POST -H 'Content-
↪Type: application/json' -d @search.json
```

Example JSON document that contains a search request:

```
{
    "q": "index:my query",
    "sort": "foo",
    "limit": 3
}
```

3.4.4 Query syntax

The CouchDB search query syntax is based on the Lucene syntax. Search queries take the form of `name:value` unless the name is omitted, in which case they use the default field, as demonstrated in the following examples:

Example search query expressions:

```
// Birds
class:bird
```

```
// Animals that begin with the letter "l"
l*
```

```
// Carnivorous birds
class:bird AND diet:carnivore
```

```
// Herbivores that start with letter "l"
l* AND diet:herbivore
```

```
// Medium-sized herbivores
min_length:[1 TO 3] AND diet:herbivore
```

```
// Herbivores that are 2m long or less
diet:herbivore AND min_length:[-Infinity TO 2]
```

```
// Mammals that are at least 1.5m long
class:mammal AND min_length:[1.5 TO Infinity]
```

```
// Find "Meles meles"
latin_name:"Meles meles"
```

```
// Mammals who are herbivore or carnivore
diet:(herbivore OR omnivore) AND class:mammal
```

```
// Return all results
*:*
```

Queries over multiple fields can be logically combined, and groups and fields can be further grouped. The available logical operators are case-sensitive and are `AND`, `+`, `OR`, `NOT` and `-`. Range queries can run over strings or numbers.

If you want a fuzzy search, you can run a query with `~` to find terms like the search term. For instance, `look~` finds the terms `book` and `took`.

Note: If the lower and upper bounds of a range query are both strings that contain only numeric digits, the bounds are treated as numbers not as strings. For example, if you search by using the query `mod_date:["20170101" TO "20171231"]`, the results include documents for which `mod_date` is between the numeric values 20170101 and 20171231, not between the strings "20170101" and "20171231".

You can alter the importance of a search term by adding `^` and a positive number. This alteration makes matches containing the term more or less relevant, proportional to the power of the boost value. The default value is 1, which means no increase or decrease in the strength of the match. A decimal value of 0 - 1 reduces importance. making the match strength weaker. A value greater than one increases importance, making the match strength stronger.

Wildcard searches are supported, for both single (`?`) and multiple (`*`) character searches. For example, `dat?` would match `date` and `data`, whereas `dat*` would match `date`, `data`, `database`, and `dates`. Wildcards must come after the search term.

Use `*:*` to return all results.

The following characters require escaping if you want to search on them:

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \ /
```

To escape one of these characters, use a preceding backslash character (\).

The response to a search query contains an `order` field for each of the results. The `order` field is an array where the first element is the field or fields that are specified in the `sort` parameter. See the *sort parameter*. If no `sort` parameter is included in the query, then the `order` field contains the Lucene relevance score.

Faceting

Nouveau Search also supports faceted searching, enabling discovery of aggregate information about matches quickly and easily. You can match all documents by using the special `?q=*:*` query syntax, and use the returned facets to refine your query.

Example of search query:

```
function(doc) {
    index("string", "type", doc.type);
    index("double", "price", doc.price);
}
```

To use facets, all the documents in the index must include all the fields that have faceting enabled. If your documents do not include all the fields, you receive a `bad_request` error with the following reason, "The `field_name` does not exist." If each document does not contain all the fields for facets, create separate indexes for each field. If you do not create separate indexes for each field, you must include only documents that contain all the fields. Verify that the fields exist in each document by using a single `if` statement.

Example if statement to verify that the required fields exist in each document:

```
if (typeof doc.town == "string" && typeof doc.name == "string") {
    index("string", "town", doc.town);
    index("string", "name", doc.name);
   }
```

Counts

Note: The `counts` option is only available when making global queries.

The `counts` facet syntax takes a list of fields, and returns the number of query results for each unique value of each named field.

Note: The `count` operation works only if the indexed values are strings. The indexed values cannot be mixed types. For example, if 100 strings are indexed, and one number, then the index cannot be used for `count` operations. You can check the type by using the `typeof` operator, and convert it by using the `parseInt`, `parseFloat`, or `.toString()` functions.

Example of a query using the counts facet syntax:

```
?q=*:*&counts=["type"]
```

Example response after using of the counts facet syntax:

```
{
    "total_rows":100000,
    "bookmark":"g...",
    "rows":[...],
    "counts":{
        "type":{
            "sofa": 10,
            "chair": 100,
            "lamp": 97
        }
    }
}
```

Ranges

Note: The `ranges` option is only available when making global queries.

The `range` facet syntax reuses the standard Lucene syntax for ranges to return counts of results that fit into each specified category. Inclusive range queries are denoted by brackets (`[`, `]`). Exclusive range queries are denoted by curly brackets (`{`, `}`).

Note: The `range` operation works only if the indexed values are numbers. The indexed values cannot be mixed types. For example, if 100 strings are indexed, and one number, then the index cannot be used for `range` operations. You can check the type by using the `typeof` operator, and convert it by using the `parseInt`, `parseFloat`, or `.toString()` functions.

Example of a request that uses faceted search for matching ranges:

```
?q=*:*&ranges={"price":{"cheap":"[0 TO 100]","expensive":"{100 TO Infinity}"}}
```

Example results after a ranges check on a faceted search:

```
{
    "total_rows":100000,
    "bookmark":"g...",
    "rows":[...],
    "ranges": {
        "price": {
            "expensive": 278682,
            "cheap": 257023
        }
    }
}
```

Note: Previously, the functionality provided by CouchDB's design documents, in combination with document attachments, was referred to as "CouchApps." The general principle was that entire web applications could be hosted in CouchDB, without need for an additional application server.

Use of CouchDB as a combined standalone database and application server is no longer recommended. There are significant limitations to a pure CouchDB web server application stack, including but not limited to: fully-fledged fine-grained security, robust templating and scaffolding, complete developer tooling, and most importantly, a thriving ecosystem of developers, modules and frameworks to choose from.

The developers of CouchDB believe that web developers should pick "the right tool for the right job". Use CouchDB as your database layer, in conjunction with any number of other server-side web application frameworks, such as

the entire Node.JS ecosystem, Python's Django and Flask, PHP's Drupal, Java's Apache Struts, and more.

CHAPTER

FOUR

BEST PRACTICES

In this chapter, we present some of the best ways to use Apache CouchDB. These usage patterns reflect many years of real-world use. We hope that these will jump-start your next project, or improve the performance of your current system.

4.1 Document Design Considerations

When designing your database, and your document structure, there are a number of best practices to take into consideration. Especially for people accustomed to relational databases, some of these techniques may be non-obvious.

4.1.1 Don't rely on CouchDB's auto-UUID generation

While CouchDB will generate a unique identifier for the `_id` field of any doc that you create, in most cases you are better off generating them yourself for a few reasons:

- If for any reason you miss the `200 OK` reply from CouchDB, and storing the document is attempted again, you would end up with the same document content stored under multiple `_id`s. This could easily happen with intermediary proxies and cache systems that may not inform developers that the failed transaction is being retried.
- `_id`s are the only unique enforced value within CouchDB so you might as well make use of this. CouchDB stores its documents in a B+ tree. Each additional or updated document is stored as a leaf node, and may require re-writing intermediary and parent nodes. You may be able to take advantage of sequencing your own ids more effectively than the automatically generated ids if you can arrange them to be sequential yourself.

4.1.2 Alternatives to auto-incrementing sequences

Because of replication, as well as the distributed nature of CouchDB, it is not practical to use auto-incrementing sequences with CouchDB. These are often used to ensure unique identifiers for each row in a database table. CouchDB generates unique ids on its own and you can specify your own as well, so you don't really need a sequence here. If you use a sequence for something else, you will be better off finding another way to express it in CouchDB in another way.

4.1.3 Pre-aggregating your data

If your intent for CouchDB is as a collect-and-report model, not a real-time view, you may not need to store a single document for every event you're recording. In this case, pre-aggregating your data may be a good idea. You probably don't need 1000 documents per second if all you are trying to do is to track summary statistics about those documents. This reduces the computational pressure on CouchDB's MapReduce engine(s), as well as reduces its storage requirements.

In this case, using an in-memory store to summarize your statistical information, then writing out to CouchDB every 10 seconds / 1 minute / whatever level of granularity you need would greatly reduce the number of documents you'll put in your database.

Later, you can then further decimate your data by walking the entire database and generating documents to be stored in a new database with a lower level of granularity (say, 1 document a day). You can then delete the older, more fine-grained database when you're done with it.

4.1.4 Designing an application to work with replication

Whilst CouchDB includes replication and a conflict-flagging mechanism, this is not the whole story for building an application which replicates in a way which users expect.

Here we consider a simple example of a bookmarks application. The idea is that a user can replicate their own bookmarks, work with them on another machine, and then synchronise their changes later.

Let's start with a very simple definition of bookmarks: an ordered, nestable mapping of name to URL. Internally the application might represent it like this:

```
[
  {"name":"Weather", "url":"http://www.bbc.co.uk/weather"},
  {"name":"News", "url":"http://news.bbc.co.uk/"},
  {"name":"Tech", "bookmarks": [
    {"name":"Register", "url":"http://www.theregister.co.uk/"},
    {"name":"CouchDB", "url":"http://couchdb.apache.org/"}
  ]}
]
```

It can then present the bookmarks menu and sub-menus by traversing this structure.

Now consider this scenario: the user has a set of bookmarks on her PC, and then replicates it to her laptop. On the laptop, she changes the News link to point to CNN, renames "Register" to "The Register", and adds a new link to slashdot just after it. On the desktop, her husband deletes the Weather link, and adds a new link to CNET in the Tech folder.

So after these changes, the laptop has:

```
[
  {"name":"Weather", "url":"http://www.bbc.co.uk/weather"},
  {"name":"News", "url":"http://www.cnn.com/"},
  {"name":"Tech", "bookmarks": [
    {"name":"The Register", "url":"http://www.theregister.co.uk/"},
    {"name":"Slashdot", "url":"http://www.slashdot.new/"},
    {"name":"CouchDB", "url":"http://couchdb.apache.org/"}
  ]}
]
```

and the PC has:

```
[
  {"name":"News", "url":"http://www.cnn.com/"},
  {"name":"Tech", "bookmarks": [
```

(continues on next page)

(continued from previous page)

```
    {"name":"Register", "url":"http://www.theregister.co.uk/"},
    {"name":"CouchDB", "url":"http://couchdb.apache.org/"},
    {"name":"CNET", "url":"http://news.cnet.com/"}
  ]}
]
```

Upon the next synchronisation, we want the expected merge to take place. That is: links which were changed, added or deleted on one side are also changed, added or deleted on the other side - with no human intervention required unless absolutely necessary.

We will also assume that both sides are doing a CouchDB "compact" operation periodically, and are disconnected for more than this time before they resynchronise.

All of the approaches below which allow automated merging of changes rely on having some sort of history, back to the point where the replicas diverged.

CouchDB does not provide a mechanism for this itself. It stores arbitrary numbers of old _ids for one document (trunk now has a mechanism for pruning the _id history), for the purposes of replication. However it will not keep the documents themselves through a compaction cycle, except where there are conflicting versions of a document.

Do not rely on the CouchDB revision history mechanism to help you build an application-level version history. Its sole purpose is to ensure eventually consistent replication between databases. It is up to you to maintain history explicitly in whatever form makes sense for your application, and to prune it to avoid excessive storage utilisation, whilst not pruning past the point where live replicas last diverged.

Approach 1: Single JSON doc

The above structure is already valid JSON, and so could be represented in CouchDB just by wrapping it in an object and storing as a single document:

```
{
  "bookmarks":
  // ... same as above
}
```

This makes life very easy for the application, as the ordering and nesting is all taken care of. The trouble here is that on replication, only two sets of bookmarks will be visible: example B and example C. One will be chosen as the main revision, and the other will be stored as a conflicting revision.

At this point, the semantics are very unsatisfactory from the user's point of view. The best that can be offered is a choice saying "Which of these two sets of bookmarks do you wish to keep: B or C?" However neither represents the desired outcome. There is also insufficient data to be able to correctly merge them, since the base revision A is lost.

This is going to be highly unsatisfactory for the user, who will have to apply one set of changes again manually.

Approach 2: Separate document per bookmark

An alternative solution is to make each field (bookmark) a separate document in its own right. Adding or deleting a bookmark is then just a case of adding or deleting a document, which will never conflict (although if the same bookmark is added on both sides, then you will end up with two copies of it). Changing a bookmark will only conflict if both sides made changes to the same one, and then it is reasonable to ask the user to choose between them.

Since there will now be lots of small documents, you may either wish to keep a completely separate database for bookmarks, or else add an attribute to distinguish bookmarks from other kinds of document in the database. In the latter case, a view can be made to return only bookmark documents.

Whilst replication is now fixed, care is needed with the "ordered" and "nestable" properties of bookmarks.

For ordering, one suggestion is to give each item a floating-point index, and then when inserting an object between A and B, give it an index which is the average of A and B's indices. Unfortunately, this will fail after a while when you run out of precision, and the user will be bemused to find that their most recent bookmarks no longer remember the exact position they were put in.

A better way is to keep a string representation of index, which can grow as the tree is subdivided. This will not suffer the above problem, but it may result in this string becoming arbitrarily long after time. They could be renumbered, but the renumbering operation could introduce a lot of conflicts, especially if attempted by both sides independently.

For "nestable", you can have a separate doc which represents a list of bookmarks, and each bookmark can have a "belongs to" field which identifies the list. It may be useful anyway to be able to have multiple top-level bookmark sets (Bob's bookmarks, Jill's bookmarks etc). Some care is needed when deleting a list or sub-list, to ensure that all associated bookmarks are also deleted, otherwise they will become orphaned.

Building the entire bookmark set can be performed through the use of emitting a compound key that describes the path to the document, then using group levels to retrieve the position of the tree in the document. The following code excerpt describes a tree of files, where the path to the file is stored in the document under the `"path"` key:

```
// map function
function(doc) {
  if (doc.type === "file") {
    if (doc.path.substr(-1) === "/") {
      var raw_path = doc.path.slice(0, -1);
    } else {
      var raw_path = doc.path;
    }
    emit (raw_path.split('/'), 1);
  }
}

// reduce
_sum
```

This will emit rows into the view of the form `["opt", "couchdb", "etc", "local.ini"]` for a `doc.path` of `/opt/couchdb/etc/local.ini`. You can then query a list of files in the `/opt/couchdb/etc` directory by specifying a `startkey` of `["opt", "couchdb", "etc"]` and an `endkey` of `["opt", "couchdb", "etc", {}]`.

Approach 3: Immutable history / event sourcing

Another approach to consider is Event Sourcing or Command Logging, as implemented in many NoSQL databases and as used in many operational transformation systems.

In this model, instead of storing individual bookmarks, you store records of changes made - "Bookmark added", "Bookmark changed", "Bookmark moved", "Bookmark deleted". These are stored in an append-only fashion. Since records are never modified or deleted, only added to, there are never any replication conflicts.

These records can also be stored as an array in a single CouchDB document. Replication can cause a conflict, but in this case it is easy to resolve by simply combining elements from the two arrays.

In order to see the full set of bookmarks, you need to start with a baseline set (initially empty) and run all the change records since the baseline was created; and/or you need to maintain a most-recent version and update it with changes not yet seen.

Care is needed after replication when merging together history from multiple sources. You may get different results depending on how you order them - consider taking all A's changes before B's, taking all B's before A's, or interleaving them (e.g. if each change has a timestamp).

Also, over time the amount of storage used can grow arbitrarily large, even if the set of bookmarks itself is small. This can be controlled by moving the baseline version forwards and then keeping only the changes after that point.

However, care is needed not to move the baseline version forward so far that there are active replicas out there which last synchronised before that time, as this may result in conflicts which cannot be resolved automatically.

If there is any uncertainty, it is best to present the user with a prompt to assist with merging the content in the application itself.

Approach 4: Keep historic versions explicitly

If you are going to keep a command log history, then it may be simpler just to keep old revisions of the bookmarks list itself around. The intention is to subvert CouchDB's automatic behaviour of purging old revisions, by keeping these revisions as separate documents.

You can keep a pointer to the 'most current' revision, and each revision can point to its predecessor. On replication, merging can take place by diffing each of the previous versions (in effect synthesising the command logs) back to a common ancestor.

This is the sort of behaviour which revision control systems such as Git implement as a matter of routine, although generally comparing text files line-by-line rather than comparing JSON objects field-by-field.

Systems like Git will accumulate arbitrarily large amounts of history (although they will attempt to compress it by packing multiple revisions so that only their diffs are stored). With Git you can use "history rewriting" to remove old history, but this may prohibit merging if history doesn't go back far enough in time.

4.1.5 Adding client-side security with a translucent database

Many applications do not require a thick layer of security at the server. It is possible to use a modest amount of encryption and one-way functions to obscure the sensitive columns or key-value pairs, a technique often called a translucent database. (See a description.)

The simplest solutions use a one-way function like SHA-256 at the client to scramble the name and password before storing the information. This solution gives the client control of the data in the database without requiring a thick layer on the database to test each transaction. Some advantages are:

- Only the client or someone with the knowledge of the name and password can compute the value of SHA256 and recover the data.
- Some columns are still left in the clear, an advantage for computing aggregated statistics.
- Computation of SHA256 is left to the client side computer which usually has cycles to spare.
- The system prevents server-side snooping by insiders and any attacker who might penetrate the OS or any of the tools running upon it.

There are limitations:

- There is no root password. If the person forgets their name and password, their access is gone forever. This limits its use to databases that can continue by issuing a new user name and password.

There are many variations on this theme detailed in the book Translucent Databases, including:

- Adding a backdoor with public-key cryptography.
- Adding a second layer with steganography.
- Dealing with typographical errors.
- Mixing encryption with one-way functions.

4.2 Document submission using HTML Forms

It is possible to write to a CouchDB document directly from an HTML form by using a document *update function*. Here's how:

4.2.1 The HTML form

First, write an HTML form. Here's a simple "Contact Us" form excerpt:

```
<form action="/dbname/_design/ddocname/_update/contactform" method="post">
    <div>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" />
    </div>
    <div>
        <label for="mail">Email:</label>
        <input type="text" id="mail" name="email" />
    </div>
    <div>
        <label for="msg">Message:</label>
        <textarea id="msg" name="message"></textarea>
    </div>
</form>
```

Customize the `/dbname/_design/ddocname/_update/contactform` portion of the form action URL to reflect the exact path to your database, design document and update function (see below).

As CouchDB *no longer recommends the use of CouchDB-hosted web applications* , you may want to use a reverse proxy to expose CouchDB as a subdirectory of your web application. If so, add that prefix to the `action` destination in the form.

Another option is to alter CouchDB's *CORS* settings and use a cross-domain POST. *Be sure you understand all security implications before doing this!*

4.2.2 The update function

Then, write an update function. This is the server-side JavaScript function that will receive the `POST`-ed data.

The first argument to the function will be the document that is being processed (if it exists). Because we are using `POST` and not `PUT`, this should be empty in our scenario - but we should check to be sure. The `POST`-ed data will be passed as the second parameter to the function, along with any query parameters and the full request headers.

Here's a sample handler that extracts the form data, generates a document _id based on the email address and timestamp, and saves the document. It then returns a JSON success response back to the browser.

```
function(doc, req) {

    if (doc) {
        return [doc, toJSON({"error": "request already filed"})]
    }

    if !(req.form && req.form.email) {
        return [null, toJSON({"error": "incomplete form"})]
    }

    var date = new Date()
    var newdoc = req.form
```

(continues on next page)

(continued from previous page)

```
    newdoc._id = req.form.email + "_" + date.toISOString()

    return [newdoc, toJSON({"success":"ok"})]
}
```

Place the above function in your design document under the `updates` key.

Note that this function does not attempt any sort of input validation or sanitization. That is best handled by a *validate document update function* instead. (A "VDU" will validate any document written to the database, not just those that use your update function.)

If the first element passed to `return` is a document, the HTTP response headers will include `X-Couch-Id`, the `_id` value for the newly created document, and `X-Couch-Update-NewRev`, the `_rev` value for the newly created document. This is handy if your client-side code wants to access or update the document in a future call.

4.2.3 Example output

Here's the worked sample above, using `curl` to simulate the form POST.

```
$ curl -X PUT adm:pass@localhost:5984/testdb/_design/myddoc -d '{ "updates": {
↪"contactform": "function(doc, req) { ... }" } }'
{"ok":true,"id":"_design/myddoc","rev":"1-2a2b0951fcaf7287817573b03bba02ed"}

$ curl --data "name=Lin&email=lin@example.com&message=I Love CouchDB" http://
↪adm:pass@localhost:5984/testdb/_design/myddoc/_update/contactform
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 5984 (#1)
> POST /testdb/_design/myddoc/_update/contactform HTTP/1.1
> Host: localhost:5984
> User-Agent: curl/7.59.0
> Accept: */*
> Content-Length: 53
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 53 out of 53 bytes
< HTTP/1.1 201 Created
< Content-Length: 16
< Content-Type: text/html; charset=utf-8
< Date: Thu, 05 Apr 2018 19:56:42 GMT
< Server: CouchDB/2.2.0-948a1311c (Erlang OTP/19)
< X-Couch-Id: lin%40example.com_2018-04-05T19:51:22.278Z
< X-Couch-Request-ID: 03a5f4fbe0
< X-Couch-Update-NewRev: 1-34483732407fcc6cfc5b60ace48b9da9
< X-CouchDB-Body-Time: 0
<
* Connection #1 to host localhost left intact
{"success":"ok"}

$ curl http://adm:pass@localhost:5984/testdb/lin\@example.com_2018-04-05T19:51:22.278Z
{"_id":"lin@example.com_2018-04-05T19:51:22.278Z","_rev":"1-
↪34483732407fcc6cfc5b60ace48b9da9","name":"Lin","email":"lin@example.com","message":
↪"I Love CouchDB"}
```

4.3 Using an ISO Formatted Date for Document IDs

The ISO 8601 date standard describes a useful scheme for representing a date string in a Year-Month-DayTHour:Minute:Second.microsecond format. For time-bound documents in a CouchDB database this can be a very handy way to create a unique identifier, since JavaScript can directly use it to create a Date object. Using this sample `map` function:

```
function(doc) {
  var dt = new Date(doc._id);
  emit([dt.getDate(), doc.widget], 1);
}
```

simply use `group_level` to zoom in on whatever time you wish to use.

```
curl -X GET "http://adm:pass@localhost:5984/transactions/_design/widget_count/_view/
↪toss?group_level=1"

{"rows":[
{"key":[20],"value":10},
{"key":[21],"value":20}
]}

curl -X GET "http://adm:pass@localhost:5984/transactions/_design/widget_count/_view/
↪toss?group_level=2"

{"rows":[
{"key":[20,widget],"value":10},
{"key":[21,widget],"value":10},
{"key":[21,thing],"value":10}
]}
```

Another method is using `parseint()` and `datetime.substr()` to cut out useful values for a return key:

```
function (doc) {
  var datetime = doc._id;
  var year = parseInt(datetime.substr(0, 4));
  var month = parseInt(datetime.substr(5, 2), 10);
  var day = parseInt(datetime.substr(8, 2), 10);
  var hour = parseInt(datetime.substr(11, 2), 10);
  var minute = parseInt(datetime.substr(14, 2), 10);
  emit([doc.widget, year, month, day, hour, minute], 1);
}
```

4.4 JavaScript development tips

Working with Apache CouchDB's JavaScript environment is a lot different than working with traditional JavaScript development environments. Here are some tips and tricks that will ease the difficulty.

- Check the JavaScript version being used by your CouchDB. As of version 3.2.0, this is reported in the output of `GET /_node/_local/_versions`. Prior to version 3.2.0, you will need to see which JavaScript library is installed by your CouchDB binary distribution, provided by your operating system, or linked by your compilation process.

  If the version is 1.8.5, this is an **old** version of JavaScript, only supporting the ECMA-262 5th edition ("ES5") of the language. ES6/2015 and newer constructs **cannot** be used.

  Fortunately, there are many tools available for transpiling modern JavaScript into code compatible with older JS engines. The Babel Project website, for example, offers an in-browser text editor which transpiles

JavaScript in real-time. Configuring CouchDB-compatibility is as easy as enabling the `ENV PRESET` option, and typing "firefox 4.0" into the *TARGETS* field.

- The `log()` function will log output to the CouchDB log file or stream. You can log strings, objects, and arrays directly, without first converting to JSON. Use this in conjunction with a local CouchDB instance for best results.
- Be sure to guard all document accesses to avoid exceptions when fields or subfields are missing: `if (doc && doc.myarray && doc.myarray.length)...`

4.5 View recommendations

Here are some tips and tricks for working with CouchDB's (JavaScript-based) views.

4.5.1 Deploying a view change in a live environment

It is possible to change the definition of a view, build the index, then make those changes go live without causing downtime for your application. The trick to making this work is that CouchDB's JavaScript view index files are based on the contents of the design document - not its name, `_id` or revision. This means that two design documents with identical view code will share the same on-disk view index files.

Here is a worked example, assuming your `/db/_design/ddoc` needs to be updated.

1. Upload the old design doc to `/db/_design/ddoc-old` (or copy the document) if you want an easy way to rollback in case of problems. The `ddoc-old` document will reference the same view indexes already built for `_design/ddoc`.
2. Upload the updated design doc to `/db/_design/ddoc-new`.
3. Query a view in the new design document to trigger secondary index generation. You can track the indexing progress via the `/_active_tasks` endpoint, or through the *Fauxton* web interface.
4. When the index is done being built, re-upload the updated design document to `/db/_design/ddoc` (or copy the document). The `ddoc` document will now reference the same view indexes already built for `_design/ddoc-new`.
5. Delete `/db/_design/ddoc-new` and/or `/db/_design/ddoc-old` at your discretion. Don't forget to trigger *Views cleanup* to reclaim disk space after deleting `ddoc-old`.

The *COPY* HTTP verb can be used to copy the design document with a single command:

```
curl -X COPY <URL of source design document> -H "Destination: <ID of destination␣
↪design document>"
```

4.6 Reverse Proxies

4.6.1 Reverse proxying with HAProxy

CouchDB recommends the use of HAProxy as a load balancer and reverse proxy. The team's experience with using it in production has shown it to be superior for configuration and monitoring capabilities, as well as overall performance.

CouchDB's sample haproxy configuration is present in the code repository and release tarball as `rel/haproxy.cfg`. It is included below. This example is for a 3 node CouchDB cluster:

```
global
    maxconn 512
    spread-checks 5

defaults
    mode http
    log global
    monitor-uri /_haproxy_health_check
    option log-health-checks
    option httplog
    balance roundrobin
    option forwardfor
    option redispatch
    retries 4
    option http-server-close
    timeout client 150000
    timeout server 3600000
    timeout connect 500

    stats enable
    stats uri /_haproxy_stats
    # stats auth admin:admin # Uncomment for basic auth

frontend http-in
     # This requires HAProxy 1.5.x
     # bind *:$HAPROXY_PORT
     bind *:5984
     default_backend couchdbs

backend couchdbs
    option httpchk GET /_up
    http-check disable-on-404
    server couchdb1 x.x.x.x:5984 check inter 5s
    server couchdb2 x.x.x.x:5984 check inter 5s
    server couchdb3 x.x.x.x:5984 check inter 5s
```

4.6.2 Reverse proxying with nginx

Basic Configuration

Here's a basic excerpt from an nginx config file in `<nginx config directory>/sites-available/default`. This will proxy all requests from `http://domain.com/...` to `http://localhost:5984/...`

```
location / {
    proxy_pass http://localhost:5984;
    proxy_redirect off;
    proxy_buffering off;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

Proxy buffering **must** be disabled, or continuous replication will not function correctly behind nginx.

Reverse proxying CouchDB in a subdirectory with nginx

It can be useful to provide CouchDB as a subdirectory of your overall domain, especially to avoid CORS concerns. Here's an excerpt of a basic nginx configuration that proxies the URL `http://domain.com/couchdb` to `http://localhost:5984` so that requests appended to the subdirectory, such as `http://domain.com/couchdb/db1/doc1` are proxied to `http://localhost:5984/db1/doc1`.

```
location /couchdb {
    rewrite ^ $request_uri;
    rewrite ^/couchdb/(.*) /$1 break;
    proxy_pass http://localhost:5984$uri;
    proxy_redirect off;
    proxy_buffering off;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

Session based replication is default functionality since CouchDB 2.3.0. To enable session based replication with reverse proxied CouchDB in a subdirectory.

```
location /_session {
    proxy_pass http://localhost:5984/_session;
    proxy_redirect off;
    proxy_buffering off;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

Authentication with nginx as a reverse proxy

Here's a sample config setting with basic authentication enabled, placing CouchDB in the `/couchdb` subdirectory:

```
location /couchdb {
    auth_basic "Restricted";
    auth_basic_user_file htpasswd;
    rewrite /couchdb/(.*) /$1 break;
    proxy_pass http://localhost:5984;
    proxy_redirect off;
    proxy_buffering off;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Authorization "";
}
```

This setup leans entirely on nginx performing authorization, and forwarding requests to CouchDB with no authentication (with CouchDB in Admin Party mode), which isn't sufficient in CouchDB 3.0 anymore as Admin Party has been removed. You'd need to at the very least hard-code user credentials into this version with headers.

For a better solution, see *Proxy Authentication*.

SSL with nginx

In order to enable SSL, just enable the nginx SSL module, and add another proxy header:

```
ssl on;
ssl_certificate PATH_TO_YOUR_PUBLIC_KEY.pem;
ssl_certificate_key PATH_TO_YOUR_PRIVATE_KEY.key;
ssl_protocols SSLv3;
ssl_session_cache shared:SSL:1m;

location / {
    proxy_pass http://localhost:5984;
    proxy_redirect off;
    proxy_set_header Host $host;
    proxy_buffering off;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Ssl on;
}
```

The `X-Forwarded-Ssl` header tells CouchDB that it should use the `https` scheme instead of the `http` scheme. Otherwise, all CouchDB-generated redirects will fail.

4.6.3 Reverse Proxying with Caddy 2

Caddy is `https-by-default`, and will automatically acquire, install, activate and, when necessary, renew a trusted SSL certificate for you - all in the background. Certificates are issued by the **Let's Encrypt** certificate authority.

Basic configuration

Here's a basic excerpt from a Caddyfile in `/etc/caddy/Caddyfile`. This will proxy all requests from `http(s)://domain.com/...` to `http://localhost:5984/...`

```
domain.com {

   reverse_proxy localhost:5984

}
```

Reverse proxying CouchDB in a subdirectory with Caddy 2

It can be useful to provide CouchDB as a subdirectory of your overall domain, especially to avoid CORS concerns. Here's an excerpt of a basic Caddy configuration that proxies the URL `http(s)://domain.com/couchdb` to `http://localhost:5984` so that requests appended to the subdirectory, such as `http(s)://domain.com/couchdb/db1/doc1` are proxied to `http://localhost:5984/db1/doc1`.

```
domain.com {

    reverse_proxy /couchdb/* localhost:5984

}
```

Reverse proxying + load balancing for CouchDB clusters

Here's a basic excerpt from a Caddyfile in `/<path>/<to>/<site>/Caddyfile`. This will proxy and evenly distribute all requests from `http(s)://domain.com/...` among 3 CouchDB cluster nodes at `localhost:15984`, `localhost:25984` and `localhost:35984`.

Caddy will check the status, i.e. health, of each node every 5 seconds; if a node goes down, Caddy will avoid proxying requests to that node until it comes back online.

```
domain.com {

    reverse_proxy http://localhost:15984 http://localhost:25984 http://
↪localhost:35984 {
    lb_policy round_robin
    lb_try_interval 500ms

    health_interval 5s
    }

}
```

Authentication with Caddy 2 as a reverse proxy

Here's a sample config setting with basic authentication enabled, placing CouchDB in the `/couchdb` subdirectory:

```
domain.com {

    basicauth /couchdb/* {
        couch_username couchdb_hashed_password_base64
    }

    reverse_proxy /couchdb/* localhost:5984

}
```

This setup leans entirely on nginx performing authorization, and forwarding requests to CouchDB with no authentication (with CouchDB in Admin Party mode), which isn't sufficient in CouchDB 3.0 anymore as Admin Party has been removed. You'd need to at the very least hard-code user credentials into this version with headers.

For a better solution, see *Proxy Authentication*.

4.6.4 Reverse Proxying with Apache HTTP Server

> **Warning:** As of this writing, there is no way to fully disable the buffering between Apache HTTPD Server and CouchDB. This may present problems with continuous replication. The Apache CouchDB team strongly recommend the use of an alternative reverse proxy such as `haproxy` or `nginx`, as described earlier in this section.

Basic Configuration

Here's a basic excerpt for using a `VirtualHost` block config to use Apache as a reverse proxy for CouchDB. You need at least to configure Apache with the `--enable-proxy --enable-proxy-http` options and use a version equal to or higher than Apache 2.2.7 in order to use the `nocanon` option in the `ProxyPass` directive. The `ProxyPass` directive adds the `X-Forwarded-For` header needed by CouchDB, and the `ProxyPreserveHost` directive ensures the original client `Host` header is preserved.

```
<VirtualHost *:80>
   ServerAdmin webmaster@dummy-host.example.com
   DocumentRoot "/opt/websites/web/www/dummy"
   ServerName couchdb.localhost
   AllowEncodedSlashes On
   ProxyRequests Off
   KeepAlive Off
   <Proxy *>
      Order deny,allow
      Deny from all
      Allow from 127.0.0.1
   </Proxy>
   ProxyPass / http://localhost:5984 nocanon
   ProxyPassReverse / http://localhost:5984
   ProxyPreserveHost On
   ErrorLog "logs/couchdb.localhost-error_log"
   CustomLog "logs/couchdb.localhost-access_log" common
</VirtualHost>
```

CHAPTER

FIVE

INSTALLATION

5.1 Installation on Unix-like systems

Warning: CouchDB 3.0+ will not run without an admin user being created first. Be sure to *create an admin user* before starting CouchDB!

5.1.1 Installation using the Apache CouchDB convenience binary packages

If you are running one of the following operating systems, the easiest way to install CouchDB is to use the convenience binary packages:

- CentOS/RHEL 7
- CentOS/RHEL 8
- CentOS/RHEL 9 (with caveats)
- Debian 10 (buster)
- Debian 11 (bullseye)
- Debian 12 (bookworm)
- Ubuntu 18.04 (bionic)
- Ubuntu 20.04 (focal)
- Ubuntu 22.04 (jammy)

These RedHat-style rpm packages and Debian-style deb packages will install CouchDB at `/opt/couchdb` and ensure CouchDB is run at system startup by the appropriate init subsystem (SysV-style initd or systemd).

The Debian-style deb packages *also* pre-configure CouchDB as a standalone or clustered node, prompt for the address to which it will bind, and a password for the admin user. Responses to these prompts may be pre-seeded using standard `debconf` tools. Further details are in the README.Debian file.

For distributions lacking a compatible SpiderMonkey library, Apache CouchDB also provides packages for the 1.8.5 version.

Enabling the Apache CouchDB package repository

Debian or Ubuntu: Run the following commands:

```
sudo apt update && sudo apt install -y curl apt-transport-https gnupg
curl https://couchdb.apache.org/repo/keys.asc | gpg --dearmor | sudo tee /usr/share/
↪keyrings/couchdb-archive-keyring.gpg >/dev/null 2>&1
source /etc/os-release
echo "deb [signed-by=/usr/share/keyrings/couchdb-archive-keyring.gpg] https://apache.
↪jfrog.io/artifactory/couchdb-deb/ ${VERSION_CODENAME} main" \
    | sudo tee /etc/apt/sources.list.d/couchdb.list >/dev/null
```

RedHat(<9) or CentOS: Run the following commands:

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://couchdb.apache.org/repo/couchdb.repo
```

RedHat(>=9): Run the following commands:

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://couchdb.apache.org/repo/couchdb.repo
# Enable EPEL for the SpiderMonkey dependency
sudo dnf config-manager --set-enabled crb
sudo dnf install epel-release epel-next-release
```

Installing the Apache CouchDB packages

Debian or Ubuntu: Run the following commands:

```
sudo apt update
sudo apt install -y couchdb
```

Debian/Ubuntu installs from binaries can be pre-configured for single node or clustered installations. For clusters, multiple nodes will still need to be joined together and configured consistently across all machines; **follow the** *Cluster Setup* **walkthrough** to complete the process.

RedHat(<9)/CentOS: Run the command:

```
sudo yum install -y couchdb
```

RedHat(>=9): Run the following commands:

```
sudo yum install -y mozjs78
sudo yum install -y couchdb
```

Once installed, *create an admin user* by hand before starting CouchDB, if your installer didn't do this for you already.

You can now start the service.

Your installation is not complete. Be sure to complete the *Setup* **steps for a single node or clustered installation.**

Relax! CouchDB is installed and running.

GPG keys used for signing the CouchDB repositories

As of 2021.04.25, the *repository* signing key for both types of supported packages is:

```
pub   rsa8192 2015-01-19 [SC]
      390EF70BB1EA12B2773962950EE62FB37A00258D
uid           The Apache Software Foundation (Package repository signing key)
↪<root@apache.org>
```

As of 2021.04.25, the *package* signing key (only used for `rpm` packages) is:

```
pub   rsa4096 2017-07-28 [SC] [expires: 2022-07-27]
      2EC788AE3F239FA13E82D215CDE711289384AE37
uid           Joan Touzet (Apache Code Signing Key) <wohali@apache.org>
```

As of 2021.11.13, the *package* signing key (only used for `rpm` packages) is:

```
pub   rsa4096 2019-09-05 [SC] [expires: 2039-01-02]
      0BD7A98499C4AB41C910EE65FC04DFBC9657A78E
uid           Nicolae Vatamaniuc <vatamane@apache.org>
uid           default <vatamane@gmail.com>
```

All are available from most popular GPG key servers. The `rpm` signing keys should be listed in the KEYS list as well.

5.1.2 Installation from source

The remainder of this document describes the steps required to install CouchDB directly from source code.

This guide, as well as the INSTALL.Unix document in the official tarball release are the canonical sources of installation information. However, many systems have gotchas that you need to be aware of. In addition, dependencies frequently change as distributions update their archives.

5.1.3 Dependencies

You should have the following installed:

- Erlang OTP (24.x, 25.x)
- ICU
- OpenSSL
- Mozilla SpiderMonkey (1.8.5, 60, 68, 78, 91)
- GNU Make
- GNU Compiler Collection
- help2man
- Python (>=3.6) for docs and tests
- Java (required for *nouveau*, minimum version 11, recommended version 19 or 20)

help2man is only need if you plan on installing the CouchDB man pages. Documentation build can be disabled by adding the `--disable-docs` flag to the `configure` script.

Debian-based Systems

You can install the dependencies by running:

```
sudo apt-get --no-install-recommends -y install \
    build-essential pkg-config erlang \
    libicu-dev libmozjs185-dev
```

Be sure to update the version numbers to match your system's available packages.

RedHat-based (Fedora, CentOS, RHEL) Systems

You can install the dependencies by running:

```
sudo yum install autoconf autoconf-archive automake \
    erlang-asn1 erlang-erts erlang-eunit gcc-c++ \
    erlang-os_mon erlang-xmerl erlang-erl_interface help2man \
    libicu-devel libtool perl-Test-Harness
```

Warning: To build a release for CouchDB the erlang-reltool package is required, yet on CentOS/RHEL this package depends on erlang-wx which pulls in wxGTK and several X11 libraries. If CouchDB is being built on a console only server it might be a good idea to install this in a separate step to the rest of the dependencies, so that the package and all its dependencies can be removed using the `yum history` tool after the release is built. (reltool is needed only during release build but not for CouchDB functioning)

The package can be installed by running:

```
sudo yum install erlang-reltool
```

Fedora 36

On Fedora 36, you may need these packages in addition to the ones listed above:

- *mozjs91-devel*
- *erlang-rebar*

If the system contains dangling links to Erlang chunk files, the compiler will abort. They can be deleted with the following command:

```
find -L /usr/lib64/erlang/lib/ -type l -name chunks | xargs rm -f
```

Fauxton is not built on the Node.js version (v16) shipped by the system. The installation of v12.22.12 can be done via:

```
wget https://nodejs.org/download/release/v12.22.12/node-v12.22.12-linux-x64.tar.gz
mkdir -p /usr/local/lib/nodejs
tar -xvf node-v12.22.12-linux-x64.tar.gz -C /usr/local/lib/nodejs
export PATH=/usr/local/lib/nodejs/node-v12.22.12-linux-x64/bin:$PATH
```

Note that due to a problem with the Python package sphinx-build, it is not possible to compile the documentation on Fedora 36. You can skip compiling the documentation via:

```
./configure --disable-docs --spidermonkey-version 91
```

Mac OS X

Follow *Installation with Homebrew* reference for Mac App installation.

If you are installing from source, you will need to install the Command Line Tools:

```
xcode-select --install
```

You can then install the other dependencies by running:

```
brew install autoconf autoconf-archive automake libtool \
    erlang icu4c spidermonkey pkg-config
```

You will need *Homebrew* installed to use the `brew` command.

Some versions of Mac OS X ship a problematic OpenSSL library. If you're experiencing troubles with CouchDB crashing intermittently with a segmentation fault or a bus error, you will need to install your own version of OpenSSL. See the wiki, mentioned above, for more information.

See also:

- Homebrew

FreeBSD

FreeBSD requires the use of GNU Make. Where `make` is specified in this documentation, substitute `gmake`.

You can install this by running:

```
pkg install gmake
```

5.1.4 Installing

Once you have satisfied the dependencies you should run:

```
./configure
```

If you wish to customize the installation, pass `--help` to this script.

If everything was successful you should see the following message:

```
You have configured Apache CouchDB, time to relax.
```

Relax.

To build CouchDB you should run:

```
make release
```

Try `gmake` if `make` is giving you any problems.

If include paths or other compiler options must be specified, they can be passed to rebar, which compiles CouchDB, with the ERL_CFLAGS environment variable. Likewise, options may be passed to the linker with the ERL_LDFLAGS environment variable:

```
make release ERL_CFLAGS="-I/usr/local/include/js -I/usr/local/lib/erlang/usr/include"
```

If everything was successful you should see the following message:

```
... done
You can now copy the rel/couchdb directory anywhere on your system.
Start CouchDB with ./bin/couchdb from within that directory.
```

Relax.

Note: a fully-fledged `./configure` with the usual GNU Autotools options for package managers and a corresponding `make install` are in development, but not part of the 2.0.0 release.

5.1.5 User Registration and Security

For OS X, in the steps below, substitute `/Users/couchdb` for `/home/couchdb`.

You should create a special `couchdb` user for CouchDB.

On many Unix-like systems you can run:

```
adduser --system \
        --shell /bin/bash \
        --group --gecos \
        "CouchDB Administrator" couchdb
```

On Mac OS X you can use the Workgroup Manager to create users up to version 10.9, and dscl or sysadminctl after version 10.9. Search Apple's support site to find the documentation appropriate for your system. As of recent versions of OS X, this functionality is also included in Server.app, available through the App Store only as part of OS X Server.

You must make sure that the user has a working POSIX shell and a writable home directory.

You can test this by:

- Trying to log in as the `couchdb` user
- Running `pwd` and checking the present working directory

As a recommendation, copy the `rel/couchdb` directory into `/home/couchdb` or `/Users/couchdb`.

Ex: copy the built couchdb release to the new user's home directory:

```
cp -R /path/to/couchdb/rel/couchdb /home/couchdb
```

Change the ownership of the CouchDB directories by running:

```
chown -R couchdb:couchdb /home/couchdb
```

Change the permission of the CouchDB directories by running:

```
find /home/couchdb -type d -exec chmod 0770 {} \;
```

Update the permissions for your ini files:

```
chmod 0644 /home/couchdb/etc/*
```

5.1.6 First Run

Note: Be sure to *create an admin user* before trying to start CouchDB!

You can start the CouchDB server by running:

```
sudo -i -u couchdb /home/couchdb/bin/couchdb
```

This uses the `sudo` command to run the `couchdb` command as the `couchdb` user.

When CouchDB starts it should eventually display following messages:

```
{database_does_not_exist,[{mem3_shards,load_shards_from_db,"_users" ...
```

Don't be afraid, we will fix this in a moment.

To check that everything has worked, point your web browser to:

```
http://127.0.0.1:5984/_utils/index.html
```

From here you should verify your installation by pointing your web browser to:

```
http://localhost:5984/_utils/index.html#verifyinstall
```

Your installation is not complete. Be sure to complete the *Setup* steps for a single node or clustered installation.

5.1.7 Running as a Daemon

CouchDB no longer ships with any daemonization scripts.

The CouchDB team recommends runit to run CouchDB persistently and reliably. According to official site:

> *runit* is a cross-platform Unix init scheme with service supervision, a replacement for sysvinit, and other init schemes. It runs on GNU/Linux, *BSD, MacOSX, Solaris, and can easily be adapted to other Unix operating systems.

Configuration of runit is straightforward; if you have questions, contact the CouchDB user mailing list or IRC-channel #couchdb in FreeNode network.

Let's consider configuring runit on Ubuntu 18.04. The following steps should be considered only as an example. Details will vary by operating system and distribution. Check your system's package management tools for specifics.

Install runit:

```
sudo apt-get install runit
```

Create a directory where logs will be written:

```
sudo mkdir /var/log/couchdb
sudo chown couchdb:couchdb /var/log/couchdb
```

Create directories that will contain runit configuration for CouchDB:

```
sudo mkdir /etc/sv/couchdb
sudo mkdir /etc/sv/couchdb/log
```

Create /etc/sv/couchdb/log/run script:

```
#!/bin/sh
exec svlogd -tt /var/log/couchdb
```

Basically it determines where and how exactly logs will be written. See `man svlogd` for more details.

Create /etc/sv/couchdb/run:

```
#!/bin/sh
export HOME=/home/couchdb
exec 2>&1
exec chpst -u couchdb /home/couchdb/bin/couchdb
```

This script determines how exactly CouchDB will be launched. Feel free to add any additional arguments and environment variables here if necessary.

Make scripts executable:

```
sudo chmod u+x /etc/sv/couchdb/log/run
sudo chmod u+x /etc/sv/couchdb/run
```

Then run:

```
sudo ln -s /etc/sv/couchdb/ /etc/service/couchdb
```

In a few seconds runit will discover a new symlink and start CouchDB. You can control CouchDB service like this:

```
sudo sv status couchdb
sudo sv stop couchdb
sudo sv start couchdb
```

Naturally now CouchDB will start automatically shortly after system starts.

You can also configure systemd, launchd or SysV-init daemons to launch CouchDB and keep it running using standard configuration files. Consult your system documentation for more information.

5.2 Installation on Windows

There are two ways to install CouchDB on Windows.

5.2.1 Installation from binaries

This is the simplest way to go.

Warning: Windows 8, 8.1, and 10 require the .NET Framework v3.5 to be installed.

1. Get the latest Windows binaries from the CouchDB web site. Old releases are available at archive.
2. Follow the installation wizard steps. **Be sure to install CouchDB to a path with no spaces, such as** `C:\CouchDB`.
3. **Your installation is not complete. Be sure to complete the** *Setup* **steps for a single node or clustered installation.**
4. Open up Fauxton
5. It's time to Relax!

Note: In some cases you might been asked to reboot Windows to complete installation process, because of using on different Microsoft Visual C++ runtimes by CouchDB.

Note: Upgrading note

It's recommended to uninstall previous CouchDB version before upgrading, especially if the new one is built against different Erlang release. The reason is simple: there may be leftover libraries with alternative or incompatible versions from old Erlang release that may create conflicts, errors and weird crashes.

In this case, make sure you backup of your *local.ini* config and CouchDB database/index files.

Silent Install

The Windows installer supports silent installs. Here are some sample commands, supporting the new features of the 3.0 installer.

Install CouchDB without a service, but with an admin user:password of `admin:hunter2`:

```
msiexec /i apache-couchdb-3.0.0.msi /quiet ADMINUSER=admin ADMINPASSWORD=hunter2 /
↪norestart
```

The same as above, but also install and launch CouchDB as a service:

```
msiexec /i apache-couchdb-3.0.0.msi /quiet INSTALLSERVICE=1 ADMINUSER=admin␣
↪ADMINPASSWORD=hunter2 /norestart
```

Unattended uninstall of CouchDB to target directory *D:CouchDB*:

```
msiexec /x apache-couchdb-3.0.0.msi INSTALLSERVICE=1 APPLICATIONFOLDER="D:\CouchDB"␣
↪ADMINUSER=admin ADMINPASSWORD=hunter2 /quiet /norestart
```

Unattended uninstall if the installer file is unavailable:

```
msiexec /x {4CD776E0-FADF-4831-AF56-E80E39F34CFC} /quiet /norestart
```

Add `/l* log.txt` to any of the above to generate a useful logfile for debugging.

5.2.2 Installation from sources

See also:

Glazier: Automate building of CouchDB from source on Windows

5.3 Installation on macOS

5.3.1 Installation using the Apache CouchDB native application

The easiest way to run CouchDB on macOS is through the native macOS application. Just follow the below instructions:

1. Download Apache CouchDB for macOS. Old releases are available at archive.
2. Double click on the Zip file
3. Drag and drop the Apache CouchDB.app into Applications folder

That's all, now CouchDB is installed on your Mac:

1. Run Apache CouchDB application
2. Open up Fauxton, the CouchDB admin interface
3. Verify the install by clicking on *Verify*, then *Verify Installation*.
4. **Your installation is not complete. Be sure to complete the** *Setup* **steps for a single node or clustered installation.**
5. Time to Relax!

5.3.2 Installation with Homebrew

CouchDB can be installed via Homebrew. Fetch the newest version of Homebrew and all formulae and install CouchDB with the following commands:

```
brew update
brew install couchdb
```

5.3.3 Installation from source

Installation on macOS is possible from source. Download the source tarball, extract it, and follow the instructions in the `INSTALL.Unix.md` file.

Running as a Daemon

CouchDB itself no longer ships with any daemonization scripts.

The CouchDB team recommends runit to run CouchDB persistently and reliably. Configuration of runit is straightforward; if you have questions, reach out to the CouchDB user mailing list.

Naturally, you can configure launchd or other init daemons to launch CouchDB and keep it running using standard configuration files.

Consult your system documentation for more information.

5.4 Installation on FreeBSD

5.4.1 Installation

Use the pre-built binary packages to install CouchDB:

```
pkg install couchdb3
```

Alternatively, it is possible installing CouchDB from the Ports Collection:

```
cd /usr/ports/databases/couchdb3
make install clean
```

Note: Be sure to *create an admin user* before starting CouchDB for the first time!

5.4.2 Service Configuration

The port is shipped with a script that integrates CouchDB with FreeBSD's rc.d service framework. The following options for `/etc/rc.conf` or `/etc/rc.conf.local` are supported (defaults shown):

```
couchdb3_enable="NO"
couchdb3_user="couchdb"
couchdb3_erl_flags="-couch_ini /usr/local/libexec/couchdb3/etc/default.ini /usr/local/
↪etc/couchdb3/local.ini"
couchdb3_chdir="/var/db/couchdb3"
```

After enabling the `couchdb3` service (by setting `couchdb3_enable` to `"YES"`), use the following command to start CouchDB:

```
service couchdb3 start
```

This script responds to the arguments `start`, `stop`, `status`, `rcvar` etc. If the service is not yet enabled in `rc.conf`, use `onestart` to start it up ad-hoc.

The service will also use settings from the following config files:

- `/usr/local/libexec/couchdb3/etc/default.ini`
- `/usr/local/etc/couchdb3/local.ini`

The `default.ini` should be left read-only, and will be replaced on upgrades and re-installs without warning. Therefore administrators should use `default.ini` as a reference and only modify the `local.ini` file.

5.4.3 Post Install

The installation is not complete. Be sure to complete the *Setup* steps for a single node or clustered installation.

Also note that the port will probably show some messages after the installation happened. Make note of these instructions, although they can be found in the ports tree for later reference.

5.5 Installation via Docker

Apache CouchDB provides 'convenience binary' Docker images through Docker Hub at `apache/couchdb`. This is our upstream release; it is usually mirrored downstream at Docker's top-level `couchdb` as well.

At least these tags are always available on the image:

- `latest` - always the latest
- `3`: always the latest 3.x version
- `2`: always the latest 2.x version
- `1`, `1.7`, `1.7.2`: CouchDB 1.7.2 (convenience only; no longer supported)
- `1-couchperuser`, `1.7-couchperuser`, `1.7.2-couchperuser`: CouchDB 1.7.2 with couchperuser plugin (convenience only; no longer supported)

These images expose CouchDB on port `5984` of the container, run everything as user `couchdb` (uid `5984`), and support use of a Docker volume for data at `/opt/couchdb/data`.

Your installation is not complete. Be sure to complete the *Setup* steps for a single node or clustered installation.

Further details on the Docker configuration are available in our couchdb-docker git repository.

5.6 Installation via Snap

Apache CouchDB provides 'convenience binary' Snap builds through the Ubuntu snapcraft repository under the name couchdb snap. These are available in separate snap channels for each major/minor release stream, e.g., `2.x`, `3.3`, and a `latest` stream.

Once you've completed installing snapd, you can install the CouchDB snap via:

```
$ sudo snap install couchdb
```

After installation, set up an admin password and a cookie using a snap hook. Then, restart the snap for changes to take effect:

```
$ sudo snap set couchdb admin=[your-password] setcookie=[your-cookie]
$ sudo snap restart couchdb
```

CouchDB will be installed (read only) at `/snap/couchdb/current/`. Data files will be written to `/var/snap/couchdb/common/data`, and (writable) configuration files will be stored in `/var/snap/couchdb/current/etc`.

Note: Your installation is not complete. Follow the *Setup* steps for a single node or clustered installation.

Snaps use AppArmor and are closely tied to systemd. They enforce that only writable files are housed under `/var/snap`. Ensure that `/var` has sufficient space for your data requirements.

To view logs, access them via `journalctl snap.couchdb` or using the `snap logs` command:

```
$ sudo snap logs couchdb -f
```

When installing from a specific channel, snaps are automatically refreshed with new revisions. Revert to a previous installation with:

```
$ sudo snap revert couchdb
```

After this, updates will no longer be received. View installed snaps and alternative channels using the list and info commands:

```
$ snap list
$ snap info couchdb
```

As easily as they are installed, snaps can be removed:

```
$ sudo snap remove couchdb
$ sudo snap remove couchdb --purge
```

The first command stops the server, removes couchdb from the list, and the filesystem (keeping a backup for about 30 days if space permits). If you reinstall couchdb, it tries to restore the backup. The second command removes couchdb and purges any backups.

When troubleshooting couchdb snap, check the logs first. You'll likely need to inspect `/var/snap/couchdb/current/etc/local.ini` to verify the data directory or modify admin settings, port, or address bindings. Also, anything related to Erlang runtime check `/var/snap/couchdb/current/etc/vm.args` to view the erlang name.

The most common issue is couchdb not finding the database files. Ensure that `local.ini` includes the following stanza and points to your data files:

```
[couchdb]
;max_document_size = 4294967296 ; bytes
;os_process_timeout = 5000
database_dir = /var/snap/couchdb/common/data
view_index_dir = /var/snap/couchdb/common/data
```

Note: Remember, you cannot modify the `/snap/couchdb/` directory, even with sudo, as the filesystem is mounted read-only for security reasons.

For additional details on the snap build process, refer to our couchdb-pkg git repository. This includes instructions on setting up a cluster using the command line.

5.7 Installation on Kubernetes

Apache CouchDB provides a Helm chart to enable deployment to Kubernetes.

To install the chart with the release name `my-release`:

```
helm repo add couchdb https://apache.github.io/couchdb-helm

helm repo update

helm install --name my-release couchdb/couchdb
```

Further details on the configuration options are available in the Helm chart readme.

5.8 Search Plugin Installation

New in version 3.0.

CouchDB can build and query full-text search indexes using an external Java service that embeds Apache Lucene. Typically, this service is installed on the same host as CouchDB and communicates with it over the loopback network.

The search plugin is runtime-compatible with Java JDKs 6, 7 and 8. Building a release from source requires JDK 6. **It will not work with any newer version of Java.** Sorry about that.

5.8.1 Installation of Binary Packages

Binary packages that bundle all the necessary dependencies of the search plugin are available on GitHub. The files in each release should be unpacked into a directory on the Java classpath. If you do not have a classpath already set, or you wish to explicitly set the classpath location for Clouseau, then add the line:

```
-classpath '/path/to/clouseau/*'
```

to the server command below. If clouseau is installed in `/opt/clouseau` the line would be:

```
-classpath '/opt/clouseau/*'
```

The service expects to find a couple of configuration files conventionally called `clouseau.ini` and `log4j.properties` with the following content:

clouseau.ini:

```
[clouseau]

; the name of the Erlang node created by the service, leave this unchanged
name=clouseau@127.0.0.1

; set this to the same distributed Erlang cookie used by the CouchDB nodes
cookie=brumbrum

; the path where you would like to store the search index files
dir=/path/to/index/storage

; the number of search indexes that can be open simultaneously
max_indexes_open=500
```

log4j.properties:

```
log4j.rootLogger=debug, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} %c [%p] %m%n
```

Once these files are in place the service can be started with an invocation like the following:

```
java -server \
     -Xmx2G \
     -Dsun.net.inetaddr.ttl=30 \
     -Dsun.net.inetaddr.negative.ttl=30 \
     -Dlog4j.configuration=file:/path/to/log4j.properties \
     -XX:OnOutOfMemoryError="kill -9 %p" \
     -XX:+UseConcMarkSweepGC \
     -XX:+CMSParallelRemarkEnabled \
     com.cloudant.clouseau.Main \
     /path/to/clouseau.ini
```

5.8.2 Chef

The CouchDB cookbook can build the search plugin from source and install it on a server alongside CouchDB.

5.8.3 Kubernetes

Users running CouchDB on Kubernetes via the Helm chart can add the search service to each CouchDB Pod by setting `enableSearch: true` in the chart values.

5.8.4 Additional Details

The *Search User Guide* provides detailed information on creating and querying full-text indexes using this plugin.

The source code for the plugin and additional configuration documentation is available on GitHub at https://github.com/cloudant-labs/clouseau.

5.9 Nouveau Server Installation

New in version 4.0.

CouchDB can build and query full-text search indexes using an external Java service that embeds Apache Lucene. Typically, this service is installed on the same host as CouchDB and communicates with it over the loopback network.

Nouveau server is runtime-compatible with Java 11 or higher.

5.9.1 Enable Nouveau

You need to enable nouveau in CouchDB configuration;

```
[nouveau]
enable = true
```

5.9.2 Installation of Binary Packages

The nouveau server code is contained in a single `jar` file that ships with a compatible CouchDB release, named `server-$version-dist.jar`

To start the nouveau server:

```
java -jar /path/to/server-$version-dist.jar server /path/to/nouveau.yaml
```

We ship a basic `nouveau.yaml` configuration with useful defaults; that file for details.

nouveau.yaml:

```
maxIndexesOpen: 100
commitIntervalSeconds: 30
idleSeconds: 60
rootDir: target/indexes
```

As a DropWizard project you can also use the many configuration options that it supports. See configuration reference.

By default Nouveau will attempt a clean shutdown if sent a `TERM` signal, committing any outstanding index updates, completing any in-progress segment merges, and finally closes all indexes. This is not essential and you may safely kill the JVM without letting it do this, though any uncommitted changes are necessarily lost. Once the JVM is started again this indexing work will be attempted again.

5.10 Upgrading from prior CouchDB releases

5.10.1 Important Notes

- **Always back up your `data/` and `etc/` directories prior to upgrading CouchDB.**
- We recommend that you overwrite your `etc/default.ini` file with the version provided by the new release. New defaults sometimes contain mandatory changes to enable default functionality. Always places your customizations in `etc/local.ini` or any `etc/local.d/*.ini` file.

5.10.2 Upgrading from CouchDB 2.x

If you are coming from a prior release of CouchDB 2.x, upgrading is simple.

Standalone (single) node upgrades

If you are running a standalone (single) CouchDB node:

1. Plan for downtime.
2. Backup everything.
3. Check for new recommended settings in the shipped `etc/local.ini` file, and merge any changes desired into your own local settings file(s).
4. Stop CouchDB.
5. Upgrade CouchDB in place.
6. Be sure to *create an admin user* if you do not have one. CouchDB 3.0+ **require** an admin user to start (the admin party has ended).
7. Start CouchDB.
8. Relax! You're done.

Cluster upgrades

CouchDB 2.x and 3.x are explicitly designed to allow "mixed clusters" during the upgrade process. This allows you to perform a rolling restart across a cluster, upgrading one node at a time, for a *zero downtime upgrade*. The process is also entirely scriptable within your configuration management tool of choice.

We're proud of this feature, and you should be, too!

If you are running a CouchDB cluster:

1. Backup everything.
2. Check for new recommended settings in the shipped `etc/local.ini` file, and merge any changes desired into your own local settings file(s), staging these changes to occur as you upgrade the node.
3. Stop CouchDB on a single node.
4. Upgrade that CouchDB install in place.
5. Start CouchDB.
6. Double-check that the node has re-joined the cluster through the */_membership* endpoint. If your load balancer has health check functionality driven by the */_up* endpoint, check whether it thinks the node is healthy as well.
7. Repeat the last 4 steps on the remaining nodes in the cluster.
8. Relax! You're done.

5.10.3 Upgrading from CouchDB 1.x

To upgrade from CouchDB 1.x, first upgrade to a version of CouchDB 2.x. You will need to convert all databases to CouchDB 2.x format first; see the Upgrade Notes there for instructions. Then, upgrade to CouchDB 3.x.

5.11 Troubleshooting an Installation

5.11.1 First Install

If your CouchDB doesn't start after you've just installed, check the following things:

- On UNIX-like systems, this is usually this is a permissions issue. Ensure that you've followed the *User Registration and Security* `chown/chmod` commands. This problem is indicated by the presence of the keyword `eacces` somewhere in the error output from CouchDB itself.
- Some Linux distributions split up Erlang into multiple packages. For your distribution, check that you **really** installed all the required Erlang modules. This varies from platform to platform, so you'll just have to work it out for yourself. For example, on recent versions of Ubuntu/Debian, the `erlang` package includes all Erlang modules.
- Confirm that Erlang itself starts up with crypto (SSL) support:

```
## what version of erlang are you running? Ensure it is supported
erl -noshell -eval 'io:put_chars(erlang:system_info(otp_release)).' -s erlang halt
## are the erlang crypto (SSL) libraries working?
erl -noshell -eval 'case application:load(crypto) of ok -> io:put_chars("yay_crypto\n
↪") ; _ -> exit(no_crypto) end.' -s init stop
```

- Next, identify where your Erlang CouchDB libraries are installed. This will typically be the lib/ subdirectory of the release that you have installed.
- Use this to start up Erlang with the CouchDB libraries in its path:

```
erl -env ERL_LIBS $ERL_LIBS:/path/to/couchdb/lib -couch_ini -s crypto
```

- In that Erlang shell, let's check that the key libraries are running. The `%%` lines are comments, so you can skip them:

```
%% test SSL support. If this fails, ensure you have the OTP erlang-crypto library␣
↪installed
crypto:md5_init().

%% test Snappy compression. If this fails, check your CouchDB configure script output␣
↪or alternatively
%% if your distro comes with erlang-snappy make sure you're using only the CouchDB␣
↪supplied version
snappy:compress("gogogogogogogogogogogogogogo").

%% test the CouchDB JSON encoder. CouchDB uses different encoders in each release,␣
↪this one matches
%% what is used in 2.0.x.
jiffy:decode(jiffy:encode(<<"[1,2,3,4,5]">>)).

%% this is how you quit the erlang shell.
q().
```

- The output should resemble this, or an error will be thrown:

```
Erlang/OTP 17 [erts-6.2] [source] [64-bit] [smp:2:2] [async-threads:10] [kernel-
↪poll:false]

Eshell V6.2  (abort with ^G)
1> crypto:md5_init().
<<1,35,69,103,137,171,205,239,254,220,186,152,118,84,50,
```

(continues on next page)

(continued from previous page)

```
  16,0,0,0,0,0,0,0,0,0,0,0,0,0,...>>
2> snappy:compress("gogogogogogogogogogogogogogo").
{ok,<<28,4,103,111,102,2,0>>}
3> jiffy:decode(jiffy:encode(<<"[1,2,3,4,5]">>)).
<<"[1,2,3,4,5]">>
4> q().
```

- At this point the only remaining dependency is your system's Unicode support library, ICU, and the Spidermonkey Javascript VM from Mozilla. Make sure that your `LD_LIBRARY_PATH` or equivalent for non-Linux systems (`DYLD_LIBRARY_PATH` on macOS) makes these available to CouchDB. Linux example running as normal user:

```
  LD_LIBRARY_PATH=/usr/local/lib:/usr/local/spidermonkey/lib couchdb

Linux example running as couchdb user:
```

```
echo LD_LIBRARY_PATH=/usr/local/lib:/usr/local/spidermonkey/lib couchdb | sudo -u␣
↪couchdb sh
```

- If you receive an error message including the key word `eaddrinuse`, such as this:

```
  Failure to start Mochiweb: eaddrinuse

edit your ``etc/default.ini`` or ``etc/local.ini`` file and change the
``[chttpd] port = 5984`` line to an available port.
```

- If you receive an error including the string:

```
... OS Process Error ... {os_process_error,{exit_status,127}}
```

then it is likely your SpiderMonkey JavaScript VM installation is not correct. Please recheck your build dependencies and try again.

- If you receive an error including the string:

```
... OS Process Error ... {os_process_error,{exit_status,139}}
```

this is caused by the fact that SELinux blocks access to certain areas of the file system. You must re-configure SELinux, or you can fully disable SELinux using the command:

```
setenforce 0
```

- If you are still not able to get CouchDB to start at this point, keep reading.

5.11.2 Quick Build

Having problems getting CouchDB to run for the first time? Follow this simple procedure and report back to the user mailing list or IRC with the output of each step. Please put the output of these steps into a paste service (such as https://paste.ee/) rather than including the output of your entire run in IRC or the mailing list directly.

1. Note down the name and version of your operating system and your processor architecture.
2. Note down the installed versions of CouchDB's dependencies.
3. Follow the checkout instructions to get a fresh copy of CouchDB's trunk.
4. Configure from the couchdb directory:

```
./configure
```

5. Build the release:

```
make release
```

6. Run the couchdb command and log the output:

```
cd rel/couchdb
bin/couchdb
```

7. Use your system's kernel trace tool and log the output of the above command.
   a) For example, linux systems should use `strace`:

```
strace bin/couchdb 2> strace.out
```

8. Report back to the mailing list (or IRC) with the output of each step.

5.11.3 Upgrading

Are you upgrading from CouchDB 1.x? Install CouchDB into a fresh directory. CouchDB's directory layout has changed and may be confused by libraries present from previous releases.

5.11.4 Runtime Errors

Erlang stack trace contains `system_limit`, `open_port`, or `emfile`

Modern Erlang has a default limit of 65536 ports (8196 on Windows), where each open file handle, tcp connection, and linked-in driver uses one port. OSes have different soft and hard limits on the number of open handles per process, often as low as 1024 or 4096 files. You've probably exceeded this.

There are two settings that need changing to increase this value. Consult your OS documentation for how to increase the limit for your process. Under Linux and systemd, this setting can be adjusted via `systemctl edit couchdb` and adding the lines:

```
[Service]
LimitNOFILE=65536
```

to the file in the editor.

To increase this value higher than 65536, you must also add the Erlang `+Q` parameter to your `etc/vm.args` file by adding the line:

```
+Q 102400
```

The old `ERL_MAX_PORTS` environment variable is ignored by the version of Erlang supplied with CouchDB.

Lots of memory being used on startup

Is your CouchDB using a lot of memory (several hundred MB) on startup? This one seems to especially affect Dreamhost installs. It's really an issue with the Erlang VM pre-allocating data structures when ulimit is very large or unlimited. A detailed discussion can be found on the erlang-questions list, but the short answer is that you should decrease `ulimit -n` or lower the `vm.args` parameter `+Q` to something reasonable like 1024.

Function raised exception (Cannot encode 'undefined' value as JSON)

If you see this in the CouchDB error logs, the JavaScript code you are using for either a map or reduce function is referencing an object member that is not defined in at least one document in your database. Consider this document:

```
{
  "_id":"XYZ123",
  "_rev":"1BB2BB",
  "field":"value"
}
```

and this map function:

```
function(doc) {
  emit(doc.name, doc.address);
}
```

This will fail on the above document, as it does not contain a `name` or `address` member. Instead, use guarding to make sure the function only accesses members when they exist in a document:

```
function(doc) {
  if(doc.name && doc.address) {
    emit(doc.name, doc.address);
  }
}
```

While the above guard will work in most cases, it's worth bearing JavaScript's understanding of 'false' values in mind. Testing against a property with a value of 0 (zero), `''` (empty String), `false` or `null` will return false. If this is undesired, a guard of the form `if (doc.foo !== undefined)` should do the trick.

This error can also be caused if a reduce function does not return a value. For example, this reduce function will cause an error:

```
function(key, values) {
  sum(values);
}
```

The function needs to return a value:

```
function(key, values) {
  return sum(values);
}
```

Erlang stack trace contains `bad_utf8_character_code`

CouchDB 1.1.1 and later contain stricter handling of UTF8 encoding. If you are replicating from older versions to newer versions, then this error may occur during replication.

A number of work-arounds exist; the simplest is to do an in-place upgrade of the relevant CouchDB and then compact prior to replicating.

Alternatively, if the number of documents impacted is small, use filtered replication to exclude only those documents.

FIPS mode

Operating systems can be configured to disallow the use of OpenSSL MD5 hash functions in order to prevent use of MD5 for cryptographic purposes. CouchDB makes use of MD5 hashes for verifying the integrity of data (and not for cryptography) and will not run without the ability to use MD5 hashes.

The message below indicates that the operating system is running in "FIPS mode," which, among other restrictions, does not allow the use of OpenSSL's MD5 functions:

```
md5_dgst.c(82): OpenSSL internal error, assertion failed: Digest MD5 forbidden in␣
↪FIPS mode!
[os_mon] memory supervisor port (memsup): Erlang has closed
[os_mon] cpu supervisor port (cpu_sup): Erlang has closed
Aborted
```

A workaround for this is provided with the `--erlang-md5` compile flag. Use of the flag results in CouchDB substituting the OpenSSL MD5 function calls with equivalent calls to Erlang's built-in library `erlang:md5.` NOTE: there may be a performance penalty associated with this workaround.

Because CouchDB does not make use of MD5 hashes for cryptographic purposes, this workaround does not defeat the purpose of "FIPS mode," provided that the system owner is aware of and consents to its use.

Debugging startup

If you've compiled from scratch and are having problems getting CouchDB to even start up, you may want to see more detail. Start by enabling logging at the debug level:

```
[log]
level = debug
```

You can then pass the `-init_debug +W i +v +V -emu_args` flags in the `ERL_FLAGS` environment variable to turn on additional debugging information that CouchDB developers can use to help you.

Then, reach out to the CouchDB development team using the links provided on the CouchDB home page for assistance.

5.11.5 macOS Known Issues

undefined error, exit_status 134

Sometimes the `Verify Installation` fails with an `undefined` error. This could be due to a missing dependency with Mac. In the logs, you will find `couchdb exit_status,134`.

Installing the missing `nspr` via `brew install nspr` resolves the issue. (see: https://github.com/apache/couchdb/issues/979)

CHAPTER

SIX

SETUP

CouchDB 2.x can be deployed in either a single-node or a clustered configuration. This section covers the first-time setup steps required for each of these configurations.

6.1 Single Node Setup

Many users simply need a single-node CouchDB 2.x installation. Operationally, it is roughly equivalent to the CouchDB 1.x series. Note that a single-node setup obviously doesn't take any advantage of the new scaling and fault-tolerance features in CouchDB 2.x.

After installation and initial startup, visit Fauxton at `http://127.0.0.1:5984/_utils#setup`. You will be asked to set up CouchDB as a single-node instance or set up a cluster. When you click "Single-Node-Setup", you will get asked for an admin username and password. Choose them well and remember them.

You can also bind CouchDB to a public address, so it is accessible within your LAN or the public, if you are doing this on a public VM. Or, you can keep the installation private by binding only to 127.0.0.1 (localhost). Binding to 0.0.0.0 will bind to all addresses. The wizard then configures your admin username and password and creates the three system databases `_users`, `_replicator` and `_global_changes` for you.

Another option is to set the configuration parameter `[couchdb] single_node=true` in your `local.ini` file. When doing this, CouchDB will create the system database for you on restart.

Alternatively, if you don't want to use the Setup Wizard or set that value, and run 3.x as a single node with a server administrator already configured via *config file*, make sure to create the three system databases manually on startup:

```
curl -X PUT http://adm:pass@127.0.0.1:5984/_users

curl -X PUT http://adm:pass@127.0.0.1:5984/_replicator

curl -X PUT http://adm:pass@127.0.0.1:5984/_global_changes
```

Note that the last of these is not necessary if you do not expect to be using the global changes feed. Feel free to delete this database if you have created it, it has grown in size, and you do not need the function (and do not wish to waste system resources on compacting it regularly.)

6.2 Cluster Set Up

This section describes everything you need to know to prepare, install, and set up your first CouchDB 2.x/3.x cluster.

6.2.1 Ports and Firewalls

CouchDB uses the following ports:

Port Number	Protocol	Recommended binding	Usage
5984	tcp	As desired, by default `localhost`	Standard clustered port for all HTTP API requests
4369	tcp	`localhost` for single node installs. Private interface if clustered	Erlang port mapper daemon (epmd)
Random above 1024 (see below)	tcp	Private interface	Communication with other CouchDB nodes in the cluster

CouchDB in clustered mode uses the port `5984`, just as in a standalone configuration. Port `5986`, previously used in CouchDB 2.x, has been removed in CouchDB 3.x. All endpoints previously accessible at that port are now available under the `/_node/{node-name}/...` hierarchy via the primary `5984` port.

CouchDB uses Erlang-native clustering functionality to achieve a clustered installation. Erlang uses TCP port `4369` (EPMD) to find other nodes, so all servers must be able to speak to each other on this port. In an Erlang cluster, all nodes are connected to all other nodes, in a mesh network configuration.

Every Erlang application running on that machine (such as CouchDB) then uses automatically assigned ports for communication with other nodes. Yes, this means random ports. This will obviously not work with a firewall, but it is possible to force an Erlang application to use a specific port range.

This documentation will use the range TCP `9100-9200`, but this range is unnecessarily broad. If you only have a single Erlang application running on a machine, the range can be limited to a single port: `9100-9100`, since the ports erlang assigns are for *inbound connections* only. Three CouchDB nodes running on a single machine, as in a development cluster scenario, would need three ports in this range.

> **Warning:** If you expose the distribution port to the Internet or any other untrusted network, then the only thing protecting you is the Erlang cookie.

6.2.2 Configure and Test the Communication with Erlang

Make CouchDB use correct IP|FQDN and the open ports

In file `etc/vm.args` change the line `-name couchdb@127.0.0.1` to `-name couchdb@<reachable-ip-address|fully-qualified-domain-name>` which defines the name of the node. Each node must have an identifier that allows remote systems to talk to it. The node name is of the form `<name>@<reachable-ip-address|fully-qualified-domain-name>`.

The name portion can be couchdb on all nodes, unless you are running more than 1 CouchDB node on the same server with the same IP address or domain name. In that case, we recommend names of `couchdb1`, `couchdb2`, etc.

The second portion of the node name must be an identifier by which other nodes can access this node – either the node's fully qualified domain name (FQDN) or the node's IP address. The FQDN is preferred so that you can renumber the node's IP address without disruption to the cluster. (This is common in cloud-hosted environments.)

> **Warning:** Changing the name later is somewhat cumbersome (i.e. moving shards), which is why you will want to set it once and not have to change it.

Open `etc/vm.args`, on all nodes, and add `-kernel inet_dist_listen_min 9100` and `-kernel inet_dist_listen_max 9200` like below:

```
-name ...
-setcookie ...
...
-kernel inet_dist_listen_min 9100
-kernel inet_dist_listen_max 9200
```

Again, a small range is fine, down to a single port (set both to `9100`) if you only ever run a single CouchDB node on each machine.

Confirming connectivity between nodes

For this test, you need 2 servers with working hostnames. Let us call them server1.test.com and server2.test.com. They reside at `192.168.0.1` and `192.168.0.2`, respectively.

On server1.test.com:

```
erl -name bus@192.168.0.1 -setcookie 'brumbrum' -kernel inet_dist_listen_min 9100 -
↪kernel inet_dist_listen_max 9200
```

Then on server2.test.com:

```
erl -name car@192.168.0.2 -setcookie 'brumbrum' -kernel inet_dist_listen_min 9100 -
↪kernel inet_dist_listen_max 9200
```

An explanation to the commands:

- `erl` the Erlang shell.
- `-name bus@192.168.0.1` the name of the Erlang node and its IP address or FQDN.
- `-setcookie 'brumbrum'` the "password" used when nodes connect to each other.
- `-kernel inet_dist_listen_min 9100` the lowest port in the range.
- `-kernel inet_dist_listen_max 9200` the highest port in the range.

This gives us 2 Erlang shells. shell1 on server1, shell2 on server2. Time to connect them. Enter the following, being sure to end the line with a period (`.`):

In shell1:

```
net_kernel:connect_node('car@192.168.0.2').
```

This will connect to the node called `car` on the server called `192.168.0.2`.

If that returns true, then you have an Erlang cluster, and the firewalls are open. This means that 2 CouchDB nodes on these two servers will be able to communicate with each other successfully. If you get false or nothing at all, then you have a problem with the firewall, DNS, or your settings. Try again.

If you're concerned about firewall issues, or having trouble connecting all nodes of your cluster later on, repeat the above test between all pairs of servers to confirm connectivity and system configuration is correct.

6.2.3 Preparing CouchDB nodes to be joined into a cluster

Before you can add nodes to form a cluster, you must have them listening on an IP address accessible from the other nodes in the cluster. You should also ensure that a few critical settings are identical across all nodes before joining them.

The settings we recommend you set now, before joining the nodes into a cluster, are:

1. `etc/vm.args` settings as described in the *previous two sections*
2. At least one *server administrator* user (and password)
3. Bind the node's clustered interface (port `5984`) to a reachable IP address
4. A consistent *UUID*. The UUID is used in identifying the cluster when replicating. If this value is not consistent across all nodes in the cluster, replications may be forced to rewind the changes feed to zero, leading to excessive memory, CPU and network use.
5. A consistent *`httpd secret`*. The secret is used in calculating and evaluating cookie and proxy authentication, and should be set consistently to avoid unnecessary repeated session cookie requests.

As of CouchDB 3.0, steps 4 and 5 above are automatically performed for you when using the setup API endpoints described below.

If you use a configuration management tool, such as Chef, Ansible, Puppet, etc., then you can place these settings in a `.ini` file and distribute them to all nodes ahead of time. Be sure to pre-encrypt the password (cutting and pasting from a test instance is easiest) if you use this route to avoid CouchDB rewriting the file.

If you do not use configuration management, or are just experimenting with CouchDB for the first time, use these commands *once per server* to perform steps 2-4 above. Be sure to change the `password` to something secure, and again, use the same password on all nodes. You may have to run these commands locally on each node; if so, replace `<server-IP|FQDN>` below with `127.0.0.1`.

```
# First, get two UUIDs to use later on. Be sure to use the SAME UUIDs on all nodes.
curl http://<server-IP|FQDN>:5984/_uuids?count=2

# CouchDB will respond with something like:
#   {"uuids":["60c9e8234dfba3e2fdab04bf92001142","60c9e8234dfba3e2fdab04bf92001cc2"]}
# Copy the provided UUIDs into your clipboard or a text editor for later use.
# Use the first UUID as the cluster UUID.
# Use the second UUID as the cluster shared http secret.

# Create the admin user and password:
curl -X PUT http://<server-IP|FQDN>:5984/_node/_local/_config/admins/admin -d '
↪"password"'

# Now, bind the clustered interface to all IP addresses available on this machine
curl -X PUT http://<server-IP|FQDN>:5984/_node/_local/_config/chttpd/bind_address -d '
↪"0.0.0.0"'

# If not using the setup wizard / API endpoint, the following 2 steps are required:
# Set the UUID of the node to the first UUID you previously obtained:
curl -X PUT http://<server-IP|FQDN>:5984/_node/_local/_config/couchdb/uuid -d '"FIRST-
↪UUID-GOES-HERE"'

# Finally, set the shared http secret for cookie creation to the second UUID:
curl -X PUT http://<server-IP|FQDN>:5984/_node/_local/_config/chttpd_auth/secret -d '
↪"SECOND-UUID-GOES-HERE"'
```

6.2.4 The Cluster Setup Wizard

CouchDB 2.x/3.x comes with a convenient Cluster Setup Wizard as part of the Fauxton web administration interface. For first-time cluster setup, and for experimentation, this is your best option.

It is **strongly recommended** that the minimum number of nodes in a cluster is 3. For more explanation, see the *Cluster Theory* section of this documentation.

After installation and initial start-up of all nodes in your cluster, ensuring all nodes are reachable, and the pre-configuration steps listed above, visit Fauxton at `http://<server1>:5984/_utils#setup`. You will be asked to set up CouchDB as a single-node instance or set up a cluster.

When you click "Setup Cluster" you are asked for admin credentials again, and then to add nodes by IP address. To get more nodes, go through the same install procedure for each node, using the same machine to perform the setup process. Be sure to specify the total number of nodes you expect to add to the cluster before adding nodes.

Now enter each node's IP address or FQDN in the setup wizard, ensuring you also enter the previously set server admin username and password.

Once you have added all nodes, click "Setup" and Fauxton will finish the cluster configuration for you.

To check that all nodes have been joined correctly, visit `http://<server-IP|FQDN>:5984/_membership` on each node. The returned list should show all of the nodes in your cluster:

```
{
  "all_nodes": [
    "couchdb@server1.test.com",
    "couchdb@server2.test.com",
    "couchdb@server3.test.com"
  ],
  "cluster_nodes": [
    "couchdb@server1.test.com",
    "couchdb@server2.test.com",
    "couchdb@server3.test.com"
  ]
}
```

The `cluster_nodes` section is the list of *expected* nodes; the `all_nodes` section is the list of *actually connected* nodes. Be sure the two lists match.

Now your cluster is ready and available! You can send requests to any one of the nodes, and all three will respond as if you are working with a single CouchDB cluster.

For a proper production setup, you'd now set up an HTTP reverse proxy in front of the cluster, for load balancing and SSL termination. We recommend HAProxy, but others can be used. Sample configurations are available in the *Best Practices* section.

6.2.5 The Cluster Setup API

If you would prefer to manually configure your CouchDB cluster, CouchDB exposes the `_cluster_setup` endpoint for that purpose. After installation and initial setup/config, we can set up the cluster. On each node we need to run the following command to set up the node:

```
curl -X POST -H "Content-Type: application/json" http://admin:password@127.0.0.1:5984/
↪_cluster_setup -d '{"action": "enable_cluster", "bind_address":"0.0.0.0", "username
↪": "admin", "password":"password", "node_count":"3"}'
```

After that we can join all the nodes together. Choose one node as the "setup coordination node" to run all these commands on. This "setup coordination node" only manages the setup and requires all other nodes to be able to see it and vice versa. *It has no special purpose beyond the setup process; CouchDB does not have the concept of a "master" node in a cluster.*

Setup will not work with unavailable nodes. All nodes must be online and properly preconfigured before the cluster setup process can begin.

To join a node to the cluster, run these commands for each node you want to add:

```
curl -X POST -H "Content-Type: application/json" http://admin:password@<setup-
↪coordination-node>:5984/_cluster_setup -d '{"action": "enable_cluster", "bind_
↪address":"0.0.0.0", "username": "admin", "password":"password", "port": 5984, "node_
↪count": "3", "remote_node": "<remote-node-ip>", "remote_current_user": "<remote-
↪node-username>", "remote_current_password": "<remote-node-password>" }'
curl -X POST -H "Content-Type: application/json" http://admin:password@<setup-
↪coordination-node>:5984/_cluster_setup -d '{"action": "add_node", "host":"<remote-
↪node-ip>", "port": <remote-node-port>, "username": "admin", "password":"password"}'
```

This will join the two nodes together. Keep running the above commands for each node you want to add to the cluster. Once this is done run the following command to complete the cluster setup and add the system databases:

```
curl -X POST -H "Content-Type: application/json" http://admin:password@<setup-
↪coordination-node>:5984/_cluster_setup -d '{"action": "finish_cluster"}'
```

Verify install:

```
curl http://admin:password@<setup-coordination-node>:5984/_cluster_setup
```

Response:

```
{"state":"cluster_finished"}
```

Verify all cluster nodes are connected:

```
curl http://admin:password@<setup-coordination-node>:5984/_membership
```

Response:

```
{
    "all_nodes": [
        "couchdb@couch1.test.com",
        "couchdb@couch2.test.com",
        "couchdb@couch3.test.com",
    ],
    "cluster_nodes": [
        "couchdb@couch1.test.com",
        "couchdb@couch2.test.com",
        "couchdb@couch3.test.com",
    ]
}
```

If the cluster is enabled and `all_nodes` and `cluster_nodes` lists don't match, use curl to add nodes with PUT `/_node/_local/_nodes/couchdb@<reachable-ip-address|fully-qualified-domain-name>` and remove nodes with DELETE `/_node/_local/_nodes/couchdb@<reachable-ip-address|fully-qualified-domain-name>`

You CouchDB cluster is now set up.

CHAPTER

SEVEN

CONFIGURATION

7.1 Introduction To Configuring

7.1.1 Configuration files

By default, CouchDB reads configuration files from the following locations, in the following order:

1. `etc/default.ini`
2. `etc/default.d/*.ini`
3. `etc/local.ini`
4. `etc/local.d/*.ini`

Configuration files in the `*.d/` directories are sorted by name, that means for example a file with the name `etc/local.d/00-shared.ini` is loaded before `etc/local.d/10-server-specific.ini`.

All paths are specified relative to the CouchDB installation directory: `/opt/couchdb` recommended on UNIX-like systems, `C:\CouchDB` recommended on Windows systems, and a combination of two directories on macOS: `Applications/Apache CouchDB.app/Contents/Resources/couchdbx-core/etc` for the `default.ini` and `default.d` directories, and one of `/Users/<your-user>/Library/Application Support/CouchDB2/etc/couchdb` or `/Users/<your-user>/Library/Preferences/couchdb2-local.ini` for the `local.ini` and `local.d` directories.

Settings in successive documents override the settings in earlier entries. For example, setting the *chttpd/bind_address* parameter in `local.ini` would override any setting in `default.ini`.

> **Warning:** The `default.ini` file may be overwritten during an upgrade or re-installation, so localised changes should be made to the `local.ini` file or files within the `local.d` directory.

The configuration file chain may be changed by setting the ERL_FLAGS environment variable:

```
export ERL_FLAGS="-couch_ini /path/to/my/default.ini /path/to/my/local.ini"
```

or by placing the `-couch_ini ..` flag directly in the `etc/vm.args` file. Passing `-couch_ini ..` as a command-line argument when launching `couchdb` is the same as setting the `ERL_FLAGS` environment variable.

> **Warning:** The environment variable/command-line flag overrides any `-couch_ini` option specified in the `etc/vm.args` file. And, **BOTH** of these options **completely** override CouchDB from searching in the default locations. Use these options only when necessary, and be sure to track the contents of `etc/default.ini`, which may change in future releases.

If there is a need to use different `vm.args` or `sys.config` files, for example, in different locations to the ones provided by CouchDB, or you don't want to edit the original files, the default locations may be changed by setting the COUCHDB_ARGS_FILE or COUCHDB_SYSCONFIG_FILE environment variables:

```
export COUCHDB_ARGS_FILE="/path/to/my/vm.args"
export COUCHDB_SYSCONFIG_FILE="/path/to/my/sys.config"
```

7.1.2 Parameter names and values

All parameter names are *case-sensitive*. Every parameter takes a value of one of five types: *boolean*, *integer*, *string*, tuple and proplist. Boolean values can be written as `true` or `false`.

Parameters with value type of *tuple* or *proplist* are following the Erlang requirement for style and naming.

7.1.3 Setting parameters via the configuration file

Changed in version 3.3: added ability to have = in parameter names

Changed in version 3.3: removed the undocumented ability to have multi-line values.

The common way to set some parameters is to edit the `local.ini` file (location explained above).

For example:

```
; This is a comment
[section]
param = value ; inline comments are allowed
```

Each configuration file line may contains *section* definition, *parameter* specification, empty (space and newline characters only) or *commented* line. You can set up *inline* commentaries for *sections* or *parameters*.

The *section* defines group of parameters that are belongs to some specific CouchDB subsystem. For instance, *`httpd`* section holds not only HTTP server parameters, but also others that directly interacts with it.

The *parameter* specification contains two parts divided by the *equal* sign (=): the parameter name on the left side and the parameter value on the right one. The leading and following whitespace for = is an optional to improve configuration readability.

Since version 3.3 it's possible to use = in parameter names, but only when the parameter and value are separated `` = , i.e. the equal sign is surrounded by at least one space on each side. This might be useful in the ``[jwt_keys]` section, where base64 encoded keys may contain some = characters.

The semicolon (`;`) signals the start of a comment. Everything after this character is ignored by CouchDB.

After editing the configuration file, CouchDB should be restarted to apply any changes.

7.1.4 Setting parameters via the HTTP API

Alternatively, configuration parameters can be set via the *HTTP API*. This API allows changing CouchDB configuration on-the-fly without requiring a server restart:

```
curl -X PUT http://adm:pass@localhost:5984/_node/<name@host>/_config/uuids/algorithm -
↪d '"random"'
```

The old parameter's value is returned in the response:

```
"sequential"
```

You should be careful changing configuration via the HTTP API since it's possible to make CouchDB unreachable, for example, by changing the *`chttpd/bind_address`*:

```
curl -X PUT http://adm:pass@localhost:5984/_node/<name@host>/_config/chttpd/bind_
↪address -d '"10.10.0.128"'
```

If you make a typo or the specified IP address is not available from your network, CouchDB will be unreachable. The only way to resolve this will be to remote into the server, correct the config file, and restart CouchDB. To protect yourself against such accidents you may set the *chttpd/config_whitelist* of permitted configuration parameters for updates via the HTTP API. Once this option is set, further changes to non-whitelisted parameters must take place via the configuration file, and in most cases, will also require a server restart before taking effect.

7.1.5 Configuring the local node

While the *HTTP API* allows configuring all nodes in the cluster, as a convenience, you can use the literal string `_local` in place of the node name, to interact with the local node's configuration. For example:

```
curl -X PUT http://adm:pass@localhost:5984/_node/_local/_config/uuids/algorithm -d '
↪"random"'
```

7.2 Base Configuration

7.2.1 Base CouchDB Options

[couchdb]

attachment_stream_buffer_size

Higher values may result in better read performance due to fewer read operations and/or more OS page cache hits. However, they can also increase overall response time for writes when there are many attachment write requests in parallel.

```
[couchdb]
attachment_stream_buffer_size = 4096
```

database_dir

Specifies location of CouchDB database files (`*.couch` named). This location should be writable and readable for the user the CouchDB service runs as (`couchdb` by default).

```
[couchdb]
database_dir = /var/lib/couchdb
```

default_security

Changed in version 3.0: `admin_only` is now the default.

Default security object for databases if not explicitly set. When set to `everyone`, anyone can performs reads and writes. When set to `admin_only`, only admins can read and write. When set to `admin_local`, sharded databases can be read and written by anyone but the shards can only be read and written by admins.

```
[couchdb]
default_security = admin_only
```

enable_database_recovery

Enable this to only "soft-delete" databases when *DELETE /{db}* DELETE requests are made. This will rename all shards of the database with a suffix of the form `<dbname>.YMD.HMS.deleted.couchdb`. You can then manually delete these files later, as desired.

Default is `false`.

```
[couchdb]
enable_database_recovery = false
```

file_compression

Changed in version 1.2: Added Google Snappy compression algorithm.

Method used to compress everything that is appended to database and view index files, except for attachments (see the *attachments* section). Available methods are:

- `none`: no compression
- `snappy`: use Google Snappy, a very fast compressor/decompressor
- `deflate_N`: use zlib's deflate; `N` is the compression level which ranges from `1` (fastest, lowest compression ratio) to `9` (slowest, highest compression ratio)

```
[couchdb]
file_compression = snappy
```

maintenance_mode

A CouchDB node may be put into two distinct maintenance modes by setting this configuration parameter.

- `true`: The node will not respond to clustered requests from other nodes and the /_up endpoint will return a 404 response.
- `nolb`: The /_up endpoint will return a 404 response.
- `false`: The node responds normally, /_up returns a 200 response.

It is expected that the administrator has configured a load balancer in front of the CouchDB nodes in the cluster. This load balancer should use the /_up endpoint to determine whether or not to send HTTP requests to any particular node. For HAProxy, the following config is appropriate:

```
http-check disable-on-404
option httpchk GET /_up
```

max_dbs_open

This option places an upper bound on the number of databases that can be open at once. CouchDB reference counts database accesses internally and will close idle databases as needed. Sometimes it is necessary to keep more than the default open at once, such as in deployments where many databases will be replicating continuously.

```
[couchdb]
max_dbs_open = 100
```

max_document_size

Changed in version 3.0.0.

Limit maximum document body size. Size is calculated based on the serialized Erlang representation of the JSON document body, because that reflects more accurately the amount of storage consumed on disk. In particular, this limit does not include attachments.

HTTP requests which create or update documents will fail with error code 413 if one or more documents is larger than this configuration value.

In case of `_update` handlers, document size is checked after the transformation and right before being inserted into the database.

```
[couchdb]
max_document_size = 8000000 ; bytes
```

> **Warning:** Before version 2.1.0 this setting was implemented by simply checking http request body sizes. For individual document updates via *PUT* that approximation was close enough, however that is not the case for `_bulk_docs` endpoint. After 2.1.0 a separate configuration parameter was

> defined: *chttpd/max_http_request_size*, which can be used to limit maximum http request sizes. After upgrade, it is advisable to review those settings and adjust them accordingly.

os_process_timeout

If an external process, such as a query server or external process, runs for this amount of milliseconds without returning any results, it will be terminated. Keeping this value smaller ensures you get expedient errors, but you may want to tweak it for your specific needs.

```
[couchdb]
os_process_timeout = 5000 ; 5 sec
```

single_node

New in version 3.0.0.

When this configuration setting is set to `true`, automatically create the system databases on startup. Must be set `false` for a clustered CouchDB installation.

uri_file

This file contains the full URI that can be used to access this instance of CouchDB. It is used to help discover the port CouchDB is running on (if it was set to `0` (e.g. automatically assigned any free one). This file should be writable and readable for the user that runs the CouchDB service (`couchdb` by default).

```
[couchdb]
uri_file = /var/run/couchdb/couchdb.uri
```

users_db_security_editable

New in version 3.0.0.

When this configuration setting is set to `false`, reject any attempts to modify the `_users` database security object. Modification of this object is deprecated in 3.x and will be completely disallowed in CouchDB 4.x.

users_db_suffix

Specifies the suffix (last component of a name) of the system database for storing CouchDB users.

```
[couchdb]
users_db_suffix = _users
```

> **Warning:** If you change the database name, do not forget to remove or clean up the old database, since it will no longer be protected by CouchDB.

util_driver_dir

Specifies location of binary drivers (*icu*, *ejson*, etc.). This location and its contents should be readable for the user that runs the CouchDB service.

```
[couchdb]
util_driver_dir = /usr/lib/couchdb/erlang/lib/couch-1.5.0/priv/lib
```

uuid

New in version 1.3.

Unique identifier for this CouchDB cluster.

```
[couchdb]
uuid = 0a959b9b8227188afc2ac26ccdf345a6
```

view_index_dir

Specifies location of CouchDB view index files. This location should be writable and readable for the user that runs the CouchDB service (`couchdb` by default).

```
[couchdb]
view_index_dir = /var/lib/couchdb
```

write_xxhash_checksums

New in version 3.4.

The default value in version 3.4 is `false`. The legacy checksum algorithm will be used for writing couch_file blocks. During reads, both xxHash and the legacy checksum algorithm will be used to verify data integrity. In a future version of CouchDB the default value will become `true`. However, it would still be possible to safely downgrade to version 3.4, which would be able to verify both xxHash and legacy checksums. If CouchDB version downgrade is not a concern, enabling xxHash checksums can result in a measuralbe document read performance, especially for larger document sizes:

```
[couchdb]
write_xxhash_checksums = false
```

7.3 Configuring Clustering

7.3.1 Cluster Options

[cluster]

q

Sets the default number of shards for newly created databases. The default value, `2`, splits a database into 2 separate partitions.

```
[cluster]
q = 2
```

For systems with only a few, heavily accessed, large databases, or for servers with many CPU cores, consider increasing this value to `4` or `8`.

The value of `q` can also be overridden on a per-DB basis, at DB creation time.

See also:

PUT /{db}

n

Sets the number of replicas of each document in a cluster. CouchDB will only place one replica per node in a cluster. When set up through the *Cluster Setup Wizard*, a standalone single node will have `n = 1`, a two node cluster will have `n = 2`, and any larger cluster will have `n = 3`. It is recommended not to set `n` greater than `3`.

```
[cluster]
n = 3
```

placement

> **Warning:** Use of this option will **override** the `n` option for replica cardinality. Use with care.

Sets the cluster-wide replica placement policy when creating new databases. The value must be a comma-delimited list of strings of the format `zone_name:#`, where `zone_name` is a zone as specified in the `nodes` database and `#` is an integer indicating the number of replicas to place on nodes with a matching `zone_name`.

This parameter is not specified by default.

```
[cluster]
placement = metro-dc-a:2,metro-dc-b:1
```

See also:

Placing a database on specific nodes

`seedlist`

An optional, comma-delimited list of node names that this node should contact in order to join a cluster. If a seedlist is configured the `_up` endpoint will return a 404 until the node has successfully contacted at least one of the members of the seedlist and replicated an up-to-date copy of the `_nodes`, `_dbs`, and `_users` system databases.

[cluster] seedlist = couchdb@node1.example.com,couchdb@node2.example.com

`reconnect_interval_sec`

New in version 3.3.

Period in seconds specifying how often to attempt reconnecting to disconnected nodes. There is a 25% random jitter applied to this value.

7.3.2 RPC Performance Tuning

`[rexi]`

CouchDB uses distributed Erlang to communicate between nodes in a cluster. The `rexi` library provides an optimized RPC mechanism over this communication channel. There are a few configuration knobs for this system, although in general the defaults work well.

`buffer_count`

The local RPC server will buffer messages if a remote node goes unavailable. This flag determines how many messages will be buffered before the local server starts dropping messages. Default value is `2000`.

`server_per_node`

By default, rexi will spawn one local gen_server process for each node in the cluster. Disabling this flag will cause CouchDB to use a single process for all RPC communication, which is not recommended in high throughput deployments.

`stream_limit`

New in version 3.0.

This flag comes into play during streaming operations like views and change feeds. It controls how many messages a remote worker process can send to a coordinator without waiting for an acknowledgement from the coordinator process. If this value is too large the coordinator can become overwhelmed by messages from the worker processes and actually deliver lower overall throughput to the client. In CouchDB 2.x this value was hard-coded to `10`. In the 3.x series it is configurable and defaults to `5`. Databases with a high `q` value are especially sensitive to this setting.

7.4 Database Per User

7.4.1 Database Per User Options

[couch_peruser]

enable

If set to `true`, couch_peruser ensures that a private per-user database exists for each document in `_users`. These databases are writable only by the corresponding user. Database names are in the following form: `userdb-{UTF-8 hex encoded username}`.

```
[couch_peruser]
enable = false
```

Note: The `_users` database must exist before couch_peruser can be enabled.

Tip: Under NodeJS, user names can be converted to and from database names thusly:

```
function dbNameToUsername(prefixedHexName) {
  return Buffer.from(prefixedHexName.replace('userdb-', ''), 'hex').toString(
↪'utf8');
}

function usernameToDbName(name) {
  return 'userdb-' + Buffer.from(name).toString('hex');
}
```

delete_dbs

If set to `true` and a user is deleted, the respective database gets deleted as well.

```
[couch_peruser]
delete_dbs = false
```

Note: When using JWT authorization, the provided token must include a custom `_couchdb.roles=['_admin']` claim to for the peruser database to be properly created and accessible for the user provided in the `sub=` claim.

q

If set, specify the sharding value for per-user databases. If unset, the cluster default value will be used.

```
[couch_peruser]
q = 1
```

7.5 Disk Monitor Configuration

Apache CouchDB can react proactively when disk space gets low.

7.5.1 Disk Monitor Options

[disk_monitor]

New in version 3.4:

background_view_indexing_threshold

The percentage of used disk space on the `view_index_dir` above which CouchDB will no longer start background view indexing jobs. Defaults to `80`.

```
[disk_monitor]
background_view_indexing_threshold = 80
```

interactive_database_writes_threshold

The percentage of used disk space on the `database_dir` above which CouchDB will no longer allow interactive document updates (writes or deletes).

Replicated updates and database deletions are still permitted.

In a clustered write an error will be returned if enough nodes are above the `interactive_database_writes_threshold`.

Defaults to `90`.

```
[disk_monitor]
interactive_database_writes_threshold = 90
```

enable

Enable disk monitoring subsystem. Defaults to `false`.

```
[disk_monitor]
enable = false
```

interactive_view_indexing_threshold

The percentage of used disk space on the `view_index_dir` above which CouchDB will no longer update stale view indexes when queried.

View indexes that are already up to date can still be queried, and stale view indexes can be queried if either `stale=ok` or `update=false` are set.

Attempts to query a stale index without either parameter will yield a `507 Insufficient Storage` error. Defaults to `90`.

```
[disk_monitor]
interactive_view_indexing_threshold = 90
```

7.6 CouchDB HTTP Server

7.6.1 HTTP Server Options

[chttpd]

Note: In CouchDB 2.x and 3.x, the *chttpd* section refers to the standard, clustered port. All use of CouchDB, aside from a few specific maintenance tasks as described in this documentation, should be performed over this port.

bind_address

Defines the IP address by which the clustered port is available:

```
[chttpd]
bind_address = 127.0.0.1
```

To let CouchDB listen any available IP address, use `0.0.0.0`:

```
[chttpd]
bind_address = 0.0.0.0
```

For IPv6 support you need to set `::1` if you want to let CouchDB listen correctly:

```
[chttpd]
bind_address = ::1
```

or `::` for any available:

```
[chttpd]
bind_address = ::
```

port

Defines the port number to listen:

```
[chttpd]
port = 5984
```

To let CouchDB use any free port, set this option to `0`:

```
[chttpd]
port = 0
```

prefer_minimal

If a request has the header `"Prefer": "return=minimal"`, CouchDB will only send the headers that are listed for the `prefer_minimal` configuration.:

```
[chttpd]
prefer_minimal = Cache-Control, Content-Length, Content-Range, Content-Type,␣
↪ETag, Server, Transfer-Encoding, Vary
```

Warning: Removing the Server header from the settings will mean that the CouchDB server header is replaced with the MochiWeb server header.

authentication_handlers

List of authentication handlers used by CouchDB. You may extend them via third-party plugins or remove some of them if you won't let users to use one of provided methods:

```
[chttpd]
authentication_handlers = {chttpd_auth, cookie_authentication_handler},
↪{chttpd_auth, default_authentication_handler}
```

- `{chttpd_auth, cookie_authentication_handler}`: used for Cookie auth;
- `{chttpd_auth, proxy_authentication_handler}`: used for Proxy auth;
- `{chttpd_auth, jwt_authentication_handler}`: used for JWT auth;
- `{chttpd_auth, default_authentication_handler}`: used for Basic auth;
- `{couch_httpd_auth, null_authentication_handler}`: disables auth, breaks CouchDB.

buffer_response

Changed in version 3.1.1.

Set this to `true` to delay the start of a response until the end has been calculated. This increases memory usage, but simplifies client error handling as it eliminates the possibility that a response may be deliberately terminated midway through, due to a timeout. This config value may be changed at runtime, without impacting any in-flight responses.

Even if this is set to `false` (the default), buffered responses can be enabled on a per-request basis for any delayed JSON response call by adding `?buffer_response=true` to the request's parameters.

allow_jsonp

Changed in version 3.2: moved from [httpd] to [chttpd] section

The `true` value of this option enables JSONP support (it's `false` by default):

```
[chttpd]
allow_jsonp = false
```

changes_timeout

Changed in version 3.2: moved from [httpd] to [chttpd] section

Specifies default *timeout* value for *Changes Feed* in milliseconds (60000 by default):

```
[chttpd]
changes_timeout = 60000 ; 60 seconds
```

config_whitelist

Changed in version 3.2: moved from [httpd] to [chttpd] section

Sets the configuration modification whitelist. Only whitelisted values may be changed via the *config API*. To allow the admin to change this value over HTTP, remember to include `{chttpd, config_whitelist}` itself. Excluding it from the list would require editing this file to update the whitelist:

```
[chttpd]
config_whitelist = [{chttpd,config_whitelist}, {log,level}, {etc,etc}]
```

enable_cors

New in version 1.3.

Changed in version 3.2: moved from [httpd] to [chttpd] section

Controls *CORS* feature:

```
[chttpd]
enable_cors = false
```

secure_rewrites

Changed in version 3.2: moved from [httpd] to [chttpd] section

This option allow to isolate databases via subdomains:

```
[chttpd]
secure_rewrites = true
```

x_forwarded_host

Changed in version 3.2: moved from [httpd] to [chttpd] section

The *x_forwarded_host* header (`X-Forwarded-Host` by default) is used to forward the original value of the `Host` header field in case, for example, if a reverse proxy is rewriting the "Host" header field to some internal host name before forward the request to CouchDB:

```
[chttpd]
x_forwarded_host = X-Forwarded-Host
```

This header has higher priority above `Host` one, if only it exists in the request.

x_forwarded_proto

Changed in version 3.2: moved from [httpd] to [chttpd] section

x_forwarded_proto header (`X-Forwarder-Proto` by default) is used for identifying the originating protocol of an HTTP request, since a reverse proxy may communicate with CouchDB instance using HTTP even if the request to the reverse proxy is HTTPS:

```
[chttpd]
x_forwarded_proto = X-Forwarded-Proto
```

x_forwarded_ssl

Changed in version 3.2: moved from [httpd] to [chttpd] section

The *x_forwarded_ssl* header (`X-Forwarded-Ssl` by default) tells CouchDB that it should use the *https* scheme instead of the *http*. Actually, it's a synonym for `X-Forwarded-Proto: https` header, but used by some reverse proxies:

```
[chttpd]
x_forwarded_ssl = X-Forwarded-Ssl
```

enable_xframe_options

Changed in version 3.2: moved from [httpd] to [chttpd] section

Controls *Enables or disabled* feature:

```
[chttpd]
enable_xframe_options = false
```

max_http_request_size

Changed in version 3.2: moved from [httpd] to [chttpd] section

Limit the maximum size of the HTTP request body. This setting applies to all requests and it doesn't discriminate between single vs. multi-document operations. So setting it to 1MB would block a *PUT* of a document larger than 1MB, but it might also block a `_bulk_docs` update of 1000 1KB documents, or a multipart/related update of a small document followed by two 512KB attachments. This setting is intended to be used as a protection against maliciously large HTTP requests rather than for limiting maximum document sizes.

```
[chttpd]
max_http_request_size = 4294967296 ; 4 GB
```

> **Warning:** Before version 2.1.0 *couchdb/max_document_size* was implemented effectively as `max_http_request_size`. That is, it checked HTTP request bodies instead of document sizes. After the upgrade, it is advisable to review the usage of these configuration settings.

bulk_get_use_batches

New in version 3.3.

Set to false to revert to a previous `_bulk_get` implementation using single doc fetches internally. Using batches should be faster, however there may be bugs in the new new implemention, so expose this option to allow reverting to the old behavior.

```
[chttpd]
bulk_get_use_batches = true
```

admin_only_all_dbs

New in version 2.2: implemented for `_all_dbs` defaulting to `false`

Changed in version 3.0: default switched to `true`, applies to `_all_dbs`

Changed in version 3.3: applies for `_all_dbs` and `_dbs_info`

When set to `true` admin is required to access `_all_dbs` and `_dbs_info`.

```
[chttpd]
admin_only_all_dbs = true
```

server_header_versions

New in version 3.4.

Set to false to remove the CouchDB and Erlang/OTP versions from the Server response header.

```
[chttpd]
server_header_versions = true
```

disconnect_check_msec

New in version 3.4.

How often, in milliseconds, to check for client disconnects while processing streaming requests such as _all_docs, _find, _changes and views.

```
[chttpd]
disconnect_check_msec = 30000
```

disconnect_check_jitter_msec

New in version 3.4.

How much random jitter to apply to the `disconnect_check_msec` period. This is to avoid stampede in case of a large number of concurrent clients.

```
[chttpd]
disconnect_check_jitter_msec = 15000
```

[httpd]

Changed in version 3.2: These options were moved to [chttpd] section: *allow_jsonp*, *changes_timeout*, *config_whitelist*, *enable_cors*, *secure_rewrites*, *x_forwarded_host*, *x_forwarded_proto*, *x_forwarded_ssl*, *enable_xframe_options*, *max_http_request_size*.

server_options

Server options for the MochiWeb component of CouchDB can be added to the configuration files:

```
[httpd]
server_options = [{backlog, 128}, {acceptor_pool_size, 16}]
```

The options supported are a subset of full options supported by the TCP/IP stack. A list of the supported options are provided in the Erlang inet documentation.

socket_options

The socket options for the listening socket in CouchDB, as set at the beginning of ever request, can be specified as a list of tuples. For example:

```
[httpd]
socket_options = [{sndbuf, 262144}]
```

The options supported are a subset of full options supported by the TCP/IP stack. A list of the supported options are provided in the Erlang inet documentation.

7.6.2 HTTPS (SSL/TLS) Options

[ssl]

CouchDB supports TLS/SSL natively, without the use of a proxy server.

HTTPS setup can be tricky, but the configuration in CouchDB was designed to be as easy as possible. All you need is two files; a certificate and a private key. If you have an official certificate from a certificate authority, both should be in your possession already.

If you just want to try this out and don't want to go through the hassle of obtaining an official certificate, you can create a self-signed certificate. Everything will work the same, but clients will get a warning about an insecure certificate.

You will need the OpenSSL command line tool installed. It probably already is.

```
shell> mkdir /etc/couchdb/cert
shell> cd /etc/couchdb/cert
shell> openssl genrsa > privkey.pem
shell> openssl req -new -x509 -key privkey.pem -out couchdb.pem -days 1095
shell> chmod 600 privkey.pem couchdb.pem
shell> chown couchdb privkey.pem couchdb.pem
```

Now, you need to edit CouchDB's configuration, by editing your `local.ini` file. Here is what you need to do.

Under the `[ssl]` section, enable HTTPS and set up the newly generated certificates:

```
[ssl]
enable = true
cert_file = /etc/couchdb/cert/couchdb.pem
key_file = /etc/couchdb/cert/privkey.pem
```

For more information please read certificates HOWTO.

Now start (or restart) CouchDB. You should be able to connect to it using HTTPS on port 6984:

```
shell> curl https://127.0.0.1:6984/
curl: (60) SSL certificate problem, verify that the CA cert is OK. Details:
error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
More details here: http://curl.haxx.se/docs/sslcerts.html
```

(continues on next page)

(continued from previous page)

```
curl performs SSL certificate verification by default, using a "bundle"
of Certificate Authority (CA) public keys (CA certs). If the default
bundle file isn't adequate, you can specify an alternate file
using the --cacert option.
If this HTTPS server uses a certificate signed by a CA represented in
the bundle, the certificate verification probably failed due to a
problem with the certificate (it might be expired, or the name might
not match the domain name in the URL).
If you'd like to turn off curl's verification of the certificate, use
the -k (or --insecure) option.
```

Oh no! What happened?! Remember, clients will notify their users that your certificate is self signed. `curl` is the client in this case and it notifies you. Luckily you trust yourself (don't you?) and you can specify the `-k` option as the message reads:

```
shell> curl -k https://127.0.0.1:6984/
{"couchdb":"Welcome","version":"1.5.0"}
```

All done.

For performance reasons, and for ease of setup, you may still wish to terminate HTTPS connections at your load balancer / reverse proxy, then use unencrypted HTTP between it and your CouchDB cluster. This is a recommended approach.

Additional detail may be available in the CouchDB wiki.

cacert_file

The path to a file containing PEM encoded CA certificates. The CA certificates are used to build the server certificate chain, and for client authentication. Also the CAs are used in the list of acceptable client CAs passed to the client when a certificate is requested. May be omitted if there is no need to verify the client and if there are not any intermediate CAs for the server certificate:

```
[ssl]
cacert_file = /etc/ssl/certs/ca-certificates.crt
```

cert_file

Path to a file containing the user's certificate:

```
[ssl]
cert_file = /etc/couchdb/cert/couchdb.pem
```

key_file

Path to file containing user's private PEM encoded key:

```
[ssl]
key_file = /etc/couchdb/cert/privkey.pem
```

password

String containing the user's password. Only used if the private key file is password protected:

```
[ssl]
password = somepassword
```

ssl_certificate_max_depth

Maximum peer certificate depth (must be set even if certificate validation is off):

```
[ssl]
ssl_certificate_max_depth = 1
```

verify_fun

The verification fun (optional) if not specified, the default verification fun will be used:

```
[ssl]
verify_fun = {Module, VerifyFun}
```

verify_ssl_certificates

Set to `true` to validate peer certificates:

```
[ssl]
verify_ssl_certificates = false
```

fail_if_no_peer_cert

Set to `true` to terminate the TLS/SSL handshake with a `handshake_failure` alert message if the client does not send a certificate. Only used if `verify_ssl_certificates` is `true`. If set to `false` it will only fail if the client sends an invalid certificate (an empty certificate is considered valid):

```
[ssl]
fail_if_no_peer_cert = false
```

secure_renegotiate

Set to `true` to reject renegotiation attempt that does not live up to RFC 5746:

```
[ssl]
secure_renegotiate = true
```

ciphers

Set to the cipher suites that should be supported which can be specified in erlang format "{ecdhe_ecdsa,aes_128_cbc,sha256}" or in OpenSSL format "ECDHE-ECDSA-AES128-SHA256".

```
[ssl]
ciphers = ["ECDHE-ECDSA-AES128-SHA256", "ECDHE-ECDSA-AES128-SHA"]
```

tls_versions

Set to a list of permitted SSL/TLS protocol versions:

```
[ssl]
tls_versions = [tlsv1 | 'tlsv1.1' | 'tlsv1.2']
```

7.6.3 Cross-Origin Resource Sharing

[cors]

New in version 1.3: added CORS support, see JIRA COUCHDB-431

Changed in version 3.2: moved from [httpd] to [chttpd] section

CORS, or "Cross-Origin Resource Sharing", allows a resource such as a web page running JavaScript inside a browser, to make AJAX requests (XMLHttpRequests) to a different domain, without compromising the security of either party.

A typical use case is to have a static website hosted on a CDN make requests to another resource, such as a hosted CouchDB instance. This avoids needing an intermediary proxy, using *JSONP* or similar workarounds to retrieve and host content.

While CouchDB's integrated HTTP server has support for document attachments makes this less of a constraint for pure CouchDB projects, there are many cases where separating the static content from the database access is desirable, and CORS makes this very straightforward.

By supporting CORS functionality, a CouchDB instance can accept direct connections to protected databases and instances, without the browser functionality being blocked due to same-origin constraints. CORS is supported today on over 90% of recent browsers.

CORS support is provided as experimental functionality in 1.3, and as such will need to be enabled specifically in CouchDB's configuration. While all origins are forbidden from making requests by default, support is available for simple requests, preflight requests and per-vhost configuration.

This section requires *chttpd/enable_cors* option have `true` value:

```
[chttpd]
enable_cors = true
```

credentials

By default, neither authentication headers nor cookies are included in requests and responses. To do so requires both setting `XmlHttpRequest.withCredentials = true` on the request object in the browser and enabling credentials support in CouchDB.

```
[cors]
credentials = true
```

CouchDB will respond to a credentials-enabled CORS request with an additional header, `Access-Control-Allow-Credentials=true`.

origins

List of origins separated by a comma, `*` means accept all. You can't set `origins = *` and `credentials = true` option at the same time:

```
[cors]
origins = *
```

Access can be restricted by protocol, host and optionally by port. Origins must follow the scheme: http://example.com:80:

```
[cors]
origins = http://localhost, https://localhost, http://couch.mydev.name:8080
```

Note that by default, no origins are accepted. You must define them explicitly.

headers

List of accepted headers separated by a comma:

```
[cors]
headers = X-Couch-Id, X-Couch-Rev
```

methods

List of accepted methods:

```
[cors]
methods = GET,POST
```

max_age

Sets the `Access-Control-Max-Age` header in seconds. Use it to avoid repeated `OPTIONS` requests.

[cors] max_age = 3600

See also:

Original JIRA implementation ticket

Standards and References:

- IETF RFCs relating to methods: **RFC 2618**, **RFC 2817**, **RFC 5789**

- IETF RFC for Web Origins: **RFC 6454**
- W3C CORS standard

Mozilla Developer Network Resources:

- Same origin policy for URIs
- HTTP Access Control
- Server-side Access Control
- JavaScript same origin policy

Client-side CORS support and usage:

- CORS browser support matrix
- COS tutorial
- XHR with CORS

Per Virtual Host Configuration

> **Warning:** Virtual Hosts are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

To set the options for a `vhosts`, you will need to create a section with the vhost name prefixed by `cors:`. Example case for the vhost *example.com*:

```
[cors:example.com]
credentials = false
; List of origins separated by a comma
origins = *
; List of accepted headers separated by a comma
headers = X-CouchDB-Header
; List of accepted methods
methods = HEAD, GET
```

A video from 2010 on vhost and rewrite configuration is available, but is not guaranteed to match current syntax or behaviour.

7.6.4 Virtual Hosts

> **Warning:** Virtual Hosts are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

[vhosts]

CouchDB can map requests to different locations based on the `Host` header, even if they arrive on the same inbound IP address.

This allows different virtual hosts on the same machine to map to different databases or design documents, etc. The most common use case is to map a virtual host to a *Rewrite Handler*, to provide full control over the application's URIs.

To add a virtual host, add a *CNAME* pointer to the DNS for your domain name. For development and testing, it is sufficient to add an entry in the hosts file, typically */etc/hosts* ` on Unix-like operating systems:

```
# CouchDB vhost definitions, refer to local.ini for further details
127.0.0.1       couchdb.local
```

Test that this is working:

```
$ ping -n 2 couchdb.local
PING couchdb.local (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_req=1 ttl=64 time=0.025 ms
64 bytes from localhost (127.0.0.1): icmp_req=2 ttl=64 time=0.051 ms
```

Finally, add an entry to your *configuration file* in the `[vhosts]` section:

```
[vhosts]
couchdb.local:5984 = /example
*.couchdb.local:5984 = /example
```

If your CouchDB is listening on the default HTTP port (80), or is sitting behind a proxy, then you don't need to specify a port number in the `vhost` key.

The first line will rewrite the request to display the content of the *example* database. This rule works only if the `Host` header is `couchdb.local` and won't work for *CNAMEs*. The second rule, on the other hand, matches all *CNAMEs* to *example* db, so that both *www.couchdb.local* and *db.couchdb.local* will work.

Rewriting Hosts to a Path

Like in the *_rewrite* handler you can match some variable and use them to create the target path. Some examples:

```
[vhosts]
*.couchdb.local = /*
:dbname. = /:dbname
:ddocname.:dbname.example.com = /:dbname/_design/:ddocname/_rewrite
```

The first rule passes the wildcard as `dbname`. The second one does the same, but uses a variable name. And the third one allows you to use any URL with `ddocname` in any database with `dbname`.

7.6.5 X-Frame-Options

X-Frame-Options is a response header that controls whether a http response can be embedded in a <frame>, <iframe> or <object>. This is a security feature to help against clickjacking.

> [x_frame_options] ; Settings same-origin will return X-Frame-Options: SAMEORIGIN. ; If same origin is set, it will ignore the hosts setting ; same_origin = true ; Settings hosts will ; return X-Frame-Options: ALLOW-FROM https://example.com/ ; List of hosts separated by a comma. * means accept all ; hosts =

If xframe_options is enabled it will return `X-Frame-Options: DENY` by default. If `same_origin` is enabled it will return `X-Frame-Options: SAMEORIGIN`. A `X-FRAME-OPTIONS: ALLOW-FROM url` will be returned when `same_origin` is false, and the HOST header matches one of the urls in the `hosts` config. Otherwise a `X-Frame-Options: DENY` will be returned.

7.7 Authentication and Authorization

7.7.1 Server Administrators

`[admins]`

Changed in version 3.0.0: CouchDB requires an admin account to start. If an admin account has not been created, CouchDB will print an error message and terminate.

CouchDB server administrators and passwords are not stored in the `_users` database, but in the last `[admins]` section that CouchDB finds when loading its ini files. See :config:intro for details on config file order and behaviour. This file (which could be something like `/opt/couchdb/etc/local.ini` or `/opt/couchdb/etc/local.d/10-admins.ini` when CouchDB is installed from packages) should be appropriately secured and readable only by system administrators:

```
[admins]
;admin = mysecretpassword
admin = -hashed-6d3c30241ba0aaa4e16c6ea99224f915687ed8cd,
↪7f4a3e05e0cbc6f48a0035e3508eef90
architect = -pbkdf2-43ecbd256a70a3a2f7de40d2374b6c3002918834,
↪921a12f74df0c1052b3e562a23cd227f,10000
```

Administrators can be added directly to the `[admins]` section, and when CouchDB is restarted, the passwords will be salted and encrypted. You may also use the HTTP interface to create administrator accounts; this way, you don't need to restart CouchDB, and there's no need to temporarily store or transmit passwords in plaintext. The HTTP `/_node/{node-name}/_config/admins` endpoint supports querying, deleting or creating new admin accounts:

```
GET /_node/nonode@nohost/_config/admins HTTP/1.1
Accept: application/json
Host: localhost:5984
```

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 196
Content-Type: application/json
Date: Fri, 30 Nov 2012 11:37:18 GMT
Server: CouchDB (Erlang/OTP)
```

```
{
    "admin": "-hashed-6d3c30241ba0aaa4e16c6ea99224f915687ed8cd,
↪7f4a3e05e0cbc6f48a0035e3508eef90",
    "architect": "-pbkdf2-43ecbd256a70a3a2f7de40d2374b6c3002918834,
↪921a12f74df0c1052b3e562a23cd227f,10000"
}
```

If you already have a salted, encrypted password string (for example, from an old ini file, or from a different CouchDB server), then you can store the "raw" encrypted string, without having CouchDB doubly encrypt it.

```
PUT /_node/nonode@nohost/_config/admins/architect?raw=true HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 89
Host: localhost:5984

"-pbkdf2-43ecbd256a70a3a2f7de40d2374b6c3002918834,921a12f74df0c1052b3e562a23cd227f,
↪10000"
```

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 89
Content-Type: application/json
Date: Fri, 30 Nov 2012 11:39:18 GMT
Server: CouchDB (Erlang/OTP)

"-pbkdf2-43ecbd256a70a3a2f7de40d2374b6c3002918834,921a12f74df0c1052b3e562a23cd227f,
↪10000"
```

Further details are available in *security*, including configuring the work factor for PBKDF2, and the algorithm itself at PBKDF2 (RFC-2898).

Changed in version 1.4: *PBKDF2* server-side hashed salted password support added, now as a synchronous call for the _config/admins API.

7.7.2 Authentication Configuration

[chttpd]

require_valid_user

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd] section

When this option is set to true, no requests are allowed from anonymous users. Everyone must be authenticated.

```
[chttpd]
require_valid_user = false
```

require_valid_user_except_for_up

When this option is set to true, no requests are allowed from anonymous users, *except* for the /_up endpoint. Everyone else must be authenticated.

```
[chttpd]
require_valid_user_except_for_up = false
```

[chttpd_auth]

Changed in version 3.2: These options were moved to [chttpd_auth] section: *authentication_redirect*, *timeout*, *auth_cache_size*, *allow_persistent_cookies*, *iterations*, *min_iterations*, *max_iterations*, *secret*, *users_db_public*, *x_auth_roles*, *x_auth_token*, *x_auth_username*, *cookie_domain*, *same_site*.

allow_persistent_cookies

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

When set to true, CouchDB will set the Max-Age and Expires attributes on the cookie, which causes user agents (like browsers) to preserve the cookie over restarts.

```
[chttpd_auth]
allow_persistent_cookies = true
```

cookie_domain

New in version 2.1.1.

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

Configures the domain attribute of the AuthSession cookie. By default the domain attribute is empty, resulting in the cookie being set on CouchDB's domain.

```
[chttpd_auth]
cookie_domain = example.com
```

same_site

New in version 3.0.0.

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

When this option is set to a non-empty value, a SameSite attribute is added to the AuthSession cookie. Valid values are none, lax or strict.:

```
[chttpd_auth]
same_site = strict
```

auth_cache_size

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

Number of *User Context Object* to cache in memory, to reduce disk lookups.

```
[chttpd_auth]
auth_cache_size = 50
```

authentication_redirect

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

Specifies the location for redirection on successful authentication if a `text/html` response is accepted by the client (via an `Accept` header).

```
[chttpd_auth]
authentication_redirect = /_utils/session.html
```

hash_algorithms

New in version 3.3.

Note: Until CouchDB version 3.3.1, *Proxy Authentication* used only the hash algorithm `sha1` as validation of *`X-Auth-CouchDB-Token`*.

Sets the HMAC hash algorithm used for cookie and proxy authentication. You can provide a comma-separated list of hash algorithms. New cookie sessions or session updates are calculated with the first hash algorithm. All values in the list can be used to decode the cookie session and the token *`X-Auth-CouchDB-Token`* for *Proxy Authentication*.

```
[chttpd_auth]
hash_algorithms = sha256, sha
```

Note: You can select any hash algorithm the version of erlang used in your CouchDB install supports. The common list of available hashes might be:

```
sha, sha224, sha256, sha384, sha512
```

To retrieve a complete list of supported hash algorithms you can use our `bin/remsh` script and retrieve a full list of available hash algorithms with `crypto:supports(hashs).` or use the *_node/{node-name}/_versions* endpoint to retrieve the hashes.

Warning: We do not recommend using the following hash algorithms:

```
md4, md5
```

iterations

New in version 1.3.

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

The number of iterations for password hashing by the PBKDF2 algorithm. A higher number provides better hash durability, but comes at a cost in performance for each request that requires authentication. When using hundreds of thousands of iterations, use session cookies, or the performance hit will be huge. (The internal hashing algorithm is SHA1, which affects the recommended number of iterations.)

```
[chttpd_auth]
iterations = 10000
```

min_iterations

New in version 1.6.

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

The minimum number of iterations allowed for passwords hashed by the PBKDF2 algorithm. Any user with fewer iterations is forbidden.

```
[chttpd_auth]
min_iterations = 100
```

max_iterations

New in version 1.6.

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

The maximum number of iterations allowed for passwords hashed by the PBKDF2 algorithm. Any user with greater iterations is forbidden.

```
[chttpd_auth]
max_iterations = 100000
```

password_regexp

New in version 3.2.

A list of Regular Expressions to check new/changed passwords. When set, new user passwords must **match** all RegExp in this list.

A RegExp can be paired with a *reason text*: `[{"RegExp", "reason text"}, ...]`. If a RegExp doesn't match, its *reason text* will be appended to the default reason of `Password does not conform to requirements.`

```
[couch_httpd_auth]
; Password must be 10 chars long and have one or more uppercase and
; lowercase char and one or more numbers.
password_regexp = [{".{10,}", "Min length is 10 chars."}, "[A-Z]+", "[a-z]+",
↪ "\\d+"]
```

proxy_use_secret

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

When this option is set to `true`, the *`chttpd_auth/secret`* option is required for *Proxy Authentication*.

```
[chttpd_auth]
proxy_use_secret = false
```

public_fields

New in version 1.4.

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

A comma-separated list of field names in user documents (in *`couchdb/users_db_suffix`*) that can be read by any user. If unset or not specified, authenticated users can only retrieve their own document.

```
[chttpd_auth]
public_fields = first_name, last_name, contacts, url
```

Note: Using the `public_fields` allowlist for user document properties requires setting the *`chttpd_auth/users_db_public`* option to `true` (the latter option has no other purpose):

```
[chttpd_auth]
users_db_public = true
```

---

secret

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

The secret token is used for *Proxy Authentication* and for *Cookie Authentication*.

```
[chttpd_auth]
secret = 92de07df7e7a3fe14808cef90a7cc0d91
```

timeout

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

Number of seconds since the last request before sessions will be expired.

```
[chttpd_auth]
timeout = 600
```

users_db_public

New in version 1.4.

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

Allow all users to view user documents. By default, only admins may browse all users documents, while users may browse only their own document.

```
[chttpd_auth]
users_db_public = false
```

x_auth_roles

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

The HTTP header name (`X-Auth-CouchDB-Roles` by default) that contains the list of a user's roles, separated by a comma. Used for *Proxy Authentication*.

```
[chttpd_auth]
x_auth_roles = X-Auth-CouchDB-Roles
```

x_auth_token

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

The HTTP header name (`X-Auth-CouchDB-Token` by default) containing the token used to authenticate the authorization. This token is an *HMAC-SHA1* created from the `chttpd_auth/secret` and `chttpd_auth/x_auth_username`. The secret key should be the same on the client and the CouchDB node. This token is optional if the value of the `chttpd_auth/proxy_use_secret` option is not `true`. Used for *Proxy Authentication*.

```
[chttpd_auth]
x_auth_token = X-Auth-CouchDB-Token
```

x_auth_username

Changed in version 3.2: moved from [couch_httpd_auth] to [chttpd_auth] section

The HTTP header name (`X-Auth-CouchDB-UserName` by default) containing the username. Used for *Proxy Authentication*.

```
[chttpd_auth]
x_auth_username = X-Auth-CouchDB-UserName
```

[jwt_auth]

required_claims

This parameter is a comma-separated list of additional mandatory JWT claims that must be present in any presented JWT token. A 404 Not Found is sent if any are missing.

```
[jwt_auth]
    required_claims = exp,iat
```

roles_claim_name

> **Warning:** `roles_claim_name` is deprecated in CouchDB 3.3, and will be removed later. Please migrate to `roles_claim_path`.

If presented, it is used as the CouchDB user's roles list as long as the JWT token is valid. The default value for `roles_claim_name` is `_couchdb.roles`.

> **Note:** Values for `roles_claim_name` can only be top-level attributes in the JWT token. If `roles_claim_path` is set, then `roles_claim_name` is ignored!

Let's assume, we have the following configuration:

```
[jwt_auth]
roles_claim_name = my-couchdb.roles
```

CouchDB will search for the attribute `my-couchdb.roles` in the JWT token.

```
{
    "my-couchdb.roles": [
        "role_1",
        "role_2"
    ]
}
```

roles_claim_path

New in version 3.3.

This parameter was introduced to overcome disadvantages of `roles_claim_name`, because it is not possible with `roles_claim_name` to map nested role attributes in the JWT token.

> **Note:** If `roles_claim_path` is set, then `roles_claim_name` is ignored!

Now it is possible the read a nested roles claim from JWT tokens into CouchDB. As always, there is some theory at the beginning to get things up and running. Don't get scared now, it's really short and easy. Honestly!

There are only two characters with a special meaning. These are

- `.` for nesting json attributes and
- `\.` to skip nesting

That's it. Really.

Let's assume there is the following data-payload in the JWT token:

```
{
    "resource_access": {
        "security.settings": {
            "account": {
                "roles": [
                    "manage-account",
                    "view-profile"
                ]
            }
        }
    }
}
```

Now, let's define the config variable `roles_claim_path` for this example. It should look like this:

```
roles_claim_path = resource_access.security\.settings.account.roles
```

If an attribute has a `.` in the key like `security.settings`, you have to escape it in the config parameter with `\.`. If you use a `.` then it gets interpreted as a nested sub-key. Let's illustrate the behavior with a second example. There is the following config parameter for `roles_claim_name` (by the way it was the default value if you didn't configured it):

```
roles_claim_name = _couchdb.roles
```

Note: CouchDB doesn't set any default or implicit value for `roles_claim_path`.

To migrate from `roles_claim_name` to `roles_claim_path` you need to change the parameter name and escape the `.` to prevent CouchDB to read this as a nested JSON key.

```
roles_claim_path = _couchdb\.roles
```

Let's assume your JWT token have the following data-payload for your couchdb roles claim:

```
{
    "_couchdb.roles": [
        "accounting-role",
        "view-role"
    ]
}
```

If you did everything right, the response from the `_session` endpoint should look something like this:

```
GET /_session HTTP/1.1
Host: localhost:5984
Authorization: Bearer <JWT token>
```

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
    "ok": true,
    "userCtx": {
        "name": "1234567890",
        "roles": [
            "accounting-role",
            "view-role"
```

(continues on next page)

(continued from previous page)

```
        ]
    },
    "info": {
    "authentication_handlers": [
        "jwt",
        "proxy",
        "cookie",
        "default"
    ],
    "authenticated": "jwt"
    }
}
```

That's all, you are done with the migration from `roles_claim_name` to `roles_claim_path` Easy, isn't it?

7.8 Compaction

7.8.1 Database Compaction Options

[database_compaction]

doc_buffer_size

Specifies the copy buffer's maximum size in bytes:

```
[database_compaction]
doc_buffer_size = 524288
```

checkpoint_after

Triggers a checkpoint after the specified amount of bytes were successfully copied to the compacted database:

```
[database_compaction]
checkpoint_after = 5242880
```

7.8.2 View Compaction Options

[view_compaction]

keyvalue_buffer_size

Specifies maximum copy buffer size in bytes used during compaction:

```
[view_compaction]
keyvalue_buffer_size = 2097152
```

7.8.3 Compaction Daemon

CouchDB ships with an automated, event-driven daemon internally known as "smoosh" that continuously re-prioritizes the database and secondary index files on each node and automatically compacts the files that will recover the most free space according to the following parameters.

`[smoosh]`

`db_channels`
: A comma-delimited list of channels that are sent the names of database files when those files are updated. Each channel can choose whether to enqueue the database for compaction; once a channel has enqueued the database, no additional channel in the list will be given the opportunity to do so.

`view_channels`
: A comma-delimited list of channels that are sent the names of secondary index files when those files are updated. Each channel can choose whether to enqueue the index for compaction; once a channel has enqueued the index, no additional channel in the list will be given the opportunity to do so.

`staleness`
: The number of minutes that the (expensive) priority calculation on an individual can be stale for before it is recalculated. Defaults to 5.

`cleanup_index_files`
: If set to true, the compaction daemon will delete the files for indexes that are no longer associated with any design document. Defaults to *false* and probably shouldn't be changed unless the node is running low on disk space, and only after considering the ramifications.

`wait_secs`
: The time a channel waits before starting compactions to allow time to observe the system and make a smarter decision about what to compact first. Hardly ever changed from the default of 30 (seconds).

`[smoosh.{channel-name}]`

The following settings control the resource allocation for a given compaction channel.

`capacity`
: The maximum number of items the channel can hold (lowest priority item is removed to make room for new items). Defaults to 9999.

`concurrency`
: The maximum number of jobs that can run concurrently in this channel. Defaults to 1.

`from`

`to`
: The time period during which this channel is allowed to execute compactions. The value for each of these parameters must obey the format *HH:MM* with HH in [0..23] and MM in [0..59]. Each channel listed in the top-level daemon configuration continuously builds its priority queue regardless of the period defined here. The default is to allow the channel to execute compactions all the time.

`strict_window`
: If set to `true`, any compaction that is still running after the end of the allowed perio will be suspended, and then resumed during the next window. It defaults to `false`, in which case any running compactions will be allowed to finish, but no new ones will be started.

There are also several settings that collectively control whether a channel will enqueue a file for compaction and how it prioritizes files within its queue:

`max_priority`
: Each item must have a priority lower than this to be enqueued. Defaults to infinity.

max_size

The item must be no larger than this many bytes in length to be enqueued. Defaults to infinity.

min_priority

The item must have a priority at least this high to be enqueued. Defaults to 5.0 for ratio and 16 MB for slack.

min_changes

The minimum number of changes since last compaction before the item will be enqueued. Defaults to 0. Currently only works for databases.

min_size

The item must be at least this many bytes in length to be enqueued. Defaults to 1mb (1048576 bytes).

priority

The method used to calculate priority. Can be ratio (calculated as `sizes.file/sizes.active`) or slack (calculated as `sizes.file - sizes.active`). Defaults to ratio.

7.9 Background Indexing

Secondary indexes in CouchDB are not updated during document write operations. In order to avoid high latencies when reading indexes following a large block of writes, CouchDB automatically kicks off background jobs to keep secondary indexes "warm". The daemon responsible for this process is internally known as "ken" and can be configured using the following settings.

[ken]

batch_channels

This setting controls the number of background view builds that can be running in parallel at any given time. The default is 20.

incremental_channels

It is possible for all the slots in the normal build system to be occupied by long-running index rebuilds (e.g. if new design documents are posted to several databases simultaneously). In order to avoid already-built indexes from falling behind when this occurs, CouchDB will allow for a number of short background indexing jobs to run even when all slots are full. This setting controls how many additional short jobs are allowed to run concurrently with the main jobs. The default is 80.

max_incremental_updates

CouchDB estimates whether an indexing job is "incremental" or not by looking at the difference in sequence numbers between the current index and the main database. If the difference is larger than the threshold defined here the background job will only be allowed to run in the main queue. Defaults to 1000.

[ken.ignore]

Entries in this configuration section can be used to tell the background indexer to skip over specific database shard files. The key must be the exact name of the shard with the `.couch` suffix omitted, for example:

```
[ken.ignore]
shards/00000000-1fffffff/mydb.1567719095 = true
```

Note: In case when you'd like to skip all views from a ddoc, you may add `autoupdate: false` to the ddoc. All views of that ddoc will then be skipped.

More at *PUT /{db}/_design/{ddoc}*.

7.10 IO Queue

CouchDB has an internal subsystem that can prioritize IO associated with certain classes of operations. This subsystem can be configured to limit the resources devoted to background operations like internal replication and compaction according to the settings described below.

[ioq]

concurrency

Specifies the maximum number of concurrent in-flight IO requests that the queueing system will submit:

```
[ioq]
concurrency = 10
```

ratio

The fraction of the time that a background IO request will be selected over an interactive IO request when both queues are non-empty:

```
[ioq]
ratio = 0.01
```

[ioq.bypass]

System administrators can choose to submit specific classes of IO directly to the underlying file descriptor or OS process, bypassing the queues altogether. Installing a bypass can yield higher throughput and lower latency, but relinquishes some control over prioritization. The following classes are recognized:

os_process

Messages on their way to an external process (e.g., `couchjs`).

read

Disk IO fulfilling interactive read requests.

write

Disk IO required to update a database.

view_update

Disk IO required to update views and other secondary indexes.

shard_sync

Disk IO issued by the background replication processes that fix any inconsistencies between shard copies.

compaction

Disk IO issued by compaction jobs.

reshard

Disk IO issued by resharding jobs.

Without any configuration CouchDB will enqueue all classes of IO. The default.ini configuration file that ships with CouchDB activates a bypass for each of the interactive IO classes and only background IO goes into the queueing system:

```
[ioq.bypass]
os_process = true
read = true
write = true
view_update = true
shard_sync = false
compaction = false
reshard = false
```

7.10.1 Recommendations

The default configuration protects against excessive IO from background operations like compaction disrupting the latency of interactive operations, while maximizing the overall IO throughput devoted to those interactive requests. There are certain situations where this configuration could be sub-optimal:

- An administrator may want to devote a larger portion of the overall IO bandwidth to compaction in order to stay ahead of the incoming write load. In this it may be necessary to disable the bypass for `write` (to help with database compaction) and/or `view_update` (to help with view index compaction) and then increase the `ratio` to give compaction a higher priority.
- A server with a large number of views that do not need to be comlpetely up-to-date may benefit from removing the bypass on `view_update` in order to optimize the latency for regular document read and write operations, and build the views during quieter periods.

7.11 Logging

7.11.1 Logging options

[log]

CouchDB logging configuration.

writer

Current writers include:

- `stderr`: Logs are sent to stderr.
- `file`: Logs are sent to the file set in *log file*.
- `syslog`: Logs are sent to the syslog daemon.
- `journald`: Logs are sent to stderr without timestamp and log levels compatible with sd-daemon.

You can also specify a full module name here if implement your own writer:

```
[log]
writer = stderr
```

file

Specifies the location of file for logging output. Only used by the `file` *writer*:

```
[log]
file = /var/log/couchdb/couch.log
```

This path should be readable and writable for user that runs CouchDB service (*couchdb* by default).

write_buffer

Specifies the size of the file log write buffer in bytes, to enable delayed log writes. Only used by the `file` *writer*:

```
[log]
write_buffer = 0
```

write_delay

Specifies the wait in milliseconds before committing logs to disk, to enable delayed log writes. Only used by the `file` *writer*:

```
[log]
write_delay = 0
```

level

Changed in version 1.3: Added `warning` level.

Logging level defines how verbose and detailed logging will be:

```
[log]
level = info
```

Available levels:

- `debug`: Detailed debug logging.
- `info`: Informative logging. Includes HTTP requests headlines, startup of an external processes etc.
- `notice`
- `warning` or `warn`: Warning messages are alerts about edge situations that may lead to errors. For instance, compaction daemon alerts about low or insufficient disk space at this level.
- `error` or `err`: Error level includes only things that go wrong, like crash reports and HTTP error responses (5xx codes).
- `critical` or `crit`
- `alert`
- `emergency` or `emerg`
- `none`: Disables logging any messages.

include_sasl

Includes SASL information in logs:

```
[log]
include_sasl = true
```

syslog_host

Note: Setting *syslog_host* is mandatory for `syslog` to work!

Specifies the syslog host to send logs to. Only used by the `syslog` *writer*:

```
[log]
syslog_host = localhost
```

syslog_port

Specifies the syslog port to connect to when sending logs. Only used by the `syslog` *writer*:

```
[log]
syslog_port = 514
```

syslog_appid

Specifies application name to the `syslog` *writer*:

```
[log]
syslog_appid = couchdb
```

syslog_facility

Specifies the syslog facility to use with the `syslog` *writer*:

```
[log]
syslog_facility = local2
```

Note: CouchDB's `syslog` only knows how to use UDP logging. Please ensure that your `syslog` server has UDP logging enabled.

For `rsyslog` you can enable the UDP module *imudp* in `/etc/rsyslog.conf`:

```
# provides UDP syslog reception
module(load="imudp")
input(type="imudp" port="514")
```

7.12 Replicator

7.12.1 Replicator Database Configuration

[replicator]

max_jobs

New in version 2.1.

Number of actively running replications. This value represents the threshold to trigger the automatic replication scheduler. The system will check every `interval` milliseconds how many replication jobs are running, and if there are more than `max_jobs` active jobs, the scheduler will pause-and-restart up to `max_churn` jobs in the scheduler queue. Making this value too high could cause performance issues, while making it too low could mean replications jobs might not have enough time to make progress before getting unscheduled again. This parameter can be adjusted at runtime and will take effect during next rescheduling cycle:

```
[replicator]
max_jobs = 500
```

interval

New in version 2.1.

Scheduling interval in milliseconds. During each reschedule cycle the scheduler might start or stop up to `max_churn` number of jobs:

```
[replicator]
interval = 60000
```

max_churn

New in version 2.1.

Maximum number of replication jobs to start and stop during rescheduling. This parameter, along with `interval`, defines the rate of job replacement. During startup, however, a much larger number of jobs could be started (up to `max_jobs`) in a short period of time:

```
[replicator]
max_churn = 20
```

max_history

Maximum number of events recorded for each job. This parameter defines an upper bound on the consecutive failure count for a job, and in turn the maximum backoff factor used when determining the delay before the job is restarted. The longer the length of the crash count, the longer the possible length of the delay:

```
[replicator]
max_history = 20
```

update_docs

New in version 2.1.

When set to `true` replicator will update replication document with error and triggered states. This approximates pre-2.1 replicator behavior:

```
[replicator]
update_docs = false
```

worker_batch_size

With lower batch sizes checkpoints are done more frequently. Lower batch sizes also reduce the total amount of used RAM memory:

```
[replicator]
worker_batch_size = 500
```

worker_processes

More worker processes can give higher network throughput but can also imply more disk and network IO:

```
[replicator]
worker_processes = 4
```

http_connections

Maximum number of HTTP connections per replication:

```
[replicator]
http_connections = 20
```

connection_timeout

HTTP connection timeout per replication. This is divided by three (3) when the replicator makes changes feed requests. Even for very fast/reliable networks it might need to be increased if a remote database is too busy:

```
[replicator]
connection_timeout = 30000
```

retries_per_request

Changed in version 2.1.1.

If a request fails, the replicator will retry it up to N times. The default value for N is 5 (before version 2.1.1 it was 10). The requests are retried with a doubling exponential backoff starting at 0.25 seconds. So by default requests would be retried in 0.25, 0.5, 1, 2, 4 second intervals. When number of retires is exhausted, the whole replication job is stopped and will retry again later:

```
[replicator]
retries_per_request = 5
```

socket_options

Some socket options that might boost performance in some scenarios:

- `{nodelay, boolean()}`
- `{sndbuf, integer()}`
- `{recbuf, integer()}`

- `{priority, integer()}`

See the inet Erlang module's man page for the full list of options:

```
[replicator]
socket_options = [{keepalive, true}, {nodelay, false}]
```

valid_socket_options

New in version 3.3.

Valid socket options. Options not in this list are ignored. Most of those options are low level and setting some of them may lead to unintended or unpredictable behavior. See inet Erlang docs for the full list of options:

```
[replicator]
valid_socket_options = buffer,keepalive,nodelay,priority,recbuf,sndbuf
```

ibrowse_options

New in version 3.4: A non-default ibrowse setting is needed to support IPV6-only replication sources or targets:

- `{prefer_ipv6, boolean()}`

See the ibrowse site for the full list of options:

```
[replicator]
ibrowse_options = [{prefer_ipv6, true}]
```

valid_ibrowse_options

New in version 3.4.

Valid ibrowse options. Options not in this list are ignored:

```
[replicator]
valid_ibrowse_options = prefer_ipv6
```

valid_endpoint_protocols

New in version 3.3.

Valid replication endpoint protocols. Replication jobs with endpoint urls not in this list will fail to run:

```
[replicator]
valid_endpoint_protocols = http,https
```

valid_endpoint_protocols_log

New in version 3.4.

When enabled, CouchDB will log any replication that uses the insecure http protocol:

```
[replicator]
valid_endpoint_protocols_log = true
```

verify_ssl_certificates_log

New in version 3.4.

When enabled, and if `ssl_trusted_certificates_file` is configured but `verify_ssl_certificates` is not, CouchDB will check the validity of the TLS certificates of all sources and targets (without causing the replication to fail) and log any issues:

```
[replicator]
verify_ssl_certificates_log = true
```

valid_proxy_protocols

New in version 3.3.

Valid replication proxy protocols. Replication jobs with proxy urls not in this list will fail to run:

```
[replicator]
valid_proxy_protocols = http,https,socks5
```

checkpoint_interval

New in version 1.6.

Defines replication checkpoint interval in milliseconds. *Replicator* will `requests` from the Source database at the specified interval:

```
[replicator]
checkpoint_interval = 5000
```

Lower intervals may be useful for frequently changing data, while higher values will lower bandwidth and make fewer requests for infrequently updated databases.

use_checkpoints

New in version 1.6.

If `use_checkpoints` is set to `true`, CouchDB will make checkpoints during replication and at the completion of replication. CouchDB can efficiently resume replication from any of these checkpoints:

```
[replicator]
use_checkpoints = true
```

Note: Checkpoints are stored in *local documents* on both the source and target databases (which requires write access).

Warning: Disabling checkpoints is **not recommended** as CouchDB will scan the Source database's changes feed from the beginning.

use_bulk_get

New in version 3.3.

If `use_bulk_get` is `true`, CouchDB will attempt to use the `_bulk_get` HTTP API endpoint to fetch documents from the source. Replicator should automatically fall back to individual doc GETs on on error; however, in some cases it may be useful to prevent spending time attempting to call `_bulk_get` altogether.

cert_file

Path to a file containing the user's certificate:

```
[replicator]
cert_file = /full/path/to/server_cert.pem
```

key_file

Path to file containing user's private PEM encoded key:

```
[replicator]
key_file = /full/path/to/server_key.pem
```

password

String containing the user's password. Only used if the private key file is password protected:

```
[replicator]
password = somepassword
```

verify_ssl_certificates

Set to true to validate peer certificates:

```
[replicator]
verify_ssl_certificates = false
```

ssl_trusted_certificates_file

File containing a list of peer trusted certificates (in the PEM format):

```
[replicator]
ssl_trusted_certificates_file = /etc/ssl/certs/ca-certificates.crt
```

ssl_certificate_max_depth

Maximum peer certificate depth (must be set even if certificate validation is off):

```
[replicator]
ssl_certificate_max_depth = 3
```

auth_plugins

New in version 2.2.

List of replicator client authentication plugins. Plugins will be tried in order and the first to initialize successfully will be used. By default there are two plugins available: *couch_replicator_auth_session* implementing session (cookie) authentication, and *couch_replicator_auth_noop* implementing basic authentication. For backwards compatibility, the no-op plugin should be used at the end of the plugin list:

```
[replicator]
auth_plugins = couch_replicator_auth_session,couch_replicator_auth_noop
```

usage_coeff

New in version 3.2.0.

Usage coefficient decays historic fair share usage every scheduling cycle. The value must be between 0.0 and 1.0. Lower values will ensure historic usage decays quicker and higher values means it will be remembered longer:

```
[replicator]
usage_coeff = 0.5
```

priority_coeff

New in version 3.2.0.

Priority coefficient decays all the job priorities such that they slowly drift towards the front of the run queue. This coefficient defines a maximum time window over which this algorithm would operate. For example, if this value is too small (0.1), after a few cycles quite a few jobs would end up at priority 0, and would render this algorithm useless. The default value of 0.98 is picked such that if a job ran for one scheduler cycle, then didn't get to run for 7 hours, it would still have priority > 0. 7 hours was picked as it was close enough to 8 hours which is the default maximum error backoff interval:

```
[replicator]
priority_coeff = 0.98
```

7.12.2 Fair Share Replicator Share Allocation

[replicator.shares]

{replicator_db}

New in version 3.2.0.

Fair share configuration section. Higher share values results in a higher chance that jobs from that db get to run. The default value is 100, minimum is 1 and maximum is 1000. The configuration may be set even if the database does not exist.

In this context the option `{replicator_db}` acts as a placeholder for your replicator database name. The default replicator database is `_replicator`. Additional replicator databases can be created. To be recognized as such by the system, their database names should end with `/_replicator`. See the *Replicator Database* section for more info.

```
[replicator.shares]
_replicator = 50
foo/_replicator = 25
bar/_replicator = 25
```

7.13 Query Servers

7.13.1 Query Servers Definition

Changed in version 2.3: Changed configuration method for Query Servers and Native Query Servers.

CouchDB delegates computation of *design documents* functions to external query servers. The external query server is a special OS process which communicates with CouchDB over standard input/output using a very simple line-based protocol with JSON messages.

An external query server may be defined with environment variables following this pattern:

```
COUCHDB_QUERY_SERVER_LANGUAGE="PATH ARGS"
```

Where:

- `LANGUAGE`: is a programming language which code this query server may execute. For instance, there are *PYTHON*, *RUBY*, *CLOJURE* and other query servers in the wild. This value in *lowercase* is also used for `ddoc` field `language` to determine which query server processes the functions.

  Note, that you may set up multiple query servers for the same programming language, but you have to name them differently (like *PYTHONDEV* etc.).

- `PATH`: is a system path to the executable binary program that runs the query server.
- `ARGS`: optionally, you may specify additional command line arguments for the executable `PATH`.

The default query server is written in *JavaScript*, running via Mozilla SpiderMonkey. It requires no special environment settings to enable, but is the equivalent of these two variables:

```
COUCHDB_QUERY_SERVER_JAVASCRIPT="/opt/couchdb/bin/couchjs /opt/couchdb/share/server/
↪main.js"
COUCHDB_QUERY_SERVER_COFFEESCRIPT="/opt/couchdb/bin/couchjs /opt/couchdb/share/server/
↪main-coffee.js"
```

By default, `couchjs` limits the max runtime allocation to 64MiB. If you run into out of memory issue in your ddoc functions, you can adjust the memory limitation (here, increasing to 512 MiB):

```
COUCHDB_QUERY_SERVER_JAVASCRIPT="/usr/bin/couchjs -S 536870912 /usr/share/server/main.
↪js"
```

For more info about the available options, please consult `couchjs -h`.

Note: CouchDB versions 3.0.0 to 3.2.2 included a performance regression for custom reduce functions. CouchDB 3.3.0 and later come with an experimental fix to this issue that is included in a separate `.js` file.

To enable the fix, you need to define a custom `COUCHDB_QUERY_SERVER_JAVASCRIPT` environment variable as outlined above. The path to `couchjs` needs to remain the same as you find it on your `couchdb` file, and the path to `main.js` needs to be set to `/path/to/couchdb/share/server/main-ast-bypass.js`.

With a default installation on Linux systems, this is going to be `COUCHDB_QUERY_SERVER_JAVASCRIPT="/opt/couchdb/bin/couchjs /opt/couchdb/share/server/main-ast-bypass.js"`

See also:

The *Mango Query Server* is a declarative language that requires *no programming*, allowing for easier indexing and finding of data in documents.

The *Native Erlang Query Server* allows running *ddocs* written in Erlang natively, bypassing stdio communication and JSON serialization/deserialization round trip overhead.

7.13.2 Query Servers Configuration

[query_server_config]

commit_freq

Specifies the delay in seconds before view index changes are committed to disk. The default value is `5`:

```
[query_server_config]
commit_freq = 5
```

os_process_limit

Hard limit on the number of OS processes usable by Query Servers. The default value is `100`:

```
[query_server_config]
os_process_limit = 100
```

Setting `os_process_limit` too low can result in starvation of Query Servers, and manifest in `os_process_timeout` errors, while setting it too high can potentially use too many system resources. Production settings are typically 10-20 times the default value.

os_process_soft_limit

Soft limit on the number of OS processes usable by Query Servers. The default value is `100`:

```
[query_server_config]
os_process_soft_limit = 100
```

Idle OS processes are closed until the total reaches the soft limit.

For example, if the hard limit is 200 and the soft limit is 100, the total number of OS processes will never exceed 200, and CouchDB will close all idle OS processes until it reaches 100, at which point it will leave the rest intact, even if some are idle.

reduce_limit

Controls *Reduce overflow* error that raises when output of *reduce functions* is too big:

```
[query_server_config]
reduce_limit = true
```

Normally, you don't have to disable (by setting `false` value) this option since main propose of *reduce* functions is to *reduce* the input.

7.13.3 Native Erlang Query Server

[native_query_servers]

> **Warning:** Due to security restrictions, the Erlang query server is disabled by default.
>
> Unlike the JavaScript query server, the Erlang one does not run in a sandbox mode. This means that Erlang code has full access to your OS, file system and network, which may lead to security issues. While Erlang functions are faster than JavaScript ones, you need to be careful about running them, especially if they were written by someone else.

CouchDB has a native Erlang query server, allowing you to write your map/reduce functions in Erlang.

First, you'll need to edit your *local.ini* to include a `[native_query_servers]` section:

```
[native_query_servers]
enable_erlang_query_server = true
```

To see these changes you will also need to restart the server.

Let's try an example of map/reduce functions which count the total documents at each number of revisions (there are x many documents at version "1", and y documents at "2"… etc). Add a few documents to the database, then enter the following functions as a view:

```
%% Map Function
fun({Doc}) ->
    <<K,_/binary>> = proplists:get_value(<<"_rev">>, Doc, null),
    V = proplists:get_value(<<"_id">>, Doc, null),
    Emit(<<K>>, V)
end.

%% Reduce Function
fun(Keys, Values, ReReduce) -> length(Values) end.
```

If all has gone well, after running the view you should see a list of the total number of documents at each revision number.

Additional examples are on the users@couchdb.apache.org mailing list.

7.13.4 Search

CouchDB's search subsystem can be configured via the `dreyfus` configuration section.

[dreyfus]

name

The name and location of the Clouseau Java service required to enable Search functionality. Defaults to `clouseau@127.0.0.1`.

retry_limit

CouchDB will try to reconnect to Clouseau using a bounded exponential backoff with the following number of iterations. Defaults to `5`.

limit

The number of results returned from a global search query if no limit is specified. Defaults to `25`.

limit_partitions

The number of results returned from a search on a partition of a database if no limit is specified. Defaults to `2000`.

max_limit

The maximum number of results that can be returned from a global search query (or any search query on a database without user-defined partitions). Attempts to set `?limit=N` `higher` than this value will be rejected. Defaults to `200`.

max_limit_partitions

The maximum number of results that can be returned when searching a partition of a database. Attempts to set `?limit=N` higher than this value will be rejected. If this config setting is not defined, CouchDB will use the value of `max_limit` instead. If neither is defined, the default is `2000`.

7.13.5 Nouveau

CouchDB's experimental search subsystem can be configured via the `nouveau` configuration section.

[nouveau]

enable

Set to `true` to enable Nouveau. If disabled, all nouveau endpoints return 404 Not Found. Defaults to `false`.

url

The URL to a running nouveau server. Defaults to `http://127.0.0.1:8080`.

max_sessions

Nouveau will configure ibrowse max_sessions to this value for the configured `url`. Defaults to `100`.

max_pipeline_size

Nouveau will configure ibrowse max_pipeline_size to this value for the configured `url`. Defaults to `1000`.

7.13.6 Mango

Mango is the Query Engine that services the *_find*, endpoint.

[mango]

index_all_disabled

Set to `true` to disable the "index all fields" text index. This can lead to out of memory issues when there are documents with nested array fields. Defaults to `false`.

```
[mango]
index_all_disabled = false
```

default_limit

Sets the default number of results that will be returned in a *_find* response. Individual requests can override this by setting `limit` directly in the query parameters. Defaults to `25`.

```
[mango]
default_limit = 25
```

index_scan_warning_threshold

This sets the ratio between documents scanned and results matched that will generate a warning in the _find response. For example, if a query requires reading 100 documents to return 10 rows, a warning will be generated if this value is `10`.

Defaults to `10`. Setting the value to `0` disables the warning.

```
[mango]
index_scan_warning_threshold = 10
```

7.14 Miscellaneous Parameters

7.14.1 Configuration of Attachment Storage

[attachments]

compression_level

Defines zlib compression level for the attachments from 1 (lowest, fastest) to 9 (highest, slowest). A value of 0 disables compression:

```
[attachments]
compression_level = 8
```

compressible_types

Since compression is ineffective for some types of files, it is possible to let CouchDB compress only some types of attachments, specified by their MIME type:

```
[attachments]
compressible_types = text/*, application/javascript, application/json,␣
↪application/xml
```

7.14.2 Statistic Calculation

[stats]

interval

Interval between gathering statistics in seconds:

```
[stats]
interval = 10
```

7.14.3 UUIDs Configuration

[uuids]

algorithm

Changed in version 1.3: Added `utc_id` algorithm.

CouchDB provides various algorithms to generate the UUID values that are used for document *_id*'s by default:

```
[uuids]
algorithm = sequential
```

Available algorithms:

- `random`: 128 bits of random awesome. All awesome, all the time:

```
{
    "uuids": [
        "5fcbbf2cb171b1d5c3bc6df3d4affb32",
        "9115e0942372a87a977f1caf30b2ac29",
        "3840b51b0b81b46cab99384d5cd106e3",
        "b848dbdeb422164babf2705ac18173e1",
        "b7a8566af7e0fc02404bb676b47c3bf7",
        "a006879afdcae324d70e925c420c860d",
        "5f7716ee487cc4083545d4ca02cd45d4",
        "35fdd1c8346c22ccc43cc45cd632e6d6",
        "97bbdb4a1c7166682dc026e1ac97a64c",
        "eb242b506a6ae330bda6969bb2677079"
    ]
}
```

- `sequential`: Monotonically increasing ids with random increments. The first 26 hex characters are random, the last 6 increment in random amounts until an overflow occurs. On overflow, the random prefix is regenerated and the process starts over.

```
{
    "uuids": [
        "4e17c12963f4bee0e6ec90da54804894",
        "4e17c12963f4bee0e6ec90da5480512f",
        "4e17c12963f4bee0e6ec90da54805c25",
        "4e17c12963f4bee0e6ec90da54806ba1",
        "4e17c12963f4bee0e6ec90da548072b3",
        "4e17c12963f4bee0e6ec90da54807609",
        "4e17c12963f4bee0e6ec90da54807718",
        "4e17c12963f4bee0e6ec90da54807754",
        "4e17c12963f4bee0e6ec90da54807e5d",
        "4e17c12963f4bee0e6ec90da54808d28"
    ]
}
```

- `utc_random`: The time since Jan 1, 1970 UTC, in microseconds. The first 14 characters are the time in hex. The last 18 are random.

```
{
    "uuids": [
        "04dd32b3af699659b6db9486a9c58c62",
        "04dd32b3af69bb1c2ac7ebfee0a50d88",
        "04dd32b3af69d8591b99a8e86a76e0fb",
        "04dd32b3af69f4a18a76efd89867f4f4",
        "04dd32b3af6a1f7925001274bbfde952",
        "04dd32b3af6a3fe8ea9b120ed906a57f",
        "04dd32b3af6a5b5c518809d3d4b76654",
        "04dd32b3af6a78f6ab32f1e928593c73",
        "04dd32b3af6a99916c665d6bbf857475",
        "04dd32b3af6ab558dd3f2c0afacb7d66"
    ]
}
```

- `utc_id`: The time since Jan 1, 1970 UTC, in microseconds, plus the `utc_id_suffix` string. The first 14 characters are the time in hex. The *`uuids/utc_id_suffix`* string value is appended to these.

```
{
```

(continues on next page)

(continued from previous page)

```
    "uuids": [
        "04dd32bd5eabcc@mycouch",
        "04dd32bd5eabee@mycouch",
        "04dd32bd5eac05@mycouch",
        "04dd32bd5eac28@mycouch",
        "04dd32bd5eac43@mycouch",
        "04dd32bd5eac58@mycouch",
        "04dd32bd5eac6e@mycouch",
        "04dd32bd5eac84@mycouch",
        "04dd32bd5eac98@mycouch",
        "04dd32bd5eacad@mycouch"
    ]
}
```

Note: **Impact of UUID choices:** the choice of UUID has a significant impact on the layout of the B-tree, prior to compaction.

For example, using a sequential UUID algorithm while uploading a large batch of documents will avoid the need to rewrite many intermediate B-tree nodes. A random UUID algorithm may require rewriting intermediate nodes on a regular basis, resulting in significantly decreased throughput and wasted disk space space due to the append-only B-tree design.

It is generally recommended to set your own UUIDs, or use the sequential algorithm unless you have a specific need and take into account the likely need for compaction to re-balance the B-tree and reclaim wasted space.

utc_id_suffix

New in version 1.3.

The `utc_id_suffix` value will be appended to UUIDs generated by the `utc_id` algorithm. Replicating instances should have unique `utc_id_suffix` values to ensure uniqueness of `utc_id` ids.

```
[uuid]
utc_id_suffix = my-awesome-suffix
```

max_count

New in version 1.5.1.

No more than this number of UUIDs will be sent in a single request. If more UUIDs are requested, an HTTP error response will be thrown.

```
[uuid]
max_count = 1000
```

7.14.4 Vendor information

[vendor]

New in version 1.3.

CouchDB distributors have the option of customizing CouchDB's welcome message. This is returned when requesting `GET /`.

```
[vendor]
name = The Apache Software Foundation
version = 1.5.0
```

7.14.5 Content-Security-Policy

[csp]

You can configure `Content-Security-Policy` header for Fauxton, attachments and show/list functions separately. See MDN Content-Security-Policy for more details on CSP.

utils_enable

Enable the sending of the header `Content-Security-Policy` for `/_utils`. Defaults to `true`:

```
[csp]
utils_enable = true
```

utils_header_value

Specifies the exact header value to send. Defaults to:

```
[csp]
utils_header_value = default-src 'self'; img-src 'self'; font-src *; script-
↪src 'self' 'unsafe-eval'; style-src 'self' 'unsafe-inline';
```

attachments_enable

Enable sending the `Content-Security-Policy` header for attachments:

```
[csp]
attachments_enable = true
```

attachments_header_value

Specifies the exact header value to send. Defaults to:

```
[csp]
attachments_header_value = sandbox
```

showlist_enable

Enable sending the `Content-Security-Policy` header for show and list functions:

```
[csp]
showlist_enable = true
```

showlist_header_value

Specifies the exact header value to send. Defaults to:

```
[csp]
showlist_header_value = sandbox
```

The pre 3.2.0 behaviour is still honoured, but we recommend updating to the new format.

Experimental support of CSP headers for `/_utils` (Fauxton).

enable

Enable the sending of the Header `Content-Security-Policy`:

```
[csp]
enable = true
```

header_value

You can change the default value for the Header which is sent:

```
[csp]
header_value = default-src 'self'; img-src *; font-src *;
```

7.14.6 Configuration of Database Purge

[purge]

max_document_id_number

New in version 3.0.

Sets the maximum number of documents allowed in a single purge request:

```
[purge]
max_document_id_number = 100
```

max_revisions_number

New in version 3.0.

Sets the maximum number of accumulated revisions allowed in a single purge request:

```
[purge]
max_revisions_number = 1000
```

index_lag_warn_seconds

New in version 3.0.

Sets the allowed duration when index is not updated for local purge checkpoint document. Default is 24 hours:

```
[purge]
index_lag_warn_seconds = 86400
```

7.14.7 Configuration of Prometheus Endpoint

[prometheus]

additional_port

New in version 3.2.

Sets whether or not to create a separate, non-authenticated port (default is false):

```
[prometheus]
additional_port = true
```

bind_address

New in version 3.2.

The IP address to bind:

```
[prometheus]
bind_address = 127.0.0.1
```

port

New in version 3.2.

The port on which clients can query prometheus endpoint data without authentication:

```
[prometheus]
port = 17986
```

7.15 Resharding

7.15.1 Resharding Configuration

[resharding]

max_jobs

Maximum number of resharding jobs per cluster node. This includes completed, failed, and running jobs. If the job appears in the _reshard/jobs HTTP API results it will be counted towards the limit. When more than `max_jobs` jobs have been created, subsequent requests will start to fail with the `max_jobs_exceeded` error:

```
[reshard]
max_jobs = 48
```

max_history

Each resharding job maintains a timestamped event log. This setting limits the maximum size of that log:

```
[reshard]
max_history = 20
```

max_retries

How many times to retry shard splitting steps if they fail. For example, if indexing or topping off fails, it will be retried up to this many times before the whole resharding job fails:

```
[reshard]
max_retries = 1
```

retry_interval_sec

How long to wait between subsequent retries:

```
[reshard]
retry_interval_sec = 10
```

delete_source

Indicates if the source shard should be deleted after resharding has finished. By default, it is `true` as that would recover the space utilized by the shard. When debugging or when extra safety is required, this can be switched to `false`:

```
[reshard]
delete_source = true
```

update_shard_map_timeout_sec

How many seconds to wait for the shard map update operation to complete. If there is a large number of shard db changes waiting to finish replicating, it might be beneficial to increase this timeout:

```
[reshard]
update_shard_map_timeout_sec = 60
```

source_close_timeout_sec

How many seconds to wait for the source shard to close. "Close" in this context means that client requests which keep the database open have all finished:

```
[reshard]
source_close_timeout_sec = 600
```

require_node_param

Require users to specify a `node` parameter when creating resharding jobs. This can be used as a safety check to avoid inadvertently starting too many resharding jobs by accident:

```
[reshard]
require_node_param = false
```

require_range_param

Require users to specify a `range` parameter when creating resharding jobs. This can be used as a safety check to avoid inadvertently starting too many resharding jobs by accident:

```
[reshard]
require_range_param = false
```

CHAPTER

EIGHT

CLUSTER MANAGEMENT

As of CouchDB 2.0.0, CouchDB can be run in two different modes of operation:

- Standalone: In this mode, CouchDB's clustering is unavailable. CouchDB's HTTP-based replication with other CouchDB installations remains available.
- Cluster: A cluster of CouchDB installations internally replicate with each other via optimized network connections. This is intended to be used with servers that are in the same data center. This allows for database sharding to improve performance.

This section details the theory behind CouchDB clusters, and provides specific operational instructions on node, database and shard management.

8.1 Theory

Before we move on, we need some theory.

As you see in `etc/default.ini` there is a section called `[cluster]`

```
[cluster]
q=2
n=3
```

- `q` - The number of shards.
- `n` - The number of copies there is of every document. Replicas.

When creating a database you can send your own values with request and thereby override the defaults in `default.ini`.

The number of copies of a document with the same revision that have to be read before CouchDB returns with a `200` is equal to a half of total copies of the document plus one. It is the same for the number of nodes that need to save a document before a write is returned with `201`. If there are less nodes than that number, then `202` is returned. Both read and write numbers can be specified with a request as `r` and `w` parameters accordingly.

We will focus on the shards and replicas for now.

A shard is a part of a database. It can be replicated multiple times. The more copies of a shard, the more you can scale out. If you have 4 replicas, that means that all 4 copies of this specific shard will live on at most 4 nodes. No node can have more than one copy of each shard replica. The default for CouchDB since 3.0.0 is `q=2` and `n=3`, meaning each database (and secondary index) is split into 2 shards, with 3 replicas per shard, for a total of 6 shard replica files. For a CouchDB cluster only hosting a single database with these default values, a maximum of 6 nodes can be used to scale horizontally.

Replicas add failure resistance, as some nodes can be offline without everything crashing down.

- `n=1` All nodes must be up.
- `n=2` Any 1 node can be down.
- `n=3` Any 2 nodes can be down.

- etc

Computers go down and sysadmins pull out network cables in a furious rage from time to time, so using `n<2` is asking for downtime. Having too high a value of `n` adds servers and complexity without any real benefit. The sweet spot is at `n=3`.

Say that we have a database with 3 replicas and 4 shards. That would give us a maximum of 12 nodes: 4*3=12.

We can lose any 2 nodes and still read and write all documents.

What happens if we lose more nodes? It depends on how lucky we are. As long as there is at least one copy of every shard online, we can read and write all documents.

So, if we are very lucky then we can lose 8 nodes at maximum.

8.2 Node Management

8.2.1 Adding a node

Go to `http://server1:5984/_membership` to see the name of the node and all the nodes it is connected to and knows about.

```
curl -X GET "http://xxx.xxx.xxx.xxx:5984/_membership" --user admin-user
```

```
{
    "all_nodes":[
        "node1@xxx.xxx.xxx.xxx"],
    "cluster_nodes":[
        "node1@xxx.xxx.xxx.xxx"]
}
```

- `all_nodes` are all the nodes that this node knows about.
- `cluster_nodes` are the nodes that are connected to this node.

To add a node simply do:

```
curl -X PUT "http://xxx.xxx.xxx.xxx/_node/_local/_nodes/node2@yyy.yyy.yyy.yyy" -d {}
```

Now look at `http://server1:5984/_membership` again.

```
{
    "all_nodes":[
        "node1@xxx.xxx.xxx.xxx",
        "node2@yyy.yyy.yyy.yyy"
    ],
    "cluster_nodes":[
        "node1@xxx.xxx.xxx.xxx",
        "node2@yyy.yyy.yyy.yyy"
    ]
}
```

And you have a 2 node cluster :)

`http://yyy.yyy.yyy.yyy:5984/_membership` will show the same thing, so you only have to add a node once.

8.2.2 Removing a node

Before you remove a node, make sure that you have moved all *shards* away from that node.

To remove `node2` from server `yyy.yyy.yyy.yyy`, you need to first know the revision of the document that signifies that node's existence:

```
curl "http://xxx.xxx.xxx.xxx/_node/_local/_nodes/node2@yyy.yyy.yyy.yyy"
{"_id":"node2@yyy.yyy.yyy.yyy","_rev":"1-967a00dff5e02add41820138abb3284d"}
```

With that `_rev`, you can now proceed to delete the node document:

```
curl -X DELETE "http://xxx.xxx.xxx.xxx/_node/_local/_nodes/node2@yyy.yyy.yyy.yyy?
↪rev=1-967a00dff5e02add41820138abb3284d"
```

8.3 Database Management

8.3.1 Creating a database

This will create a database with `3` replicas and `8` shards.

```
curl -X PUT "http://xxx.xxx.xxx.xxx:5984/database-name?n=3&q=8" --user admin-user
```

The database is in `data/shards`. Look around on all the nodes and you will find all the parts.

If you do not specify `n` and `q` the default will be used. The default is `3` replicas and `8` shards.

8.3.2 Deleting a database

```
curl -X DELETE "http://xxx.xxx.xxx.xxx:5984/database-name --user admin-user
```

8.3.3 Placing a database on specific nodes

In BigCouch, the predecessor to CouchDB 2.0's clustering functionality, there was the concept of zones. CouchDB 2.0 carries this forward with cluster placement rules.

> **Warning:** Use of the `placement` argument will **override** the standard logic for shard replica cardinality (specified by `[cluster] n`.)

First, each node must be labeled with a zone attribute. This defines which zone each node is in. You do this by editing the node's document in the system `_nodes` database, which is accessed node-local via the `GET /_node/_local/_nodes/{node-name}` endpoint.

Add a key value pair of the form:

```
"zone": "metro-dc-a"
```

Do this for all of the nodes in your cluster.

In your config file (`local.ini` or `default.ini`) on each node, define a consistent cluster-wide setting like:

```
[cluster]
placement = metro-dc-a:2,metro-dc-b:1
```

In this example, it will ensure that two replicas for a shard will be hosted on nodes with the zone attribute set to `metro-dc-a` and one replica will be hosted on a new with the zone attribute set to `metro-dc-b`.

Note that you can also use this system to ensure certain nodes in the cluster do not host *any* replicas for newly created databases, by giving them a zone attribute that does not appear in the `[cluster]` placement string.

8.4 Shard Management

8.4.1 Introduction

This document discusses how sharding works in CouchDB along with how to safely add, move, remove, and create placement rules for shards and shard replicas.

A shard is a horizontal partition of data in a database. Partitioning data into shards and distributing copies of each shard (called "shard replicas" or just "replicas") to different nodes in a cluster gives the data greater durability against node loss. CouchDB clusters automatically shard databases and distribute the subsets of documents that compose each shard among nodes. Modifying cluster membership and sharding behavior must be done manually.

Shards and Replicas

How many shards and replicas each database has can be set at the global level, or on a per-database basis. The relevant parameters are `q` and `n`.

q is the number of database shards to maintain. *n* is the number of copies of each document to distribute. The default value for `n` is `3`, and for `q` is `2`. With `q=2`, the database is split into 2 shards. With `n=3`, the cluster distributes three replicas of each shard. Altogether, that's 6 shard replicas for a single database.

In a 3-node cluster with `q=8`, each node would receive 8 shards. In a 4-node cluster, each node would receive 6 shards. We recommend in the general case that the number of nodes in your cluster should be a multiple of `n`, so that shards are distributed evenly.

CouchDB nodes have a `etc/default.ini` file with a section named cluster which looks like this:

```
[cluster]
q=2
n=3
```

These settings specify the default sharding parameters for newly created databases. These can be overridden in the `etc/local.ini` file by copying the text above, and replacing the values with your new defaults. If `[couch_peruser] q` is set, that value is used for per-user databases. (By default, it is set to 1, on the assumption that per-user dbs will be quite small and there will be many of them.) The values can also be set on a per-database basis by specifying the `q` and `n` query parameters when the database is created. For example:

```
$ curl -X PUT "$COUCH_URL:5984/database-name?q=4&n=2"
```

This creates a database that is split into 4 shards and 2 replicas, yielding 8 shard replicas distributed throughout the cluster.

Quorum

Depending on the size of the cluster, the number of shards per database, and the number of shard replicas, not every node may have access to every shard, but every node knows where all the replicas of each shard can be found through CouchDB's internal shard map.

Each request that comes in to a CouchDB cluster is handled by any one random coordinating node. This coordinating node proxies the request to the other nodes that have the relevant data, which may or may not include itself. The coordinating node sends a response to the client once a quorum of database nodes have responded; 2, by default. The default required size of a quorum is equal to `r=w=((n div 2) + 1)` where `r` refers to the size of a read quorum, `w` refers to the size of a write quorum, `n` refers to the number of replicas of each shard, and `div` is integer division, rounding down. In a default cluster where `n` is 3, `((n div 2) + 1)` would be 2.

Note: Each node in a cluster can be a coordinating node for any one request. There are no special roles for nodes inside the cluster.

The size of the required quorum can be configured at request time by setting the `r` parameter for document reads, and the `w` parameter for document writes. The `_view`, `_find`, and `_search` endpoints read only one copy no matter what quorum is configured, effectively making a quorum of 1 for these requests.

For example, here is a request that directs the coordinating node to send a response once at least two nodes have responded:

```
$ curl "$COUCH_URL:5984/{db}/{doc}?r=2"
```

Here is a similar example for writing a document:

```
$ curl -X PUT "$COUCH_URL:5984/{db}/{doc}?w=2" -d '{...}'
```

Setting `r` or `w` to be equal to `n` (the number of replicas) means you will only receive a response once all nodes with relevant shards have responded or timed out, and as such this approach does not guarantee ACIDic consistency. Setting `r` or `w` to 1 means you will receive a response after only one relevant node has responded.

8.4.2 Examining database shards

There are a few API endpoints that help you understand how a database is sharded. Let's start by making a new database on a cluster, and putting a couple of documents into it:

```
$ curl -X PUT $COUCH_URL:5984/mydb
{"ok":true}
$ curl -X PUT $COUCH_URL:5984/mydb/joan -d '{"loves":"cats"}'
{"ok":true,"id":"joan","rev":"1-cc240d66a894a7ee7ad3160e69f9051f"}
$ curl -X PUT $COUCH_URL:5984/mydb/robert -d '{"loves":"dogs"}'
{"ok":true,"id":"robert","rev":"1-4032b428c7574a85bc04f1f271be446e"}
```

First, the top level */{db}* endpoint will tell you what the sharding parameters are for your database:

```
$ curl -s $COUCH_URL:5984/db | jq .
{
  "db_name": "mydb",
...
  "cluster": {
    "q": 8,
    "n": 3,
    "w": 2,
    "r": 2
  },
```

(continues on next page)

(continued from previous page)

```
...
}
```

So we know this database was created with 8 shards (`q=8`), and each shard has 3 replicas (`n=3`) for a total of 24 shard replicas across the nodes in the cluster.

Now, let's see how those shard replicas are placed on the cluster with the */{db}/_shards* endpoint:

```
$ curl -s $COUCH_URL:5984/mydb/_shards | jq .
{
  "shards": {
    "00000000-1fffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node4@127.0.0.1"
    ],
    "20000000-3fffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node3@127.0.0.1"
    ],
    "40000000-5fffffff": [
      "node2@127.0.0.1",
      "node3@127.0.0.1",
      "node4@127.0.0.1"
    ],
    "60000000-7fffffff": [
      "node1@127.0.0.1",
      "node3@127.0.0.1",
      "node4@127.0.0.1"
    ],
    "80000000-9fffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node4@127.0.0.1"
    ],
    "a0000000-bfffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node3@127.0.0.1"
    ],
    "c0000000-dfffffff": [
      "node2@127.0.0.1",
      "node3@127.0.0.1",
      "node4@127.0.0.1"
    ],
    "e0000000-ffffffff": [
      "node1@127.0.0.1",
      "node3@127.0.0.1",
      "node4@127.0.0.1"
    ]
  }
}
```

Now we see that there are actually 4 nodes in this cluster, and CouchDB has spread those 24 shard replicas evenly across all 4 nodes.

We can also see exactly which shard contains a given document with the */{db}/_shards/{docid}* endpoint:

```
$ curl -s $COUCH_URL:5984/mydb/_shards/joan | jq .
{
  "range": "e0000000-ffffffff",
  "nodes": [
    "node1@127.0.0.1",
    "node3@127.0.0.1",
    "node4@127.0.0.1"
  ]
}
$ curl -s $COUCH_URL:5984/mydb/_shards/robert | jq .
{
  "range": "60000000-7fffffff",
  "nodes": [
    "node1@127.0.0.1",
    "node3@127.0.0.1",
    "node4@127.0.0.1"
  ]
}
```

CouchDB shows us the specific shard into which each of the two sample documents is mapped.

8.4.3 Moving a shard

When moving shards or performing other shard manipulations on the cluster, it is advisable to stop all resharding jobs on the cluster. See *Stopping Resharding Jobs* for more details.

This section describes how to manually place and replace shards. These activities are critical steps when you determine your cluster is too big or too small, and want to resize it successfully, or you have noticed from server metrics that database/shard layout is non-optimal and you have some "hot spots" that need resolving.

Consider a three-node cluster with q=8 and n=3. Each database has 24 shards, distributed across the three nodes. If you *add a fourth node* to the cluster, CouchDB will not redistribute existing database shards to it. This leads to unbalanced load, as the new node will only host shards for databases created after it joined the cluster. To balance the distribution of shards from existing databases, they must be moved manually.

Moving shards between nodes in a cluster involves the following steps:

0. *Ensure the target node has joined the cluster*.
1. Copy the shard(s) and any secondary *index shard(s) onto the target node*.
2. *Set the target node to maintenance mode*.
3. Update cluster metadata *to reflect the new target shard(s)*.
4. Monitor internal replication *to ensure up-to-date shard(s)*.
5. *Clear the target node's maintenance mode*.
6. Update cluster metadata again *to remove the source shard(s)*
7. Remove the shard file(s) and secondary index file(s) *from the source node*.

Copying shard files

Note: Technically, copying database and secondary index shards is optional. If you proceed to the next step without performing this data copy, CouchDB will use internal replication to populate the newly added shard replicas. However, copying files is faster than internal replication, especially on a busy cluster, which is why we recommend performing this manual data copy first.

Shard files live in the `data/shards` directory of your CouchDB install. Within those subdirectories are the shard files themselves. For instance, for a `q=8` database called `abc`, here is its database shard files:

```
data/shards/00000000-1fffffff/abc.1529362187.couch
data/shards/20000000-3fffffff/abc.1529362187.couch
data/shards/40000000-5fffffff/abc.1529362187.couch
data/shards/60000000-7fffffff/abc.1529362187.couch
data/shards/80000000-9fffffff/abc.1529362187.couch
data/shards/a0000000-bfffffff/abc.1529362187.couch
data/shards/c0000000-dfffffff/abc.1529362187.couch
data/shards/e0000000-ffffffff/abc.1529362187.couch
```

Secondary indexes (including JavaScript views, Erlang views and Mango indexes) are also sharded, and their shards should be moved to save the new node the effort of rebuilding the view. View shards live in `data/.shards`. For example:

```
data/.shards
data/.shards/e0000000-ffffffff/_replicator.1518451591_design
data/.shards/e0000000-ffffffff/_replicator.1518451591_design/mrview
data/.shards/e0000000-ffffffff/_replicator.1518451591_design/mrview/
↪3e823c2a4383ac0c18d4e574135a5b08.view
data/.shards/c0000000-dfffffff
data/.shards/c0000000-dfffffff/_replicator.1518451591_design
data/.shards/c0000000-dfffffff/_replicator.1518451591_design/mrview
data/.shards/c0000000-dfffffff/_replicator.1518451591_design/mrview/
↪3e823c2a4383ac0c18d4e574135a5b08.view
...
```

Since they are files, you can use `cp`, `rsync`, `scp` or other file-copying command to copy them from one node to another. For example:

```
# one one machine
$ mkdir -p data/.shards/{range}
$ mkdir -p data/shards/{range}
# on the other
$ scp {couch-dir}/data/.shards/{range}/{database}.{datecode}* \
  {node}:{couch-dir}/data/.shards/{range}/
$ scp {couch-dir}/data/shards/{range}/{database}.{datecode}.couch \
  {node}:{couch-dir}/data/shards/{range}/
```

Note: Remember to move view files before database files! If a view index is ahead of its database, the database will rebuild it from scratch.

Set the target node to `true` maintenance mode

Before telling CouchDB about these new shards on the node, the node must be put into maintenance mode. Maintenance mode instructs CouchDB to return a `404 Not Found` response on the `/_up` endpoint, and ensures it does not participate in normal interactive clustered requests for its shards. A properly configured load balancer that uses `GET /_up` to check the health of nodes will detect this 404 and remove the node from circulation, preventing requests from being sent to that node. For example, to configure HAProxy to use the `/_up` endpoint, use:

```
http-check disable-on-404
option httpchk GET /_up
```

If you do not set maintenance mode, or the load balancer ignores this maintenance mode status, after the next step is performed the cluster may return incorrect responses when consulting the node in question. You don't want this! In the next steps, we will ensure that this shard is up-to-date before allowing it to participate in end-user requests.

To enable maintenance mode:

```
$ curl -X PUT -H "Content-type: application/json" \
    $COUCH_URL:5984/_node/{node-name}/_config/couchdb/maintenance_mode \
    -d "\"true\""
```

Then, verify that the node is in maintenance mode by performing a `GET /_up` on that node's individual endpoint:

```
$ curl -v $COUCH_URL/_up
...
< HTTP/1.1 404 Object Not Found
...
{"status":"maintenance_mode"}
```

Finally, check that your load balancer has removed the node from the pool of available backend nodes.

Updating cluster metadata to reflect the new target shard(s)

Now we need to tell CouchDB that the target node (which must already be *joined to the cluster*) should be hosting shard replicas for a given database.

To update the cluster metadata, use the special `/_dbs` database, which is an internal CouchDB database that maps databases to shards and nodes. This database is automatically replicated between nodes. It is accessible only through the special `/_node/_local/_dbs` endpoint.

First, retrieve the database's current metadata:

```
$ curl http://adm:pass@localhost:5984/_node/_local/_dbs/{name}
{
  "_id": "{name}",
  "_rev": "1-e13fb7e79af3b3107ed62925058bfa3a",
  "shard_suffix": [46, 49, 53, 51, 48, 50, 51, 50, 53, 50, 54],
  "changelog": [
    ["add", "00000000-1fffffff", "node1@xxx.xxx.xxx.xxx"],
    ["add", "00000000-1fffffff", "node2@xxx.xxx.xxx.xxx"],
    ["add", "00000000-1fffffff", "node3@xxx.xxx.xxx.xxx"],
    ...
  ],
  "by_node": {
    "node1@xxx.xxx.xxx.xxx": [
      "00000000-1fffffff",
      ...
    ],
    ...
```

(continues on next page)

(continued from previous page)

```
  },
  "by_range": {
    "00000000-1fffffff": [
      "node1@xxx.xxx.xxx.xxx",
      "node2@xxx.xxx.xxx.xxx",
      "node3@xxx.xxx.xxx.xxx"
    ],
    ...
  }
}
```

Here is a brief anatomy of that document:

- `_id`: The name of the database.
- `_rev`: The current revision of the metadata.
- `shard_suffix`: A timestamp of the database's creation, marked as seconds after the Unix epoch mapped to the codepoints for ASCII numerals.
- `changelog`: History of the database's shards.
- `by_node`: List of shards on each node.
- `by_range`: On which nodes each shard is.

To reflect the shard move in the metadata, there are three steps:

1. Add appropriate changelog entries.
2. Update the `by_node` entries.
3. Update the `by_range` entries.

Warning: Be very careful! Mistakes during this process can irreparably corrupt the cluster!

As of this writing, this process must be done manually.

To add a shard to a node, add entries like this to the database metadata's `changelog` attribute:

```
["add", "{range}", "{node-name}"]
```

The `{range}` is the specific shard range for the shard. The `{node-name}` should match the name and address of the node as displayed in `GET /_membership` on the cluster.

Note: When removing a shard from a node, specify `remove` instead of `add`.

Once you have figured out the new changelog entries, you will need to update the `by_node` and `by_range` to reflect who is storing what shards. The data in the changelog entries and these attributes must match. If they do not, the database may become corrupted.

Continuing our example, here is an updated version of the metadata above that adds shards to an additional node called `node4`:

```
{
  "_id": "{name}",
  "_rev": "1-e13fb7e79af3b3107ed62925058bfa3a",
  "shard_suffix": [46, 49, 53, 51, 48, 50, 51, 50, 53, 50, 54],
  "changelog": [
    ["add", "00000000-1fffffff", "node1@xxx.xxx.xxx.xxx"],
```

(continues on next page)

(continued from previous page)

```
    ["add", "00000000-1fffffff", "node2@xxx.xxx.xxx.xxx"],
    ["add", "00000000-1fffffff", "node3@xxx.xxx.xxx.xxx"],
    ...
    ["add", "00000000-1fffffff", "node4@xxx.xxx.xxx.xxx"]
  ],
  "by_node": {
    "node1@xxx.xxx.xxx.xxx": [
      "00000000-1fffffff",
      ...
    ],
    ...
    "node4@xxx.xxx.xxx.xxx": [
      "00000000-1fffffff"
    ]
  },
  "by_range": {
    "00000000-1fffffff": [
      "node1@xxx.xxx.xxx.xxx",
      "node2@xxx.xxx.xxx.xxx",
      "node3@xxx.xxx.xxx.xxx",
      "node4@xxx.xxx.xxx.xxx"
    ],
    ...
  }
}
```

Now you can PUT this new metadata:

```
$ curl -X PUT http://adm:pass@localhost:5984/_node/_local/_dbs/{name} -d '{...}'
```

Forcing synchronization of the shard(s)

New in version 2.4.0.

Whether you pre-copied shards to your new node or not, you can force CouchDB to synchronize all replicas of all shards in a database with the */{db}/_sync_shards* endpoint:

```
$ curl -X POST $COUCH_URL:5984/{db}/_sync_shards
{"ok":true}
```

This starts the synchronization process. Note that this will put additional load onto your cluster, which may affect performance.

It is also possible to force synchronization on a per-shard basis by writing to a document that is stored within that shard.

Note: Admins may want to bump their `[mem3] sync_concurrency` value to a larger figure for the duration of the shards sync.

Monitor internal replication to ensure up-to-date shard(s)

After you complete the previous step, CouchDB will have started synchronizing the shards. You can observe this happening by monitoring the `/_node/{node-name}/_system` endpoint, which includes the `internal_replication_jobs` metric.

Once this metric has returned to the baseline from before you started the shard sync, or is `0`, the shard replica is ready to serve data and we can bring the node out of maintenance mode.

Clear the target node's maintenance mode

You can now let the node start servicing data requests by putting `"false"` to the maintenance mode configuration endpoint, just as in step 2.

Verify that the node is not in maintenance mode by performing a `GET /_up` on that node's individual endpoint.

Finally, check that your load balancer has returned the node to the pool of available backend nodes.

Update cluster metadata again to remove the source shard

Now, remove the source shard from the shard map the same way that you added the new target shard to the shard map in step 2. Be sure to add the `["remove", {range}, {source-shard}]` entry to the end of the changelog as well as modifying both the `by_node` and `by_range` sections of the database metadata document.

Remove the shard and secondary index files from the source node

Finally, you can remove the source shard replica by deleting its file from the command line on the source host, along with any view shard replicas:

```
$ rm {couch-dir}/data/shards/{range}/{db}.{datecode}.couch
$ rm -r {couch-dir}/data/.shards/{range}/{db}.{datecode}*
```

Congratulations! You have moved a database shard replica. By adding and removing database shard replicas in this way, you can change the cluster's shard layout, also known as a shard map.

8.4.4 Specifying database placement

You can configure CouchDB to put shard replicas on certain nodes at database creation time using placement rules.

> **Warning:** Use of the `placement` option will **override** the `n` option, both in the `.ini` file as well as when specified in a `URL`.

First, each node must be labeled with a zone attribute. This defines which zone each node is in. You do this by editing the node's document in the special `/_nodes` database, which is accessed through the special node-local API endpoint at `/_node/_local/_nodes/{node-name}`. Add a key value pair of the form:

```
"zone": "{zone-name}"
```

Do this for all of the nodes in your cluster. For example:

```
$ curl -X PUT http://adm:pass@localhost:5984/_node/_local/_nodes/{node-name} \
    -d '{ \
        "_id": "{node-name}",
        "_rev": "{rev}",
        "zone": "{zone-name}"
        }'
```

In the local config file (`local.ini`) of each node, define a consistent cluster-wide setting like:

```
[cluster]
placement = {zone-name-1}:2,{zone-name-2}:1
```

In this example, CouchDB will ensure that two replicas for a shard will be hosted on nodes with the zone attribute set to `{zone-name-1}` and one replica will be hosted on a new with the zone attribute set to `{zone-name-2}`.

This approach is flexible, since you can also specify zones on a per- database basis by specifying the placement setting as a query parameter when the database is created, using the same syntax as the ini file:

```
curl -X PUT $COUCH_URL:5984/{db}?zone={zone}
```

The `placement` argument may also be specified. Note that this *will* override the logic that determines the number of created replicas!

Note that you can also use this system to ensure certain nodes in the cluster do not host any replicas for newly created databases, by giving them a zone attribute that does not appear in the `[cluster]` placement string.

8.4.5 Splitting Shards

The */_reshard* is an HTTP API for shard manipulation. Currently it only supports shard splitting. To perform shard merging, refer to the manual process outlined in the *Merging Shards* section.

The main way to interact with */_reshard* is to create resharding jobs, monitor those jobs, wait until they complete, remove them, post new jobs, and so on. What follows are a few steps one might take to use this API to split shards.

At first, it's a good idea to call `GET /_reshard` to see a summary of resharding on the cluster.

```
$ curl -s $COUCH_URL:5984/_reshard | jq .
{
  "state": "running",
  "state_reason": null,
  "completed": 3,
  "failed": 0,
  "running": 0,
  "stopped": 0,
  "total": 3
}
```

Two important things to pay attention to are the total number of jobs and the state.

The `state` field indicates the state of resharding on the cluster. Normally it would be `running`, however, another user could have disabled resharding temporarily. Then, the state would be `stopped` and hopefully, there would be a reason or a comment in the value of the `state_reason` field. See *Stopping Resharding Jobs* for more details.

The `total` number of jobs is important to keep an eye on because there is a maximum number of resharding jobs per node, and creating new jobs after the limit has been reached will result in an error. Before staring new jobs it's a good idea to remove already completed jobs. See *reshard configuration section* for the default value of `max_jobs` parameter and how to adjust if needed.

For example, to remove all the completed jobs run:

```
$ for jobid in $(curl -s $COUCH_URL:5984/_reshard/jobs | jq -r '.jobs[] | select (.
↪job_state=="completed") | .id'); do \
      curl -s -XDELETE $COUCH_URL:5984/_reshard/jobs/$jobid \
  done
```

Then it's a good idea to see what the db shard map looks like.

```
$ curl -s $COUCH_URL:5984/db1/_shards | jq '.'
{
  "shards": {
    "00000000-7fffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node3@127.0.0.1"
    ],
    "80000000-ffffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node3@127.0.0.1"
    ]
  }
}
```

In this example we'll split all the copies of the `00000000-7fffffff` range. The API allows a combination of parameters such as: splitting all the ranges on all the nodes, all the ranges on just one node, or one particular range on one particular node. These are specified via the `db`, `node` and `range` job parameters.

To split all the copies of `00000000-7fffffff` we issue a request like this:

```
$ curl -s -H "Content-type: application/json" -XPOST $COUCH_URL:5984/_reshard/jobs \
  -d '{"type": "split", "db":"db1", "range":"00000000-7fffffff"}' | jq '.'
[
  {
    "ok": true,
    "id": "001-ef512cfb502a1c6079fe17e9dfd5d6a2befcc694a146de468b1ba5339ba1d134",
    "node": "node1@127.0.0.1",
    "shard": "shards/00000000-7fffffff/db1.1554242778"
  },
  {
    "ok": true,
    "id": "001-cec63704a7b33c6da8263211db9a5c74a1cb585d1b1a24eb946483e2075739ca",
    "node": "node2@127.0.0.1",
    "shard": "shards/00000000-7fffffff/db1.1554242778"
  },
  {
    "ok": true,
    "id": "001-fc72090c006d9b059d4acd99e3be9bb73e986d60ca3ededede3cb74cc01ccd1456",
    "node": "node3@127.0.0.1",
    "shard": "shards/00000000-7fffffff/db1.1554242778"
  }
]
```

The request returned three jobs, one job for each of the three copies.

To check progress of these jobs use `GET /_reshard/jobs` or `GET /_reshard/jobs/{jobid}`.

Eventually, these jobs should complete and the shard map should look like this:

```
$ curl -s $COUCH_URL:5984/db1/_shards | jq '.'
{
  "shards": {
    "00000000-3fffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node3@127.0.0.1"
    ],
```

(continues on next page)

(continued from previous page)

```
    "40000000-7fffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node3@127.0.0.1"
    ],
    "80000000-ffffffff": [
      "node1@127.0.0.1",
      "node2@127.0.0.1",
      "node3@127.0.0.1"
    ]
  }
}
```

8.4.6 Stopping Resharding Jobs

Resharding at the cluster level could be stopped and then restarted. This can be helpful to allow external tools which manipulate the shard map to avoid interfering with resharding jobs. To stop all resharding jobs on a cluster issue a PUT to `/_reshard/state` endpoint with the `"state": "stopped"` key and value. You can also specify an optional note or reason for stopping.

For example:

```
$ curl -s -H "Content-type: application/json" \
  -XPUT $COUCH_URL:5984/_reshard/state \
  -d '{"state": "stopped", "reason":"Moving some shards"}'
{"ok": true}
```

This state will then be reflected in the global summary:

```
$ curl -s $COUCH_URL:5984/_reshard | jq .
{
  "state": "stopped",
  "state_reason": "Moving some shards",
  "completed": 74,
  "failed": 0,
  "running": 0,
  "stopped": 0,
  "total": 74
}
```

To restart, issue a PUT request like above with `running` as the state. That should resume all the shard splitting jobs since their last checkpoint.

See the API reference for more details: */_reshard*.

8.4.7 Merging Shards

The `q` value for a database can be set when the database is created or it can be increased later by splitting some of the shards *Splitting Shards*. In order to decrease `q` and merge some shards together, the database must be regenerated. Here are the steps:

1. If there are running shard splitting jobs on the cluster, stop them via the HTTP API *Stopping Resharding Jobs*.
2. Create a temporary database with the desired shard settings, by specifying the q value as a query parameter during the PUT operation.
3. Stop clients accessing the database.

4. Replicate the primary database to the temporary one. Multiple replications may be required if the primary database is under active use.
5. Delete the primary database. **Make sure nobody is using it!**
6. Recreate the primary database with the desired shard settings.
7. Clients can now access the database again.
8. Replicate the temporary back to the primary.
9. Delete the temporary database.

Once all steps have completed, the database can be used again. The cluster will create and distribute its shards according to placement rules automatically.

Downtime can be avoided in production if the client application(s) can be instructed to use the new database instead of the old one, and a cut- over is performed during a very brief outage window.

8.5 Clustered Purge

The primary purpose of clustered purge is to clean databases that have multiple deleted tombstones or single documents that contain large numbers of conflicts. But it can also be used to purge any document (deleted or non-deleted) with any number of revisions.

Clustered purge is designed to maintain eventual consistency and prevent unnecessary invalidation of secondary indexes. For this, every database keeps track of a certain number of historical purges requested in the database, as well as its current `purge_seq`. Internal replications and secondary indexes process database's purges and periodically update their corresponding purge checkpoint documents to report `purge_seq` processed by them. To ensure eventual consistency, the database will remove stored historical purge requests only after they have been processed by internal replication jobs and secondary indexes.

8.5.1 Internal Structures

To enable internal replication of purge information between nodes and secondary indexes, two internal purge trees were added to a database file to track historical purges.

```
purge_tree: UUID -> {PurgeSeq, DocId, Revs}
purge_seq_tree: PurgeSeq -> {UUID, DocId, Revs}
```

Each interactive request to `_purge API`, creates an ordered set of pairs on increasing `purge_seq` and purge_request, where purge_request is a tuple that contains docid and list of revisions. For each purge_request uuid is generated. A purge request is added to internal purge trees: a tuple `{UUID -> {PurgeSeq, DocId, Revs}}` is added to `purge_tree`, a tuple is `{PurgeSeq -> {UUID, DocId, Revs}}` added `to purge_seq_tree`.

8.5.2 Compaction of Purges

During the compaction of the database the oldest purge requests are to be removed to store only `purged_infos_limit` number of purges in the database. But in order to keep the database consistent with indexes and other replicas, we can only remove purge requests that have already been processed by indexes and internal replications jobs. Thus, occasionally purge trees may store more than `purged_infos_limit` purges. If the number of stored purges in the database exceeds `purged_infos_limit` by a certain threshold, a warning is produced in logs signaling a problem of synchronization of database's purges with indexes and other replicas.

8.5.3 Local Purge Checkpoint Documents

Indexes and internal replications of the database with purges create and periodically update local checkpoint purge documents: `_local/purge-{type}-{hash}`. These documents report the last `purge_seq` processed by them and the timestamp of the last processing. These documents are only visible in `_local_docs` when you add a `include_system=true` parameter, so e.g. `/test-db/_local_docs?include_system=true`. An example of a local checkpoint purge document:

```
{
  "_id": "_local/purge-mrview-86cacdfbaf6968d4ebbc324dd3723fe7",
  "type": "mrview",
  "purge_seq": 10,
  "updated_on": 1540541874,
  "ddoc_id": "_design/foo",
  "signature": "5d10247925f826ae3e00966ec24b7bf6"
}
```

The below image shows possible local checkpoint documents that a database may have.

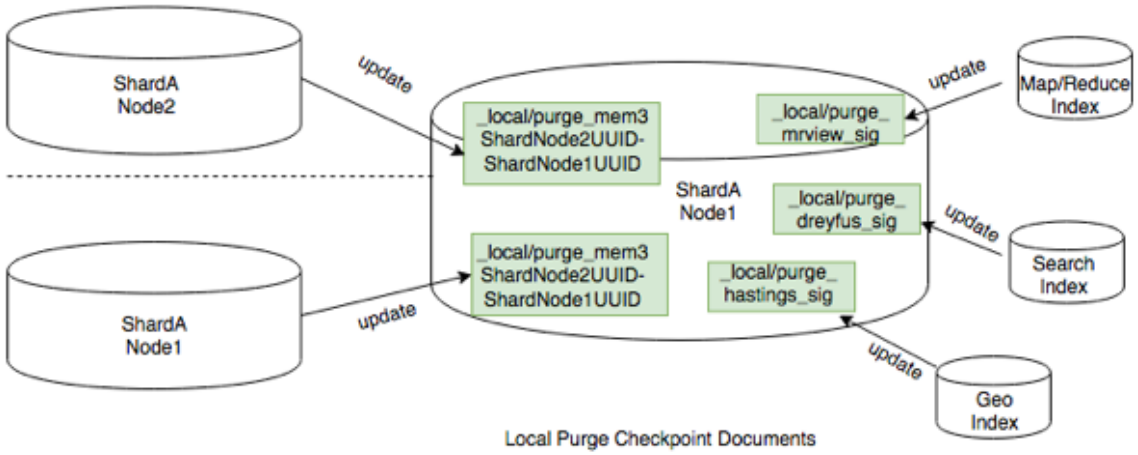

Fig. 1: Local Purge Checkpoint Documents

8.5.4 Internal Replication

Purge requests are replayed across all nodes in an eventually consistent manner. Internal replication of purges consists of two steps:

1. Pull replication. Internal replication first starts by pulling purges from target and applying them on source to make sure we don't reintroduce to target source's docs/revs that have been already purged on target. In this step, we use purge checkpoint documents stored on target to keep track of the last target's `purge_seq` processed by the source. We find purge requests occurred after this `purge_seq`, and replay them on source. This step is done by updating the target's checkpoint purge documents with the latest process `purge_seq` and timestamp.

2. Push replication. Then internal replication proceeds as usual with an extra step inserted to push source's purge requests to target. In this step, we use local internal replication checkpoint documents, that are updated both on target and source.

Under normal conditions, an interactive purge request is already sent to every node containing a database shard's replica, and applied on every replica. Internal replication of purges between nodes is just an extra step to ensure consistency between replicas, where all purge requests on one node are replayed on another node. In order not to replay the same purge request on a replica, each interactive purge request is tagged with a unique `uuid`. Internal

replication filters out purge requests with UUIDs that already exist in the replica's `purge_tree`, and applies only purge requests with UUIDs that don't exist in the `purge_tree`. This is the reason why we needed to have two internal purge trees: 1) `purge_tree`: `{UUID -> {PurgeSeq, DocId, Revs}}` allows to quickly find purge requests with UUIDs that already exist in the replica; 2) `purge_seq_tree`: `{PurgeSeq -> {UUID, DocId, Revs}}` allows to iterate from a given `purge_seq` to collect all purge requests happened after this `purge_seq`.

8.5.5 Indexes

Each purge request will bump up `update_seq` of the database, so that each secondary index is also updated in order to apply the purge requests to maintain consistency within the main database.

8.5.6 Config Settings

These settings can be updated in the default.ini or local.ini:

Field	Description	Default
max_document_id_numb	Allowed maximum number of documents in one purge request	100
max_revisions_number	Allowed maximum number of accumulated revisions in one purge request	1000
allowed_purge_seq_lag	Beside purged_infos_limit, allowed additional buffer to store purge requests	100
index_lag_warn_seconds	Allowed durations when index is not updated for local purge checkpoint document	86400

During a database compaction, we check all checkpoint purge docs. A client (an index or internal replication job) is allowed to have the last reported `purge_seq` to be smaller than the current database shard's `purge_seq` by the value of `(purged_infos_limit + allowed_purge_seq_lag)`. If the client's `purge_seq` is even smaller, and the client has not checkpointed within `index_lag_warn_seconds`, it prevents compaction of purge trees and we have to issue the following log warning for this client:

```
Purge checkpoint '_local/purge-mrview-9152d15c12011288629bcffba7693fd4'
not updated in 86400 seconds in
<<"shards/00000000-1fffffff/testdb12.1491979089">>
```

If this type of log warning occurs, check the client to see why the processing of purge requests is stalled in it.

There is a mapping relationship between a design document of indexes and local checkpoint docs. If a design document of indexes is updated or deleted, the corresponding local checkpoint document should be also automatically deleted. But in an unexpected case, when a design doc was updated/deleted, but its checkpoint document still exists in a database, the following warning will be issued:

```
"Invalid purge doc '<<"_design/bar">>' on database
<<"shards/00000000-1fffffff/testdb12.1491979089">>
with purge_seq '50'"
```

If this type of log warning occurs, remove the local purge doc from a database.

8.6 TLS Erlang Distribution

The main purpose is specifically to allow using TLS for Erlang distribution between nodes, with the ability to connect to some nodes using TCP as well. TLS distribution will enhance data security during data migration between nodes.

This section describes how to enable TLS distribution for additional verification and security.

Reference: Using TLS for Erlang Distribution

8.6.1 Generate Certificate

To distribute using TLS, appropriate certificates need to be provided. In the following example (couch_dist.conf), the cert.pem certificate must be trusted by a root certificate known to the server, and the erlserver.pem file contains the "certificate" and its "private key".

```
[{server,
  [{cacertfile, "</absolute_path/to/ca-cert.pem>"},
   {certfile,   "</absolute_path/to/erlserver.pem>"},
   {secure_renegotiate, true},
   {verify, verify_peer},
   {fail_if_no_peer_cert, true}]},
 {client,
  [{cacertfile, "</absolute_path/to/ca-cert.pem>"},
   {keyfile,    "</absolute_path/to/key.pem>"},
   {certfile,   "</absolute_path/to/cert.pem>"},
   {secure_renegotiate, true},
   {verify, verify_peer}]}].
```

You can use `{verify, verify_peer}` to enable verification, but it requires appropriate certificates to verify.

This is an example of generating certificates.

```
$ git clone https://github.com/rnewson/elixir-certs
$ cd elixir-certs
$ ./certs self-signed \
    --out-cert ca-cert.pem --out-key ca-key.pem \
    --template root-ca \
    --subject "/CN=CouchDB Root CA"
$./certs create-cert \
    --issuer-cert ca-cert.pem --issuer-key ca-key.pem \
    --out-cert cert.pem --out-key key.pem \
    --template server \
    --subject "/CN=<hostname>"
$ cat key.pem cert.pem >erlserver.pem
```

Note:

- The above examples are **not** an endorsement of specific expiration limits, key sizes, or algorithms.
- If option `verify_peer` is set, the `server_name_indication` option should also be specified.
- The option `{fail_if_no_peer_cert, true}` should only be used on the server side in OTP 26, for previous versions it can be specified both on the server side and client side.
- When generating certificates, make sure Common Name (FQDN) should be different in CA certificate and certificate. Also, FQDN in the certificate should be the same as the hostname.

8.6.2 Config Settings

To enable TLS distribution, make sure to set custom parameters in `vm.args`.

```
# Don't forget to override the paths to point to your cert and conf file!

-proto_dist couch
-couch_dist no_tls \"clouseau@127.0.0.1\"
-ssl_dist_optfile </absolute_path/to/couch_dist.conf>
```

Note:

- The default value of `no_tls` is `false`. If the user does not set any `no_tls` flag, all nodes will use `TCP`.
- To ensure "search" works, make sure to set `no_tls` option for the `clouseau` node. By default, this will be `"clouseau@127.0.0.1"`.

The `no_tls` flag can have these values:

1. Use `TLS` only, set to `false` (default value), such as:

```
-couch_dist no_tls false
```

2. Use `TCP` only, set to `true`, such as:

```
-couch_dist no_tls true
```

3. Specify some nodes to use `TCP`, others to use `TLS`, such as:

```
# Specify node1 and node2 to use TCP, others use TLS

-couch_dist no_tls '"node1@127.0.0.1"'
-couch_dist no_tls \"node2@127.0.0.1\"
```

```
# Any nodes end with "@127.0.0.1" will use TCP, others use TLS

-couch_dist no_tls \"*@127.0.0.1\"
```

Note: **Asterisk(*)**: matches a sequence of zero or more occurrences of the regular expression.

Question mark(?): matches zero or one occurrences of the regular expression.

8.6.3 Connect to Remsh

Start Erlang using a remote shell connected to Node.

- If the node uses `TCP`:

```
$ ./remsh
```

- If the node uses `TLS`:

```
$ ./remsh -t </absolute_path/to/couch_dist.conf>
```

8.7 Troubleshooting CouchDB 3 with WeatherReport

8.7.1 Overview

WeatherReport is an OTP application and set of tools that diagnoses common problems which could affect a CouchDB version 3 node or cluster (version 4 or later is not supported). It is accessed via the `weatherreport` command line escript.

Here is a basic example of using `weatherreport` followed immediately by the command's output:

```
$ weatherreport --etc /path/to/etc
[warning] Cluster member node3@127.0.0.1 is not connected to this node. Please check␣
↪whether it is down.
```

8.7.2 Usage

For most cases, you can just run the `weatherreport` command as shown above. However, sometimes you might want to know some extra detail, or run only specific checks. For that, there are command-line options. Execute `weatherreport --help` to learn more about these options:

```
$ weatherreport --help
Usage: weatherreport [-c <path>] [-d <level>] [-e] [-h] [-l] [check_name ...]

  -c, --etc                 Path to the CouchDB configuration directory
  -d, --level               Minimum message severity level (default: notice)
  -l, --list                Describe available diagnostic tasks
  -e, --expert              Perform more detailed diagnostics
  -h, --help                Display help/usage
  check_name                A specific check to run
```

To get an idea of what checks will be run, use the *–list* option:

```
$ weatherreport --list
Available diagnostic checks:

  custodian            Shard safety/liveness checks
  disk                 Data directory permissions and atime
  internal_replication Check the number of pending internal replication jobs
  ioq                  Check the total number of active IOQ requests
  mem3_sync            Check there is a registered mem3_sync process
  membership           Cluster membership validity
  memory_use           Measure memory usage
  message_queues       Check for processes with large mailboxes
  node_stats           Check useful erlang statistics for diagnostics
  nodes_connected      Cluster node liveness
  process_calls        Check for large numbers of processes with the same current/
↪initial call
  process_memory       Check for processes with high memory usage
  safe_to_rebuild      Check whether the node can safely be taken out of service
  search               Check the local search node is responsive
  tcp_queues           Measure the length of tcp queues in the kernel
```

If you want all the gory details about what WeatherReport is doing, you can run the checks at a more verbose logging level with the `--level` option:

```
$ weatherreport --etc /path/to/etc --level debug
[debug] Not connected to the local cluster node, trying to connect. alive:false␣
↪connect_failed:undefined
[debug] Starting distributed Erlang.
[debug] Connected to local cluster node 'node1@127.0.0.1'.
[debug] Local RPC: mem3:nodes([]) [5000]
[debug] Local RPC: os:getpid([]) [5000]
[debug] Running shell command: ps -o pmem,rss -p 73905
[debug] Shell command output:
%MEM    RSS
0.3  25116

[debug] Local RPC: erlang:nodes([]) [5000]
[debug] Local RPC: mem3:nodes([]) [5000]
[warning] Cluster member node3@127.0.0.1 is not connected to this node. Please check␣
↪whether it is down.
[info] Process is using 0.3% of available RAM, totalling 25116 KB of real memory.
```

Most times you'll want to use the defaults, but any syslog severity name will do (from most to least verbose): `debug`, `info`, `notice`, `warning`, `error`, `critical`, `alert`, `emergency`.

Finally, if you want to run just a single diagnostic or a list of specific ones, you can pass their name(s):

```
$ weatherreport --etc /path/to/etc nodes_connected
[warning] Cluster member node3@127.0.0.1 is not connected to this node. Please check␣
↪whether it is down.
```

CHAPTER

NINE

MAINTENANCE

9.1 Compaction

The *compaction* operation is a way to reduce disk space usage by removing unused and old data from database or view index files. This operation is very similar to the *vacuum* (SQLite ex.) operation available for other database management systems.

During compaction, CouchDB re-creates the database or view in a new file with the `.compact` extension. As this requires roughly twice the disk storage, CouchDB first checks for available disk space before proceeding.

When all actual data is successfully transferred to the newly compacted file, CouchDB transparently swaps the compacted file into service, and removes the old database or view file.

Since CouchDB 2.1.1, automated compaction is enabled by default, and is described in the next section. It is still possible to trigger manual compaction if desired or necessary. This is described in the subsequent sections.

9.1.1 Automatic Compaction

CouchDB's automatic compaction daemon, internally known as "smoosh", will trigger compaction jobs for both databases and views based on configurable thresholds for the sparseness of a file and the total amount of space that can be recovered.

Channels

Smoosh works using the concept of channels. A channel is essentially a queue of pending compactions. There are separate sets of active channels for databases and views. Each channel is assigned a configuration which defines whether a compaction ends up in the channel's queue and how compactions are prioritized within that queue.

Smoosh takes each channel and works through the compactions queued in each in priority order. Each channel is processed concurrently, so the priority levels only matter within a given channel. Each channel has an assigned number of active compactions, which defines how many compactions happen for that channel in parallel. For example, a cluster with a lot of database churn but few views might require more active compactions in the database channel(s).

It's important to remember that a channel is local to a CouchDB node; that is, each node maintains and processes an independent set of compactions. Channels are defined as either "ratio" channels or "slack" channels, depending on the type of algorithm used for prioritization:

- Ratio: uses the ratio of sizes.file / sizes.active as its driving calculation. The result X must be greater than some configurable value Y for a compaction to be added to the queue. Compactions are then prioritised for higher values of X.
- Slack: uses the difference of sizes.file - sizes.active as its driving calculation. The result X must be greater than some configurable value Y for a compaction to be added to the queue. Compactions are prioritised for higher values of X.

In both cases, Y is set using the `min_priority` configuration variable. CouchDB ships with four channels pre-configured: one channel of each type for databases, and another one for views.

Channel Configuration

Channels are defined using `[smoosh.{channel-name}]` configuration blocks, and activated by naming the channel in the `db_channels` or `view_channels` configuration setting in the `[smoosh]` block. The default configuration is

```
[smoosh]
db_channels = upgrade_dbs,ratio_dbs,slack_dbs
view_channels = upgrade_views,ratio_views,slack_views
cleanup_channels = index_cleanup

[smoosh.ratio_dbs]
priority = ratio
min_priority = 2.0

[smoosh.ratio_views]
priority = ratio
min_priority = 2.0

[smoosh.slack_dbs]
priority = slack
min_priority = 536870912

[smoosh.slack_views]
priority = slack
min_priority = 536870912
```

The "upgrade" and "cleanup_channels" are special system channels. The "upgrade" ones check whether the `disk_format_version` for the file matches the current version, and enqueue the file for compaction (which has the side effect of upgrading the file format) if that's not the case. In addition to that, the `upgrade_views` will enqueue views for compaction after the collation (libicu) library is upgraded. The "index_cleanup" channel is used for scheduling jobs used to remove stale index files and purge _local checkpoint document after design documents are updated.

Here are several additional properties that can be configured for each channel; these are documented in the *configuration API*

Scheduling Windows

Each compaction channel can be configured to run only during certain hours of the day. The channel-specific `from`, `to`, and `strict_window` configuration settings control this behavior. For example

```
[smoosh.overnight_channel]
from = 20:00
to = 06:00
strict_window = true
```

where `overnight_channel` is the name of the channel you want to configure.

Note: CouchDB determines time via the UTC (GMT) timezone, so these settings must be expressed as UTC (GMT).

The `strict_window` setting will cause the compaction daemon to suspend all active compactions in this channel when exiting the window, and resume them when re-entering. If `strict_window` is left at its default of false, the active compactions will be allowed to complete but no new compactions will be started.

Note: When a channel is created, a 60s timer is started to check if the channel should be processing any compactions based on the time window defined in your config.

The channel is set to pending and after 60s it checks if it should be running at all and is set to paused if not. At the end of the check another 60s timer is started to schedule another check.

Eventually, when in the time window, it starts processing compactions. But since it will continue running a check every 60s running compaction processes will be suspended when exiting the time window and resume them when re-entering the window.

This means that for the first 60s after exiting the time window, or when a channel is created and you are outside the time window, compactions are run for up to 60s.This is different to the behavior of the old compaction daemon which would cancel the compactions outright.

Migration Guide

Previous versions of CouchDB shipped with a simpler compaction daemon. The configuration system for the new daemon is not backwards-compatible with the old one, so users with customized compaction configurations will need to port them to the new setup. The old daemon's compaction rules configuration looked like

```
[compaction_daemon]
min_file_size = 131072
check_interval = 3600
snooze_period_ms = 3000

[compactions]
mydb = [{db_fragmentation, "70%"}, {view_fragmentation, "60%"}, {parallel_view_
↪compaction, true}]
_default = [{db_fragmentation, "50%"}, {view_fragmentation, "55%"}, {from, "20:00"},
↪{to, "06:00"}, {strict_window, true}]
```

Many of the elements of this configuration can be ported over to the new system. Examining each in detail:

- `min_file_size` is now configured on a per-channel basis using the min_size config setting.
- `db_fragmentation` is equivalent to configuring a priority = ratio channel with min_priority set to 1.0 / (1 - db_fragmentation/100) and then listing that channel in the [smoosh] db_channels config setting.
- `view_fragmention` is likewise equivalent to configuring a priority = ratio channel with min_priority set to 1.0 / (1 - view_fragmentation/100) and then listing that channel in the [smoosh] view_channels config setting.
- `from` / `to` / `strict_window`: each of these settings can be applied on a per-channel basis in the new daemon. The one behavior change is that the new daemon will suspend compactions upon exiting the allowed window instead of canceling them outright, and resume them when re-entering.
- `parallel_view_compaction`: each compaction channel has a concurrency setting that controls how many compactions will execute in parallel in that channel. The total parallelism is the sum of the concurrency settings of all active channels. This is a departure from the previous behavior, in which the daemon would only focus on one database and/or its views (depending on the value of this flag) at a time.

The `check_interval` and `snooze_period_ms` settings are obsolete in the event-driven design of the new daemon. The new daemon does not support setting database-specific thresholds as in the `mydb` setting above. Rather, channels can be configured to focus on specific classes of files: large databases, small view indexes, and so on. Most cases of named database compaction rules can be expressed using properties of those databases and/or their associated views.

9.1.2 Manual Database Compaction

Database compaction compresses the database file by removing unused file sections created during updates. Old documents revisions are replaced with small amount of metadata called `tombstone` which are used for conflicts resolution during replication. The number of stored revisions (and their `tombstones`) can be configured by using the *_revs_limit* URL endpoint.

Compaction can be manually triggered per database and runs as a background task. To start it for specific database there is need to send HTTP *POST /{db}/_compact* sub-resource of the target database:

```
curl -H "Content-Type: application/json" -X POST http://adm:pass@localhost:5984/my_db/
↪_compact
```

On success, HTTP status 202 Accepted is returned immediately:

```
HTTP/1.1 202 Accepted
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: text/plain; charset=utf-8
Date: Wed, 19 Jun 2013 09:43:52 GMT
Server: CouchDB (Erlang/OTP)
```

```
{"ok":true}
```

Although the request body is not used you must still specify Content-Type header with *application/json* value for the request. If you don't, you will be aware about with HTTP status 415 Unsupported Media Type response:

```
HTTP/1.1 415 Unsupported Media Type
Cache-Control: must-revalidate
Content-Length: 78
Content-Type: application/json
Date: Wed, 19 Jun 2013 09:43:44 GMT
Server: CouchDB (Erlang/OTP)

{"error":"bad_content_type","reason":"Content-Type must be application/json"}
```

When the compaction is successful started and running it is possible to get information about it via *database information resource*:

```
curl http://adm:pass@localhost:5984/my_db
```

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 246
Content-Type: application/json
Date: Wed, 19 Jun 2013 16:51:20 GMT
Server: CouchDB (Erlang/OTP)

{
    "committed_update_seq": 76215,
    "compact_running": true,
    "db_name": "my_db",
    "disk_format_version": 6,
    "doc_count": 5091,
    "doc_del_count": 0,
    "instance_start_time": "0",
    "purge_seq": 0,
    "sizes": {
```

(continues on next page)

(continued from previous page)

```
      "active": 3787996,
      "disk": 17703025,
      "external": 4763321
    },
    "update_seq": 76215
}
```

Note that `compact_running` field is `true` indicating that compaction is actually running. To track the compaction progress you may query the *_active_tasks* resource:

```
curl http://adm:pass@localhost:5984/_active_tasks
```

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 175
Content-Type: application/json
Date: Wed, 19 Jun 2013 16:27:23 GMT
Server: CouchDB (Erlang/OTP)

[
    {
        "changes_done": 44461,
        "database": "my_db",
        "pid": "<0.218.0>",
        "progress": 58,
        "started_on": 1371659228,
        "total_changes": 76215,
        "type": "database_compaction",
        "updated_on": 1371659241
    }
]
```

9.1.3 Manual View Compaction

Views also need compaction. Unlike databases, views are compacted by groups per *design document*. To start their compaction, send the HTTP *POST /{db}/_compact/{ddoc}* request:

Design Document:

```
{
    "_id": "_design/ddoc-name",
    "views": {
        "view-name": {
            "map": "function(doc) { emit(doc.key, doc.value) }"
        }
    }
}
```

```
curl -H "Content-Type: application/json" -X POST http://adm:pass@localhost:5984/
↪dbname/_compact/ddoc-name
```

```
{"ok":true}
```

This compacts the view index from the current version of the specified design document. The HTTP response code is 202 Accepted (like *compaction for databases*) and a compaction background task will be created.

Views cleanup

View indexes on disk are named after their *MD5* hash of the view definition. When you change a view, old indexes remain on disk. To clean up all outdated view indexes (files named after the MD5 representation of views, that does not exist anymore) you can trigger a *view cleanup*:

```
curl -H "Content-Type: application/json" -X POST http://adm:pass@localhost:5984/
↪dbname/_view_cleanup
```

```
{"ok":true}
```

9.2 Performance

With up to tens of thousands of documents you will generally find CouchDB to perform well no matter how you write your code. Once you start getting into the millions of documents you need to be a lot more careful.

9.2.1 Disk I/O

File Size

The smaller your file size, the less *I/O* operations there will be, the more of the file can be cached by CouchDB and the operating system, the quicker it is to replicate, backup etc. Consequently you should carefully examine the data you are storing. For example it would be silly to use keys that are hundreds of characters long, but your program would be hard to maintain if you only used single character keys. Carefully consider data that is duplicated by putting it in views.

Disk and File System Performance

Using faster disks, striped RAID arrays and modern file systems can all speed up your CouchDB deployment. However, there is one option that can increase the responsiveness of your CouchDB server when disk performance is a bottleneck. From the Erlang documentation for the file module:

> On operating systems with thread support, it is possible to let file operations be performed in threads of their own, allowing other Erlang processes to continue executing in parallel with the file operations. See the command line flag +A in erl(1).

Setting this argument to a number greater than zero can keep your CouchDB installation responsive even during periods of heavy disk utilization. The easiest way to set this option is through the `ERL_FLAGS` environment variable. For example, to give Erlang four threads with which to perform I/O operations add the following to `(prefix)/etc/defaults/couchdb` (or equivalent):

```
export ERL_FLAGS="+A 4"
```

9.2.2 System Resource Limits

One of the problems that administrators run into as their deployments become large are resource limits imposed by the system and by the application configuration. Raising these limits can allow your deployment to grow beyond what the default configuration will support.

CouchDB Configuration Options

max_dbs_open

In your *configuration* (local.ini or similar) familiarize yourself with the *couchdb/max_dbs_open*:

```
[couchdb]
max_dbs_open = 100
```

This option places an upper bound on the number of databases that can be open at one time. CouchDB reference counts database accesses internally and will close idle databases when it must. Sometimes it is necessary to keep more than the default open at once, such as in deployments where many databases will be continuously replicating.

Erlang

Even if you've increased the maximum connections CouchDB will allow, the Erlang runtime system will not allow more than 65536 connections by default. Adding the following directive to `(prefix)/etc/vm.args` (or equivalent) will increase this limit (in this case to 102400):

```
+Q 102400
```

Note that on Windows, Erlang will not actually increase the file descriptor limit past 8192 (i.e. the system header–defined value of `FD_SETSIZE`). On macOS, the limit may be as low as 1024. See this tip for a possible workaround and this thread for a deeper explanation.

Maximum open file descriptors (ulimit)

In general, modern UNIX-like systems can handle very large numbers of file handles per process (e.g. 100000) without problem. Don't be afraid to increase this limit on your system.

The method of increasing these limits varies, depending on your init system and particular OS release. The default value for many OSes is 1024 or 4096. On a system with many databases or many views, CouchDB can very rapidly hit this limit.

For systemd-based Linuxes (such as CentOS/RHEL 7, Ubuntu 16.04+, Debian 8 or newer), assuming you are launching CouchDB from systemd, you must override the upper limit via editing the override file. The best practice for this is via the `systemctl edit couchdb` command. Add these lines to the file in the editor:

```
[Service]
LimitNOFILE=65536
```

...or whatever value you like. To increase this value higher than 65536, you must also add the Erlang `+Q` parameter to your `etc/vm.args` file by adding the line:

```
+Q 102400
```

The old `ERL_MAX_PORTS` environment variable is ignored by the version of Erlang supplied with CouchDB.

If your system is set up to use the Pluggable Authentication Modules (PAM), and you are **not** launching CouchDB from systemd, increasing this limit is straightforward. For example, creating a file named `/etc/security/limits.d/100-couchdb.conf` with the following contents will ensure that CouchDB can open up to 65536 file descriptors at once:

```
#<domain>    <type>    <item>    <value>
couchdb      hard      nofile    65536
couchdb      soft      nofile    65536
```

If you are using our Debian/Ubuntu sysvinit script (`/etc/init.d/couchdb`), you also need to raise the limits for the root user:

```
#<domain>    <type>    <item>    <value>
root         hard      nofile    65536
root         soft      nofile    65536
```

You may also have to edit the `/etc/pam.d/common-session` and `/etc/pam.d/common-session-noninteractive` files to add the line:

```
session required pam_limits.so
```

if it is not already present.

If your system does not use PAM, a *ulimit* command is usually available for use in a custom script to launch CouchDB with increased resource limits. Typical syntax would be something like *ulimit -n 65536*.

9.2.3 Network

There is latency overhead making and receiving each request/response. In general you should do your requests in batches. Most APIs have some mechanism to do batches, usually by supplying lists of documents or keys in the request body. Be careful what size you pick for the batches. The larger batch requires more time your client has to spend encoding the items into JSON and more time is spent decoding that number of responses. Do some benchmarking with your own configuration and typical data to find the sweet spot. It is likely to be between one and ten thousand documents.

If you have a fast I/O system then you can also use concurrency - have multiple requests/responses at the same time. This mitigates the latency involved in assembling JSON, doing the networking and decoding JSON.

As of CouchDB 1.1.0, users often report lower write performance of documents compared to older releases. The main reason is that this release ships with the more recent version of the HTTP server library MochiWeb, which by default sets the TCP socket option SO_NODELAY to false. This means that small data sent to the TCP socket, like the reply to a document write request (or reading a very small document), will not be sent immediately to the network - TCP will buffer it for a while hoping that it will be asked to send more data through the same socket and then send all the data at once for increased performance. This TCP buffering behaviour can be disabled via *`httpd/socket_options`*:

```
[httpd]
socket_options = [{nodelay, true}]
```

See also:

Bulk *load* and *store* API.

Connection limit

MochiWeb handles CouchDB requests. The default maximum number of connections is 2048. To change this limit, use the `server_options` configuration variable. `max` indicates maximum number of connections.

```
[chttpd]
server_options = [{backlog, 128}, {acceptor_pool_size, 16}, {max, 4096}]
```

9.2.4 CouchDB

DELETE operation

When you DELETE a document the database will create a new revision which contains the `_id` and `_rev` fields as well as the *_deleted* flag. This revision will remain even after a *database compaction* so that the deletion can be replicated. Deleted documents, like non-deleted documents, can affect view build times, PUT and DELETE request times, and the size of the database since they increase the size of the B+Tree. You can see the number of deleted documents in `database information`. If your use case creates lots of deleted documents (for example, if you are storing short-term data like log entries, message queues, etc), you might want to periodically switch to a new database and delete the old one (once the entries in it have all expired).

Document's ID

The db file size is derived from your document and view sizes but also on a multiple of your `_id` sizes. Not only is the `_id` present in the document, but it and parts of it are duplicated in the binary tree structure CouchDB uses to navigate the file to find the document in the first place. As a real world example for one user switching from 16 byte ids to 4 byte ids made a database go from 21GB to 4GB with 10 million documents (the raw JSON text when from 2.5GB to 2GB).

Inserting with sequential (and at least sorted) ids is faster than random ids. Consequently you should consider generating ids yourself, allocating them sequentially and using an encoding scheme that consumes fewer bytes. For example, 8 bytes will take 16 hex digits to represent, and those same 8 bytes can be encoded in only 11 digits/chars in base64url (no padding).

9.2.5 Views

Views Generation

Views with the JavaScript query server are extremely slow to generate when there are a non-trivial number of documents to process. The generation process won't even saturate a single CPU let alone your I/O. The cause is the latency involved in the CouchDB server and separate *couchjs* query server, dramatically indicating how important it is to take latency out of your implementation.

You can let view access be "stale" but it isn't practical to determine when that will occur giving you a quick response and when views will be updated which will take a long time. (A 10 million document database took about 10 minutes to load into CouchDB but about 4 hours to do view generation).

In a cluster, "stale" requests are serviced by a fixed set of shards in order to present users with consistent results between requests. This comes with an availability trade-off - the fixed set of shards might not be the most responsive / available within the cluster. If you don't need this kind of consistency (e.g. your indexes are relatively static), you can tell CouchDB to use any available replica by specifying `stable=false&update=false` instead of `stale=ok`, or `stable=false&update=lazy` instead of `stale=update_after`.

View information isn't replicated - it is rebuilt on each database so you can't do the view generation on a separate sever.

Built-In Reduce Functions

If you're using a very simple view function that only performs a sum or count reduction, you can call native Erlang implementations of them by simply writing `_sum` or `_count` in place of your function declaration. This will speed up things dramatically, as it cuts down on IO between CouchDB and the *JavaScript query server*. For example, as mentioned on the mailing list, the time for outputting an (already indexed and cached) view with about 78,000 items went down from 60 seconds to 4 seconds.

Before:

```
{
    "_id": "_design/foo",
    "views": {
        "bar": {
            "map": "function (doc) { emit(doc.author, 1); }",
            "reduce": "function (keys, values, rereduce) { return sum(values); }"
        }
    }
}
```

After:

```
{
    "_id": "_design/foo",
    "views": {
        "bar": {
            "map": "function (doc) { emit(doc.author, 1); }",
            "reduce": "_sum"
        }
    }
}
```

See also:

Built-in Reduce Functions

9.3 Backing up CouchDB

CouchDB has three different types of files it can create during runtime:

- Database files (including secondary indexes)
- Configuration files (`*.ini`)
- Log files (if configured to log to disk)

Below are strategies for ensuring consistent backups of all of these files.

9.3.1 Database Backups

The simplest and easiest approach for CouchDB backup is to use *CouchDB replication* to another CouchDB installation. You can choose between *normal (one-shot) or continuous replications* depending on your need.

However, you can also copy the actual `.couch` files from the CouchDB data directory (by default, `data/`) at any time, without problem. CouchDB's append-only storage format for both databases and secondary indexes ensures that this will work without issue.

To ensure reliability of backups, it is recommended that you *back up secondary indexes* (stored under `data/.shards`) *prior to backing up the main database files* (stored under `data/shards` as well as the system-level databases at the parent `data/` directory). This is because CouchDB will automatically handle views/secondary

indexes that are slightly out of date by updating them on the next read access, but views or secondary indexes that are *newer* than their associated databases will trigger a *full rebuild of the index*. This can be a very costly and time-consuming operation, and can impact your ability to recover quickly in a disaster situation.

On supported operating systems/storage environments, you can also make use of storage snapshots. These have the advantage of being near-instantaneous when working with block storage systems such as ZFS or LVM or Amazon EBS. When using snapshots at the block-storage level, be sure to quiesce the file system with an OS-level utility such as Linux's fsfreeze if necessary. If unsure, consult your operating system's or cloud provider's documentation for more detail.

9.3.2 Configuration Backups

CouchDB's *configuration system* stores data in `.ini` files under the configuration directory (by default, `etc/`). If changes are made to the configuration at runtime, the very last file in the configuration chain will be updated with the changes.

Simple back up the entire `etc/` directory to ensure a consistent configuration after restoring from backup.

If no changes to the configuration are made at runtime through the HTTP API, and all configuration files are managed by a configuration management system (such as Ansible or Chef), there is no need to backup the configuration directory.

9.3.3 Log Backups

If *configured to log to a file*, you may want to back up the log files written by CouchDB. Any backup solution for these files works.

Under UNIX-like systems, if using log rotation software, a copy-then-truncate approach is necessary. This will truncate the original log file to zero size in place after creating a copy. CouchDB does not recognize any signal to be told to close its log file and create a new one. Because of this, and because of differences in how file handles function, there is no straightforward log rotation solution under Microsoft Windows other than periodic restarts of the CouchDB process.

CHAPTER

TEN

FAUXTON

10.1 Fauxton Setup

Fauxton is included with CouchDB 2.0, so make sure CouchDB is running, then go to:

```
http://127.0.0.1:5984/_utils/
```

You can also upgrade to the latest version of Fauxton by using npm:

```
$ npm install -g fauxton
$ fauxton
```

(Recent versions of node.js and npm are required.)

10.1.1 Fauxton Visual Guide

You can find the Visual Guide here:
http://couchdb.apache.org/fauxton-visual-guide

10.1.2 Development Server

Recent versions of node.js and npm are required.

Using the dev server is the easiest way to use Fauxton, specially when developing for it:

```
$ git clone https://github.com/apache/couchdb-fauxton.git
$ npm install && npm run dev
```

10.1.3 Understanding Fauxton Code layout

Each bit of functionality is its own separate module or addon.

All core modules are stored under *app/module* and any addons that are optional are under *app/addons*.

We use backbone.js and Backbone.layoutmanager quite heavily, so best to get an idea how they work. Its best at this point to read through a couple of the modules and addons to get an idea of how they work.

Two good starting points are *app/addon/config* and *app/modules/databases*.

Each module must have a *base.js* file, this is read and compile when Fauxton is deployed.

The *resource.js* file is usually for your `Backbone.Models` and `Backbone.Collections`, *view.js* for your `Backbone.Views`.

The *routes.js* is used to register a url path for your view along with what layout, data, breadcrumbs and api point is required for the view.

ToDo items

Checkout *JIRA* or GitHub Issues for a list of items to do.

CHAPTER

ELEVEN

EXPERIMENTAL FEATURES

This is a list of experimental features in CouchDB. They are included in a release because the development team is requesting feedback from the larger developer community. As such, please play around with these features and send us feedback, thanks!

Use at your own risk! Do not rely on these features for critical applications.

11.1 Content-Security-Policy (CSP) Header Support for /_utils (Fauxton)

This will just work with Fauxton. You can enable it in your config: you can enable the feature in general and change the default header that is sent for everything in /_utils.

```
[csp]
enable = true
```

Then restart CouchDB.

11.2 Nouveau Server (new Apache Lucene integration)

Enable nouveau in config and run the Java service.

```
[nouveau]
enable = true
```

Have fun!

CHAPTER

TWELVE

API REFERENCE

The components of the API URL path help determine the part of the CouchDB server that is being accessed. The result is the structure of the URL request both identifies and effectively describes the area of the database you are accessing.

As with all URLs, the individual components are separated by a forward slash.

As a general rule, URL components and JSON fields starting with the _ (underscore) character represent a special component or entity within the server or returned object. For example, the URL fragment `/_all_dbs` gets a list of all of the databases in a CouchDB instance.

This reference is structured according to the URL structure, as below.

12.1 API Basics

The CouchDB API is the primary method of interfacing to a CouchDB instance. Requests are made using HTTP and requests are used to request information from the database, store new data, and perform views and formatting of the information stored within the documents.

Requests to the API can be categorised by the different areas of the CouchDB system that you are accessing, and the HTTP method used to send the request. Different methods imply different operations, for example retrieval of information from the database is typically handled by the `GET` operation, while updates are handled by either a `POST` or `PUT` request. There are some differences between the information that must be supplied for the different methods. For a guide to the basic HTTP methods and request structure, see *Request Format and Responses*.

For nearly all operations, the submitted data, and the returned data structure, is defined within a JavaScript Object Notation (JSON) object. Basic information on the content and data types for JSON are provided in *JSON Basics*.

Errors when accessing the CouchDB API are reported using standard HTTP Status Codes. A guide to the generic codes returned by CouchDB are provided in *HTTP Status Codes*.

When accessing specific areas of the CouchDB API, specific information and examples on the HTTP methods and request, JSON structures, and error codes are provided.

12.1.1 Request Format and Responses

CouchDB supports the following HTTP request methods:

- `GET`

  Request the specified item. As with normal HTTP requests, the format of the URL defines what is returned. With CouchDB this can include static items, database documents, and configuration and statistical information. In most cases the information is returned in the form of a JSON document.

- `HEAD`

  The `HEAD` method is used to get the HTTP header of a `GET` request without the body of the response.

- `POST`

  Upload data. Within CouchDB `POST` is used to set values, including uploading documents, setting document values, and starting certain administration commands.

- `PUT`

  Used to put a specified resource. In CouchDB `PUT` is used to create new objects, including databases, documents, views and design documents.

- `DELETE`

  Deletes the specified resource, including documents, views, and design documents.

- `COPY`

  A special method that can be used to copy documents and objects.

If you use an unsupported HTTP request type with an URL that does not support the specified type then a `405 - Method Not Allowed` will be returned, listing the supported HTTP methods. For example:

```
{
    "error":"method_not_allowed",
    "reason":"Only GET,HEAD allowed"
}
```

12.1.2 HTTP Headers

Because CouchDB uses HTTP for all communication, you need to ensure that the correct HTTP headers are supplied (and processed on retrieval) so that you get the right format and encoding. Different environments and clients will be more or less strict on the effect of these HTTP headers (especially when not present). Where possible you should be as specific as possible.

Request Headers

- `Accept`

  Specifies the list of accepted data types to be returned by the server (i.e. that are accepted/understandable by the client). The format should be a list of one or more MIME types, separated by colons.

  For the majority of requests the definition should be for JSON data (`application/json`). For attachments you can either specify the MIME type explicitly, or use `*/*` to specify that all file types are supported. If the `Accept` header is not supplied, then the `*/*` MIME type is assumed (i.e. client accepts all formats).

  The use of `Accept` in queries for CouchDB is not required, but is highly recommended as it helps to ensure that the data returned can be processed by the client.

  If you specify a data type using the `Accept` header, CouchDB will honor the specified type in the `Content-type` header field returned. For example, if you explicitly request `application/json` in the `Accept` of a request, the returned HTTP headers will use the value in the returned `Content-type` field.

  For example, when sending a request without an explicit `Accept` header, or when specifying `*/*`:

  ```
  GET /recipes HTTP/1.1
  Host: couchdb:5984
  Accept: */*
  ```

  The returned headers are:

  ```
  HTTP/1.1 200 OK
  Server: CouchDB (Erlang/OTP)
  Date: Thu, 13 Jan 2011 13:39:34 GMT
  ```

(continues on next page)

(continued from previous page)

```
Content-Type: text/plain;charset=utf-8
Content-Length: 227
Cache-Control: must-revalidate
```

Note: The returned content type is `text/plain` even though the information returned by the request is in JSON format.

Explicitly specifying the `Accept` header:

```
GET /recipes HTTP/1.1
Host: couchdb:5984
Accept: application/json
```

The headers returned include the `application/json` content type:

```
HTTP/1.1 200 OK
Server: CouchDB (Erlang/OTP)
Date: Thu, 13 Jan 2013 13:40:11 GMT
Content-Type: application/json
Content-Length: 227
Cache-Control: must-revalidate
```

- `Content-type`

  Specifies the content type of the information being supplied within the request. The specification uses MIME type specifications. For the majority of requests this will be JSON (`application/json`). For some settings the MIME type will be plain text. When uploading attachments it should be the corresponding MIME type for the attachment or binary (`application/octet-stream`).

  The use of the `Content-type` on a request is highly recommended.

Response Headers

Response headers are returned by the server when sending back content and include a number of different header fields, many of which are standard HTTP response header and have no significance to CouchDB operation. The list of response headers important to CouchDB are listed below.

- `Cache-control`

  The cache control HTTP response header provides a suggestion for client caching mechanisms on how to treat the returned information. CouchDB typically returns the `must-revalidate`, which indicates that the information should be revalidated if possible. This is used to ensure that the dynamic nature of the content is correctly updated.

- `Content-length`

  The length (in bytes) of the returned content.

- `Content-type`

  Specifies the MIME type of the returned data. For most request, the returned MIME type is `text/plain`. All text is encoded in Unicode (UTF-8), and this is explicitly stated in the returned `Content-type`, as `text/plain;charset=utf-8`.

- `Etag`

  The `Etag` HTTP header field is used to show the revision for a document, or a view.

  ETags have been assigned to a map/reduce group (the collection of views in a single design document). Any change to any of the indexes for those views would generate a new ETag for all view URLs in a single design doc, even if that specific view's results had not changed.

Each `_view` URL has its own ETag which only gets updated when changes are made to the database that effect that index. If the index for that specific view does not change, that view keeps the original ETag head (therefore sending back `304 - Not Modified` more often).

- `Transfer-Encoding`

  If the response uses an encoding, then it is specified in this header field.

  `Transfer-Encoding: chunked` means that the response is sent in parts, a method known as chunked transfer encoding. This is used when CouchDB does not know beforehand the size of the data it will send (for example, the *changes feed*).

- `X-CouchDB-Body-Time`

  Time spent receiving the request body in milliseconds.

  Available when body content is included in the request.

- `X-Couch-Request-ID`

  Unique identifier for the request.

12.1.3 JSON Basics

The majority of requests and responses to CouchDB use the JavaScript Object Notation (JSON) for formatting the content and structure of the data and responses.

JSON is used because it is the simplest and easiest solution for working with data within a web browser, as JSON structures can be evaluated and used as JavaScript objects within the web browser environment. JSON also integrates with the server-side JavaScript used within CouchDB.

JSON supports the same basic types as supported by JavaScript, these are:

- Array - a list of values enclosed in square brackets. For example:

```
["one", "two", "three"]
```

- Boolean - a `true` or `false` value. You can use these strings directly. For example:

```
{ "value": true}
```

- Number - an integer or floating-point number.
- Object - a set of key/value pairs (i.e. an associative array, or hash). The key must be a string, but the value can be any of the supported JSON values. For example:

```
{
    "servings" : 4,
    "subtitle" : "Easy to make in advance, and then cook when ready",
    "cooktime" : 60,
    "title" : "Chicken Coriander"
}
```

  In CouchDB, the JSON object is used to represent a variety of structures, including the main CouchDB document.

- String - this should be enclosed by double-quotes and supports Unicode characters and backslash escaping. For example:

```
"A String"
```

Parsing JSON into a JavaScript object is supported through the `JSON.parse()` function in JavaScript, or through various libraries that will perform the parsing of the content into a JavaScript object for you. Libraries for parsing and generating JSON are available in many languages, including Perl, Python, Ruby, Erlang and others.

> **Warning:** Care should be taken to ensure that your JSON structures are valid, invalid structures will cause CouchDB to return an HTTP status code of 500 (server error).

Number Handling

Developers and users new to computer handling of numbers often encounter surprises when expecting that a number stored in JSON format does not necessarily return as the same number as compared character by character.

Any numbers defined in JSON that contain a decimal point or exponent will be passed through the Erlang VM's idea of the "double" data type. Any numbers that are used in views will pass through the view server's idea of a number (the common JavaScript case means even integers pass through a double due to JavaScript's definition of a number).

Consider this document that we write to CouchDB:

```
{
    "_id":"30b3b38cdbd9e3a587de9b8122000cff",
    "number": 1.1
}
```

Now let's read that document back from CouchDB:

```
{
    "_id":"30b3b38cdbd9e3a587de9b8122000cff",
    "_rev":"1-f065cee7c3fd93aa50f6c97acde93030",
    "number":1.1000000000000000888
}
```

What happens is CouchDB is changing the textual representation of the result of decoding what it was given into some numerical format. In most cases this is an IEEE 754 double precision floating point number which is exactly what almost all other languages use as well.

What Erlang does a bit differently than other languages is that it does not attempt to pretty print the resulting output to use the shortest number of characters. For instance, this is why we have this relationship:

```
ejson:encode(ejson:decode(<<"1.1">>)).
<<"1.1000000000000000888">>
```

What can be confusing here is that internally those two formats decode into the same IEEE-754 representation. And more importantly, it will decode into a fairly close representation when passed through all major parsers that we know about.

While we've only been discussing cases where the textual representation changes, another important case is when an input value contains more precision than can actually represented in a double. (You could argue that this case is actually "losing" data if you don't accept that numbers are stored in doubles).

Here's a log for a couple of the more common JSON libraries that happen to be on the author's machine:

Ejson (CouchDB's current parser) at CouchDB sha 168a663b:

```
$ ./utils/run -i
Erlang R14B04 (erts-5.8.5) [source] [64-bit] [smp:2:2] [rq:2]
[async-threads:4] [hipe] [kernel-poll:true]

Eshell V5.8.5  (abort with ^G)
1> ejson:encode(ejson:decode(<<"1.01234567890123456789012345678901234567890">>)).
<<"1.012345678901234613 5">>
2> F = ejson:encode(ejson:decode(<<"1.01234567890123456789012345678901234567890">>)).
<<"1.012345678901234613 5">>
```

(continues on next page)

(continued from previous page)

```
3> ejson:encode(ejson:decode(F)).
<<"1.0123456789012346135">>
```

Node:

```
$ node -v
v0.6.15
$ node
JSON.stringify(JSON.parse("1.01234567890123456789012345678901234567890"))
'1.0123456789012346'
var f = JSON.stringify(JSON.parse("1.01234567890123456789012345678901234567890"))
undefined
JSON.stringify(JSON.parse(f))
'1.0123456789012346'
```

Python:

```
$ python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
import json
json.dumps(json.loads("1.01234567890123456789012345678901234567890"))
'1.0123456789012346'
f = json.dumps(json.loads("1.01234567890123456789012345678901234567890"))
json.dumps(json.loads(f))
'1.0123456789012346'
```

Ruby:

```
$ irb --version
irb 0.9.5(05/04/13)
require 'JSON'
=> true
JSON.dump(JSON.load("[1.01234567890123456789012345678901234567890]"))
=> "[1.01234567890123]"
f = JSON.dump(JSON.load("[1.01234567890123456789012345678901234567890]"))
=> "[1.01234567890123]"
JSON.dump(JSON.load(f))
=> "[1.01234567890123]"
```

Note: A small aside on Ruby, it requires a top level object or array, so I just wrapped the value. Should be obvious it doesn't affect the result of parsing the number though.

Spidermonkey:

```
$ js -h 2>&1 | head -n 1
JavaScript-C 1.8.5 2011-03-31
$ js
js> JSON.stringify(JSON.parse("1.01234567890123456789012345678901234567890"))
"1.0123456789012346"
js> var f = JSON.stringify(JSON.parse("1.01234567890123456789012345678901234567890"))
js> JSON.stringify(JSON.parse(f))
"1.0123456789012346"
```

As you can see they all pretty much behave the same except for Ruby actually does appear to be losing some

precision over the other libraries.

The astute observer will notice that ejson (the CouchDB JSON library) reported an extra three digits. While its tempting to think that this is due to some internal difference, its just a more specific case of the 1.1 input as described above.

The important point to realize here is that a double can only hold a finite number of values. What we're doing here is generating a string that when passed through the "standard" floating point parsing algorithms (ie, `strtod`) will result in the same bit pattern in memory as we started with. Or, slightly different, the bytes in a JSON serialized number are chosen such that they refer to a single specific value that a double can represent.

The important point to understand is that we're mapping from one infinite set onto a finite set. An easy way to see this is by reflecting on this:

```
1.0 == 1.00 == 1.000 = 1.(infinite zeros)
```

Obviously a computer can't hold infinite bytes so we have to decimate our infinitely sized set to a finite set that can be represented concisely.

The game that other JSON libraries are playing is merely:

"How few characters do I have to use to select this specific value for a double"

And that game has lots and lots of subtle details that are difficult to duplicate in C without a significant amount of effort (it took Python over a year to get it sorted with their fancy build systems that automatically run on a number of different architectures).

Hopefully we've shown that CouchDB is not doing anything "funky" by changing input. Its behaving the same as any other common JSON library does, its just not pretty printing its output.

On the other hand, if you actually are in a position where an IEEE-754 double is not a satisfactory data type for your numbers, then the answer as has been stated is to not pass your numbers through this representation. In JSON this is accomplished by encoding them as a string or by using integer types (although integer types can still bite you if you use a platform that has a different integer representation than normal, ie, JavaScript).

Further information can be found easily, including the Floating Point Guide, and David Goldberg's Reference.

Also, if anyone is really interested in changing this behavior, we're all ears for contributions to jiffy (which is theoretically going to replace ejson when we get around to updating the build system). The places we've looked for inspiration are TCL and Python. If you know a decent implementation of this float printing algorithm give us a holler.

12.1.4 HTTP Status Codes

With the interface to CouchDB working through HTTP, error codes and statuses are reported using a combination of the HTTP status code number, and corresponding data in the body of the response data.

A list of the error codes returned by CouchDB, and generic descriptions of the related errors are provided below. The meaning of different status codes for specific request types are provided in the corresponding API call reference.

- `200 - OK`

  Request completed successfully.

- `201 - Created`

  Document created successfully.

- `202 - Accepted`

  Request has been accepted, but the corresponding operation may not have completed. This is used for background operations, such as database compaction.

- `304 - Not Modified`

  The additional content requested has not been modified. This is used with the ETag system to identify the version of information returned.

- `400 - Bad Request`

  Bad request structure. The error can indicate an error with the request URL, path or headers. Differences in the supplied MD5 hash and content also trigger this error, as this may indicate message corruption.

- `401 - Unauthorized`

  The item requested was not available using the supplied authorization, or authorization was not supplied.

- `403 - Forbidden`

  The requested item or operation is forbidden.

- `404 - Not Found`

  The requested content could not be found. The content will include further information, as a JSON object, if available. The structure will contain two keys, `error` and `reason`. For example:

  ```
  {"error":"not_found","reason":"no_db_file"}
  ```

- `405 - Method Not Allowed`

  A request was made using an invalid HTTP request type for the URL requested. For example, you have requested a `PUT` when a `POST` is required. Errors of this type can also triggered by invalid URL strings.

- `406 - Not Acceptable`

  The requested content type is not supported by the server.

- `409 - Conflict`

  Request resulted in an update conflict.

- `412 - Precondition Failed`

  The request headers from the client and the capabilities of the server do not match.

- `413 - Request Entity Too Large`

  A document exceeds the configured *`couchdb/max_document_size`* value or the entire request exceeds the *`chttpd/max_http_request_size`* value.

- `415 - Unsupported Media Type`

  The content types supported, and the content type of the information being requested or submitted indicate that the content type is not supported.

- `416 - Requested Range Not Satisfiable`

  The range specified in the request header cannot be satisfied by the server.

- `417 - Expectation Failed`

  When sending documents in bulk, the bulk load operation failed.

- `500 - Internal Server Error`

  The request was invalid, either because the supplied JSON was invalid, or invalid information was supplied as part of the request.

- `503 - Service Unavailable`

  The request can't be serviced at this time, either because the cluster is overloaded, maintenance is underway, or some other reason. The request may be retried without changes, perhaps in a couple of minutes.

12.2 Server

The CouchDB server interface provides the basic interface to a CouchDB server for obtaining CouchDB information and getting and setting configuration information.

12.2.1 /

GET /

Accessing the root of a CouchDB instance returns meta information about the instance. The response is a JSON structure containing information about the server, including a welcome message and the version of the server.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET / HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 179
Content-Type: application/json
Date: Sat, 10 Aug 2013 06:33:33 GMT
Server: CouchDB (Erlang/OTP)

{
    "couchdb": "Welcome",
    "uuid": "85fb71bf700c17267fef77535820e371",
    "vendor": {
        "name": "The Apache Software Foundation",
        "version": "1.3.1"
    },
    "version": "1.3.1"
}
```

12.2.2 /_active_tasks

Changed in version 2.1.0: Because of how the scheduling replicator works, continuous replication jobs could be periodically stopped and then started later. When they are not running they will not appear in the `_active_tasks` endpoint

Changed in version 3.3: Added *"bulk_get_attempts"* and *"bulk_get_docs"* fields for replication jobs.

GET /_active_tasks

List of running tasks, including the task type, name, status and process ID. The result is a JSON array of the currently running tasks, with each task being described with a single object. Depending on operation type set of response object fields might be different.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **changes_done** (*number*) – Processed changes
- **database** (*string*) – Source database
- **pid** (*string*) – Process ID
- **progress** (*number*) – Current percentage progress
- **started_on** (*number*) – Task start time as unix timestamp
- **status** (*string*) – Task status message
- **task** (*string*) – Task name
- **total_changes** (*number*) – Total changes to process
- **type** (*string*) – Operation Type
- **updated_on** (*number*) – Unix timestamp of last operation update

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_active_tasks HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 1690
Content-Type: application/json
Date: Sat, 10 Aug 2013 06:37:31 GMT
```

(continues on next page)

(continued from previous page)

```
Server: CouchDB (Erlang/OTP)

[
    {
        "changes_done": 64438,
        "database": "mailbox",
        "pid": "<0.12986.1>",
        "progress": 84,
        "started_on": 1376116576,
        "total_changes": 76215,
        "type": "database_compaction",
        "updated_on": 1376116619
    },
    {
        "changes_done": 14443,
        "database": "mailbox",
        "design_document": "c9753817b3ba7c674d92361f24f59b9f",
        "pid": "<0.10461.3>",
        "progress": 18,
        "started_on": 1376116621,
        "total_changes": 76215,
        "type": "indexer",
        "updated_on": 1376116650
    },
    {
        "changes_done": 5454,
        "database": "mailbox",
        "design_document": "_design/meta",
        "pid": "<0.6838.4>",
        "progress": 7,
        "started_on": 1376116632,
        "total_changes": 76215,
        "type": "indexer",
        "updated_on": 1376116651
    },
    {
        "checkpointed_source_seq": 68585,
        "continuous": false,
        "doc_id": null,
        "doc_write_failures": 0,
        "bulk_get_attempts": 4524,
        "bulk_get_docs": 4524,
        "docs_read": 4524,
        "docs_written": 4524,
        "missing_revisions_found": 4524,
        "pid": "<0.1538.5>",
        "progress": 44,
        "replication_id": "9bc1727d74d49d9e157e260bb8bbd1d5",
        "revisions_checked": 4524,
        "source": "mailbox",
        "source_seq": 154419,
        "started_on": 1376116644,
        "target": "http://mailsrv:5984/mailbox",
        "type": "replication",
        "updated_on": 1376116651
    }
```

(continues on next page)

(continued from previous page)

```
]
```

12.2.3 /_all_dbs

GET /_all_dbs

Returns a list of all the databases in the CouchDB instance.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **descending** (*boolean*) – Return the databases in descending order by key. Default is `false`.
- **endkey** (*json*) – Stop returning databases when the specified key is reached.
- **end_key** (*json*) – Alias for `endkey` param
- **limit** (*number*) – Limit the number of the returned databases to the specified number.
- **skip** (*number*) – Skip this number of databases before starting to return the results. Default is `0`.
- **startkey** (*json*) – Return databases starting with the specified key.
- **start_key** (*json*) – Alias for `startkey`.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /_all_dbs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 52
Content-Type: application/json
Date: Sat, 10 Aug 2013 06:57:48 GMT
Server: CouchDB (Erlang/OTP)

[
   "_users",
   "contacts",
   "docs",
```

(continues on next page)

(continued from previous page)

```
    "invoices",
    "locations"
]
```

12.2.4 /_dbs_info

New in version 3.2.

GET /_dbs_info

Returns a list of all the databases information in the CouchDB instance.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **descending** (*boolean*) – Return databases information in descending order by key. Default is `false`.
- **endkey** (*json*) – Stop returning databases information when the specified key is reached.
- **end_key** (*json*) – Alias for `endkey` param
- **limit** (*number*) – Limit the number of the returned databases information to the specified number.
- **skip** (*number*) – Skip this number of databases before starting to return the results. Default is `0`.
- **startkey** (*json*) – Return databases information starting with the specified key.
- **start_key** (*json*) – Alias for `startkey`.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /_dbs_info HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Thu, 18 Nov 2021 14:37:35 GMT
Server: CouchDB (Erlang OTP/23)
```

(continues on next page)

(continued from previous page)

```
[
  {
    "key": "animals",
    "info": {
      "db_name": "animals",
      "update_seq": "52232",
      "sizes": {
        "file": 1178613587,
        "external": 1713103872,
        "active": 1162451555
      },
      "purge_seq": 0,
      "doc_del_count": 0,
      "doc_count": 52224,
      "disk_format_version": 6,
      "compact_running": false,
      "cluster": {
        "q": 8,
        "n": 3,
        "w": 2,
        "r": 2
      },
      "instance_start_time": "0"
    }
  }
]
```

New in version 2.2.

POST /_dbs_info

Returns information of a list of the specified databases in the CouchDB instance. This enables you to request information about multiple databases in a single request, in place of multiple *GET /{db}* requests.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Request JSON Object

- **keys** (*array*) – Array of database names to be requested

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Missing keys or exceeded keys in request

Request:

```
POST /_dbs_info HTTP/1.1
Accept: application/json
Host: localhost:5984
Content-Type: application/json

{
```

(continues on next page)

(continued from previous page)

```
    "keys": [
        "animals",
        "plants"
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 20 Dec 2017 06:57:48 GMT
Server: CouchDB (Erlang/OTP)

[
  {
    "key": "animals",
    "info": {
      "db_name": "animals",
      "update_seq": "52232",
      "sizes": {
        "file": 1178613587,
        "external": 1713103872,
        "active": 1162451555
      },
      "purge_seq": 0,
      "doc_del_count": 0,
      "doc_count": 52224,
      "disk_format_version": 6,
      "compact_running": false,
      "cluster": {
        "q": 8,
        "n": 3,
        "w": 2,
        "r": 2
      },
      "instance_start_time": "0"
    }
  },
  {
    "key": "plants",
    "info": {
      "db_name": "plants",
      "update_seq": "303",
      "sizes": {
        "file": 3872387,
        "external": 2339,
        "active": 67475
      },
      "purge_seq": 0,
      "doc_del_count": 0,
      "doc_count": 11,
      "disk_format_version": 6,
      "compact_running": false,
      "cluster": {
        "q": 8,
```

(continues on next page)

(continued from previous page)

```
        "n": 3,
        "w": 2,
        "r": 2
      },
      "instance_start_time": "0"
    }
  }
]
```

Note: The supported number of the specified databases in the list can be limited by modifying the *max_db_number_for_dbs_info_req* entry in configuration file. The default limit is 100. Increasing the limit, while possible, creates load on the server so it is advisable to have more requests with 100 dbs, rather than a few requests with 1000s of dbs at a time.

12.2.5 /_cluster_setup

New in version 2.0.

GET /_cluster_setup

Returns the status of the node or cluster, per the cluster setup wizard.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **ensure_dbs_exist** (*array*) – List of system databases to ensure exist on the node/cluster. Defaults to `["_users","_replicator"]`.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **state** (*string*) – Current `state` of the node and/or cluster (see below)

Status Codes

- 200 OK – Request completed successfully

The `state` returned indicates the current node or cluster state, and is one of the following:

- `cluster_disabled`: The current node is completely unconfigured.
- `single_node_disabled`: The current node is configured as a single (standalone) node (`[cluster] n=1`), but either does not have a server-level admin user defined, or does not have the standard system databases created. If the `ensure_dbs_exist` query parameter is specified, the list of databases provided overrides the default list of standard system databases.
- `single_node_enabled`: The current node is configured as a single (standalone) node, has a server-level admin user defined, and has the `ensure_dbs_exist` list (explicit or default) of databases created.

- `cluster_enabled`: The current node has `[cluster] n > 1`, is not bound to `127.0.0.1` and has a server-level admin user defined. However, the full set of standard system databases have not been created yet. If the `ensure_dbs_exist` query parameter is specified, the list of databases provided overrides the default list of standard system databases.
- `cluster_finished`: The current node has `[cluster] n > 1`, is not bound to `127.0.0.1`, has a server-level admin user defined *and* has the `ensure_dbs_exist` list (explicit or default) of databases created.

Request:

```
GET /_cluster_setup HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
X-CouchDB-Body-Time: 0
X-Couch-Request-ID: 5c058bdd37
Server: CouchDB/2.1.0-7f17678 (Erlang OTP/17)
Date: Sun, 30 Jul 2017 06:33:18 GMT
Content-Type: application/json
Content-Length: 29
Cache-Control: must-revalidate

{"state":"cluster_enabled"}
```

POST /_cluster_setup

Configure a node as a single (standalone) node, as part of a cluster, or finalise a cluster.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Request JSON Object

- **action** (*string*) –
  - **enable_single_node**: Configure the current node as a single, standalone CouchDB server.
  - **enable_cluster**: Configure the local or remote node as one node, preparing it to be joined to a new CouchDB cluster.
  - **add_node**: Add the specified remote node to this cluster's list of nodes, joining it to the cluster.
  - **finish_cluster**: Finalise the cluster by creating the standard system databases.
- **bind_address** (*string*) – The IP address to which to bind the current node. The special value `0.0.0.0` may be specified to bind to all interfaces on the host. (enable_cluster and enable_single_node only)
- **username** (*string*) – The username of the server-level administrator to create. (enable_cluster and enable_single_node only), or the remote server's administrator username (add_node)

- **password** (*string*) – The password for the server-level administrator to create. (enable_cluster and enable_single_node only), or the remote server's administrator username (add_node)
- **port** (*number*) – The TCP port to which to bind this node (enable_cluster and enable_single_node only) or the TCP port to which to bind a remote node (add_node only).
- **node_count** (*number*) – The total number of nodes to be joined into the cluster, including this one. Used to determine the value of the cluster's `n`, up to a maximum of 3. (enable_cluster only)
- **remote_node** (*string*) – The IP address of the remote node to setup as part of this cluster's list of nodes. (enable_cluster only)
- **remote_current_user** (*string*) – The username of the server-level administrator authorized on the remote node. (enable_cluster only)
- **remote_current_password** (*string*) – The password of the server-level administrator authorized on the remote node. (enable_cluster only)
- **host** (*string*) – The remote node IP of the node to add to the cluster. (add_node only)
- **ensure_dbs_exist** (*array*) – List of system databases to ensure exist on the node/cluster. Defaults to `["_users","_replicator"]`.

No example request/response included here. For a worked example, please see The Cluster Setup API.

12.2.6 /_db_updates

New in version 1.4.

GET /_db_updates

Returns a list of all database events in the CouchDB instance. The existence of the `_global_changes` database is required to use this endpoint.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **feed** (*string*) –
  - **normal**: Returns all historical DB changes, then closes the connection. *Default.*
  - **longpoll**: Closes the connection after the first event.
  - **continuous**: Send a line of JSON per event. Keeps the socket open until `timeout`.
  - **eventsource**: Like, `continuous`, but sends the events in EventSource format.
- **timeout** (*number*) – Number of *milliseconds* until CouchDB closes the connection. Default is `60000`.
- **heartbeat** (*number*) – Period in *milliseconds* after which an empty line is sent in the results. Only applicable for `longpoll`, `continuous`, and `eventsource` feeds. Overrides any timeout to keep the feed alive indefinitely. Default is `60000`. May be `true` to use default value.
- **since** (*string*) – Return only updates since the specified sequence ID. If the sequence ID is specified but does not exist, all changes are returned. May be the string `now` to begin showing only new updates.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- Transfer-Encoding – chunked

Response JSON Object

- **results** (*array*) – An array of database events. For `longpoll` and `continuous` modes, the entire response is the contents of the `results` array.
- **last_seq** (*string*) – The last sequence ID reported.

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

The `results` field of database updates:

JSON Parameters

- **db_name** (*string*) – Database name.
- **type** (*string*) – A database event is one of `created`, `updated`, `deleted`.
- **seq** (*json*) – Update sequence of the event.

Request:

```
GET /_db_updates HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 18 Mar 2017 19:01:35 GMT
Etag: "C1KU98Y6H0LGM7EQQYL6VSL07"
Server: CouchDB/2.0.0 (Erlang OTP/17)
Transfer-Encoding: chunked
X-Couch-Request-ID: ad87efc7ff
X-CouchDB-Body-Time: 0

{
    "results":[
        {"db_name":"mailbox","type":"created","seq":"1-
↪g1AAAAFReJzLYWBg4MhgTmHgzcvPy09JdcjLz8gvLskBCjMlMiTJ____
↪PyuDOZExFyjAnmJhkWaeaIquGIf2JAUgmWQPMiGRAZcaB5CaePxqEkBq6vGqyWMBkgwNQAqobD4h"},
        {"db_name":"mailbox","type":"deleted","seq":"2-
↪g1AAAAFReJzLYWBg4MhgTmHgzcvPy09JdcjLz8gvLskBCjMlMiTJ____
↪PyuDOZEpFyjAnmJhkWaeaIquGIf2JAUgmWQPMiGRAZcaB5CaePxqEkBq6vGqyWMBkgwNQAqobD4hdQsg6vYTUncAou4-
↪IXUPIOpA7ssCAIFHa60"}
    ],
    "last_seq": "2-g1AAAAFReJzLYWBg4MhgTmHgzcvPy09JdcjLz8gvLskBCjMlMiTJ____
↪PyuDOZEpFyjAnmJhkWaeaIquGIf2JAUgmWQPMiGRAZcaB5CaePxqEkBq6vGqyWMBkgwNQAqobD4hdQsg6vYTUncAou4-
↪IXUPIOpA7ssCAIFHa60"
}
```

12.2.7 /_membership

New in version 2.0.

GET /_membership

Displays the nodes that are part of the cluster as `cluster_nodes`. The field `all_nodes` displays all nodes this node knows about, including the ones that are part of the cluster. The endpoint is useful when setting up a cluster, see *Node Management*

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /_membership HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 11 Jul 2015 07:02:41 GMT
Server: CouchDB (Erlang/OTP)
Content-Length: 142

{
    "all_nodes": [
        "node1@127.0.0.1",
        "node2@127.0.0.1",
        "node3@127.0.0.1"
    ],
    "cluster_nodes": [
        "node1@127.0.0.1",
        "node2@127.0.0.1",
        "node3@127.0.0.1"
    ]
}
```

12.2.8 /_replicate

Changed in version 3.3: Added *"bulk_get_attempts"* and *"bulk_get_docs"* fields to the replication history response object.

POST /_replicate

Request, configure, or stop, a replication operation.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Request JSON Object

- **cancel** (*boolean*) – Cancels the replication
- **continuous** (*boolean*) – Configure the replication to be continuous
- **create_target** (*boolean*) – Creates the target database. Required administrator's privileges on target server.
- **create_target_params** (*object*) – An object that contains parameters to be used when creating the target database. Can include the standard `q` and `n` parameters.
- **winning_revs_only** (*boolean*) – Replicate winning revisions only.
- **doc_ids** (*array*) – Array of document IDs to be synchronized. `doc_ids`, `filter`, and `selector` are mutually exclusive.
- **filter** (*string*) – The name of a *filter function*. `doc_ids`, `filter`, and `selector` are mutually exclusive.
- **selector** (*json*) – A *selector* to filter documents for synchronization. Has the same behavior as the *selector objects* in replication documents. `doc_ids`, `filter`, and `selector` are mutually exclusive.
- **source_proxy** (*string*) – Address of a proxy server through which replication from the source should occur (protocol can be "http" or "socks5")
- **target_proxy** (*string*) – Address of a proxy server through which replication to the target should occur (protocol can be "http" or "socks5")
- **source** (*string/object*) – Fully qualified source database URL or an object which contains the full URL of the source database with additional parameters like headers. Eg: 'http://example.com/source_db_name' or {"url":"url in here", "headers": {"header1":"value1", ...}} . For backwards compatibility, CouchDB 3.x will auto-convert bare database names by prepending the address and port CouchDB is listening on, to form a complete URL. This behaviour is deprecated in 3.x and will be removed in CouchDB 4.0.
- **target** (*string/object*) – Fully qualified target database URL or an object which contains the full URL of the target database with additional parameters like headers. Eg: 'http://example.com/target_db_name' or {"url":"url in here", "headers": {"header1":"value1", ...}} . For backwards compatibility, CouchDB 3.x will auto-convert bare database names by prepending the address and port CouchDB is listening on, to form a complete URL. This behaviour is deprecated in 3.x and will be removed in CouchDB 4.0.

Response Headers

- Content-Type –
  - *application/json*

– *text/plain; charset=utf-8*

Response JSON Object

- **history** (*array*) – Replication history (see below)
- **ok** (*boolean*) – Replication status
- **replication_id_version** (*number*) – Replication protocol version
- **session_id** (*string*) – Unique session ID
- **source_last_seq** (*number*) – Last sequence number read from source database

Status Codes

- 200 OK – Replication request successfully completed
- 202 Accepted – Continuous replication request has been accepted
- 400 Bad Request – Invalid JSON data
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – Either the source or target DB is not found or attempt to cancel unknown replication task
- 500 Internal Server Error – JSON specification was invalid

The specification of the replication request is controlled through the JSON content of the request. The JSON should be an object with the fields defining the source, target and other options.

The *Replication history* is an array of objects with following structure:

JSON Parameters

- **doc_write_failures** (*number*) – Number of document write failures
- **docs_read** (*number*) – Number of documents read
- **docs_written** (*number*) – Number of documents written to target
- **bulk_get_attempts** (*number*) – The total count of attempted doc revisions fetched with `_bulk_get`.
- **bulk_get_docs** (*number*) – The total count of successful docs fetched with `_bulk_get`.
- **end_last_seq** (*number*) – Last sequence number in changes stream
- **end_time** (*string*) – Date/Time replication operation completed in **RFC 2822** format
- **missing_checked** (*number*) – Number of missing documents checked
- **missing_found** (*number*) – Number of missing documents found
- **recorded_seq** (*number*) – Last recorded sequence number
- **session_id** (*string*) – Session ID for this replication operation
- **start_last_seq** (*number*) – First sequence number in changes stream
- **start_time** (*string*) – Date/Time replication operation started in **RFC 2822** format

---

Note: As of CouchDB 2.0.0, fully qualified URLs are required for both the replication `source` and `target` parameters.

Request

```
POST /_replicate HTTP/1.1
Accept: application/json
Content-Length: 80
Content-Type: application/json
Host: localhost:5984

{
    "source": "http://adm:pass@127.0.0.1:5984/db_a",
    "target": "http://adm:pass@127.0.0.1:5984/db_b"
}
```

Response

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 692
Content-Type: application/json
Date: Sun, 11 Aug 2013 20:38:50 GMT
Server: CouchDB (Erlang/OTP)

{
    "history": [
        {
            "doc_write_failures": 0,
            "docs_read": 10,
            "bulk_get_attempts": 10,
            "bulk_get_docs": 10,
            "docs_written": 10,
            "end_last_seq": 28,
            "end_time": "Sun, 11 Aug 2013 20:38:50 GMT",
            "missing_checked": 10,
            "missing_found": 10,
            "recorded_seq": 28,
            "session_id": "142a35854a08e205c47174d91b1f9628",
            "start_last_seq": 1,
            "start_time": "Sun, 11 Aug 2013 20:38:50 GMT"
        },
        {
            "doc_write_failures": 0,
            "docs_read": 1,
            "bulk_get_attempts": 1,
            "bulk_get_docs": 1,
            "docs_written": 1,
            "end_last_seq": 1,
            "end_time": "Sat, 10 Aug 2013 15:41:54 GMT",
            "missing_checked": 1,
            "missing_found": 1,
            "recorded_seq": 1,
            "session_id": "6314f35c51de3ac408af79d6ee0c1a09",
            "start_last_seq": 0,
            "start_time": "Sat, 10 Aug 2013 15:41:54 GMT"
        }
    ],
    "ok": true,
    "replication_id_version": 3,
    "session_id": "142a35854a08e205c47174d91b1f9628",
    "source_last_seq": 28
```

(continues on next page)

(continued from previous page)

```
}
```

Replication Operation

The aim of the replication is that at the end of the process, all active documents on the source database are also in the destination database and all documents that were deleted in the source databases are also deleted (if they exist) on the destination database.

Replication can be described as either push or pull replication:

- *Pull replication* is where the `source` is the remote CouchDB instance, and the `target` is the local database.

  Pull replication is the most useful solution to use if your source database has a permanent IP address, and your destination (local) database may have a dynamically assigned IP address (for example, through DHCP). This is particularly important if you are replicating to a mobile or other device from a central server.

- *Push replication* is where the `source` is a local database, and `target` is a remote database.

Specifying the Source and Target Database

You must use the URL specification of the CouchDB database if you want to perform replication in either of the following two situations:

- Replication with a remote database (i.e. another instance of CouchDB on the same host, or a different host)
- Replication with a database that requires authentication

For example, to request replication between a database local to the CouchDB instance to which you send the request, and a remote database you might use the following request:

```
POST http://couchdb:5984/_replicate HTTP/1.1
Content-Type: application/json
Accept: application/json

{
    "source" : "recipes",
    "target" : "http://coucdb-remote:5984/recipes",
}
```

In all cases, the requested databases in the `source` and `target` specification must exist. If they do not, an error will be returned within the JSON object:

```
{
    "error" : "db_not_found"
    "reason" : "could not open http://couchdb-remote:5984/ol1ka/",
}
```

You can create the target database (providing your user credentials allow it) by adding the `create_target` field to the request object:

```
POST http://couchdb:5984/_replicate HTTP/1.1
Content-Type: application/json
Accept: application/json

{
    "create_target" : true
    "source" : "recipes",
```

(continues on next page)

(continued from previous page)

```
    "target" : "http://couchdb-remote:5984/recipes",
}
```

The `create_target` field is not destructive. If the database already exists, the replication proceeds as normal.

Single Replication

You can request replication of a database so that the two databases can be synchronized. By default, the replication process occurs one time and synchronizes the two databases together. For example, you can request a single synchronization between two databases by supplying the `source` and `target` fields within the request JSON content.

```
POST http://couchdb:5984/_replicate HTTP/1.1
Accept: application/json
Content-Type: application/json

{
    "source" : "recipes",
    "target" : "recipes-snapshot",
}
```

In the above example, the databases `recipes` and `recipes-snapshot` will be synchronized. These databases are local to the CouchDB instance where the request was made. The response will be a JSON structure containing the success (or failure) of the synchronization process, and statistics about the process:

```
{
    "ok" : true,
    "history" : [
        {
            "docs_read" : 1000,
            "bulk_get_attempts": 1000,
            "bulk_get_docs": 1000,
            "session_id" : "52c2370f5027043d286daca4de247db0",
            "recorded_seq" : 1000,
            "end_last_seq" : 1000,
            "doc_write_failures" : 0,
            "start_time" : "Thu, 28 Oct 2010 10:24:13 GMT",
            "start_last_seq" : 0,
            "end_time" : "Thu, 28 Oct 2010 10:24:14 GMT",
            "missing_checked" : 0,
            "docs_written" : 1000,
            "missing_found" : 1000
        }
    ],
    "session_id" : "52c2370f5027043d286daca4de247db0",
    "source_last_seq" : 1000
}
```

Continuous Replication

Synchronization of a database with the previously noted methods happens only once, at the time the replicate request is made. To have the target database permanently replicated from the source, you must set the `continuous` field of the JSON object within the request to true.

With continuous replication changes in the source database are replicated to the target database in perpetuity until you specifically request that replication ceases.

```
POST http://couchdb:5984/_replicate HTTP/1.1
Accept: application/json
Content-Type: application/json

{
    "continuous" : true
    "source" : "recipes",
    "target" : "http://couchdb-remote:5984/recipes",
}
```

Changes will be replicated between the two databases as long as a network connection is available between the two instances.

Note: Two keep two databases synchronized with each other, you need to set replication in both directions; that is, you must replicate from `source` to `target`, and separately from `target` to `source`.

Canceling Continuous Replication

You can cancel continuous replication by adding the `cancel` field to the JSON request object and setting the value to true. Note that the structure of the request must be identical to the original for the cancellation request to be honoured. For example, if you requested continuous replication, the cancellation request must also contain the `continuous` field.

For example, the replication request:

```
POST http://couchdb:5984/_replicate HTTP/1.1
Content-Type: application/json
Accept: application/json

{
    "source" : "recipes",
    "target" : "http://couchdb-remote:5984/recipes",
    "create_target" : true,
    "continuous" : true
}
```

Must be canceled using the request:

```
POST http://couchdb:5984/_replicate HTTP/1.1
Accept: application/json
Content-Type: application/json

{
    "cancel" : true,
    "continuous" : true
    "create_target" : true,
    "source" : "recipes",
```

(continues on next page)

(continued from previous page)

```
    "target" : "http://couchdb-remote:5984/recipes",
}
```

Requesting cancellation of a replication that does not exist results in a 404 error.

12.2.9 /_scheduler/jobs

GET /_scheduler/jobs

List of replication jobs. Includes replications created via */_replicate* endpoint as well as those created from replication documents. Does not include replications which have completed or have failed to start because replication documents were malformed. Each job description will include source and target information, replication id, a history of recent event, and a few other things.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Query Parameters

- **limit** (*number*) – How many results to return
- **skip** (*number*) – How many result to skip starting at the beginning, ordered by replication ID

Response JSON Object

- **offset** (*number*) – How many results were skipped
- **total_rows** (*number*) – Total number of replication jobs
- **id** (*string*) – Replication ID.
- **database** (*string*) – Replication document database
- **doc_id** (*string*) – Replication document ID
- **history** (*list*) – Timestamped history of events as a list of objects
- **pid** (*string*) – Replication process ID
- **node** (*string*) – Cluster node where the job is running
- **source** (*string*) – Replication source
- **target** (*string*) – Replication target
- **start_time** (*string*) – Timestamp of when the replication was started

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_scheduler/jobs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 1690
Content-Type: application/json
Date: Sat, 29 Apr 2017 05:05:16 GMT
Server: CouchDB (Erlang/OTP)

{
    "jobs": [
        {
            "database": "_replicator",
            "doc_id": "cdyno-0000001-0000003",
            "history": [
                {
                    "timestamp": "2017-04-29T05:01:37Z",
                    "type": "started"
                },
                {
                    "timestamp": "2017-04-29T05:01:37Z",
                    "type": "added"
                }
            ],
            "id": "8f5b1bd0be6f9166ccfd36fc8be8fc22+continuous",
            "info": {
                "changes_pending": 0,
                "checkpointed_source_seq": "113-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ",
                "doc_write_failures": 0,
                "docs_read": 113,
                "docs_written": 113,
                "bulk_get_attempts": 113,
                "bulk_get_docs": 113,
                "missing_revisions_found": 113,
                "revisions_checked": 113,
                "source_seq": "113-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ",
                "through_seq": "113-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE01ygQLsZsYGqcamiZjKcRqRxwIkGRqA1H-oSbZgk1KMLCzTDE0wdWUBAF6HJIQ"
            },
            "node": "node1@127.0.0.1",
            "pid": "<0.1850.0>",
            "source": "http://myserver.com/foo",
            "start_time": "2017-04-29T05:01:37Z",
            "target": "http://adm:*****@localhost:15984/cdyno-0000003/",
            "user": null
        },
        {
            "database": "_replicator",
            "doc_id": "cdyno-0000001-0000002",
            "history": [
                {
                    "timestamp": "2017-04-29T05:01:37Z",
                    "type": "started"
```

(continues on next page)

(continued from previous page)

```
                },
                {
                    "timestamp": "2017-04-29T05:01:37Z",
                    "type": "added"
                }
            ],
            "id": "e327d79214831ca4c11550b4a453c9ba+continuous",
            "info": {
                "changes_pending": null,
                "checkpointed_source_seq": 0,
                "doc_write_failures": 0,
                "docs_read": 12,
                "docs_written": 12,
                "bulk_get_attempts": 12,
                "bulk_get_docs": 12,
                "missing_revisions_found": 12,
                "revisions_checked": 12,
                "source_seq": "12-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE1lzgQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSexgk4yMkhITjS0wdWUBADfEJBg",
                "through_seq": "12-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE1lzgQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSexgk4yMkhITjS0wdWUBADfEJBg"
            },
            "node": "node2@127.0.0.1",
            "pid": "<0.1757.0>",
            "source": "http://myserver.com/foo",
            "start_time": "2017-04-29T05:01:37Z",
            "target": "http://adm:*****@localhost:15984/cdyno-0000002/",
            "user": null
        }
    ],
    "offset": 0,
    "total_rows": 2
}
```

12.2.10 /_scheduler/docs

Changed in version 2.1.0: Use this endpoint to monitor the state of document-based replications. Previously needed to poll both documents and `_active_tasks` to get a complete state summary

Changed in version 3.0.0: In error states the *"info"* field switched from being a string to being an object

Changed in version 3.3: Added *"bulk_get_attempts"* and *"bulk_get_docs"* the *"info"* object.

GET /_scheduler/docs

List of replication document states. Includes information about all the documents, even in `completed` and `failed` states. For each document it returns the document ID, the database, the replication ID, source and target, and other information.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –

– *application/json*

Query Parameters

- **limit** (*number*) – How many results to return
- **skip** (*number*) – How many result to skip starting at the beginning, if ordered by document ID

Response JSON Object

- **offset** (*number*) – How many results were skipped
- **total_rows** (*number*) – Total number of replication documents.
- **id** (*string*) – Replication ID, or `null` if state is `completed` or `failed`
- **state** (*string*) – One of following states (see *Replication states* for descriptions): `initializing`, `running`, `completed`, `pending`, `crashing`, `error`, `failed`
- **database** (*string*) – Database where replication document came from
- **doc_id** (*string*) – Replication document ID
- **node** (*string*) – Cluster node where the job is running
- **source** (*string*) – Replication source
- **target** (*string*) – Replication target
- **start_time** (*string*) – Timestamp of when the replication was started
- **last_updated** (*string*) – Timestamp of last state update
- **info** (*object*) – Will contain additional information about the state. For errors, this will be an object with an `"error"` field and string value. For success states, see below.
- **error_count** (*number*) – Consecutive errors count. Indicates how many times in a row this replication has crashed. Replication will be retried with an exponential backoff based on this number. As soon as the replication succeeds this count is reset to 0. To can be used to get an idea why a particular replication is not making progress.

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

The `info` field of a scheduler doc:

JSON Parameters

- **revisions_checked** (*number*) – The count of revisions which have been checked since this replication began.
- **missing_revisions_found** (*number*) – The count of revisions which were found on the source, but missing from the target.
- **docs_read** (*number*) – The count of docs which have been read from the source.
- **docs_written** (*number*) – The count of docs which have been written to the target.
- **bulk_get_attempts** (*number*) – The total count of attempted doc revisions fetched with `_bulk_get`.
- **bulk_get_docs** (*number*) – The total count of successful docs fetched with `_bulk_get`.
- **changes_pending** (*number*) – The count of changes not yet replicated.
- **doc_write_failures** (*number*) – The count of docs which failed to be written to the target.

- **checkpointed_source_seq** (*object*) – The source sequence id which was last successfully replicated.

Request:

```
GET /_scheduler/docs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sat, 29 Apr 2017 05:10:08 GMT
Server: Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "docs": [
        {
            "database": "_replicator",
            "doc_id": "cdyno-0000001-0000002",
            "error_count": 0,
            "id": "e327d79214831ca4c11550b4a453c9ba+continuous",
            "info": {
                "changes_pending": 15,
                "checkpointed_source_seq": "60-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYEyVygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSSpgk4yMkhITjS0wdWUBAENCJEg",
                "doc_write_failures": 0,
                "docs_read": 67,
                "bulk_get_attempts": 67,
                "bulk_get_docs": 67,
                "docs_written": 67,
                "missing_revisions_found": 67,
                "revisions_checked": 67,
                "source_seq": "67-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE2VygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSepgk4yMkhITjS0wdWUBAEVKJE8",
                "through_seq": "67-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE2VygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSepgk4yMkhITjS0wdWUBAEVKJE8"
            },
            "last_updated": "2017-04-29T05:01:37Z",
            "node": "node2@127.0.0.1",
            "source_proxy": null,
            "target_proxy": null,
            "source": "http://myserver.com/foo",
            "start_time": "2017-04-29T05:01:37Z",
            "state": "running",
            "target": "http://adm:*****@localhost:15984/cdyno-0000002/"
        },
        {
            "database": "_replicator",
            "doc_id": "cdyno-0000001-0000003",
            "error_count": 0,
            "id": "8f5b1bd0be6f9166ccfd36fc8be8fc22+continuous",
            "info": {
```

(continues on next page)

(continued from previous page)

```
                "changes_pending": null,
                "checkpointed_source_seq": 0,
                "doc_write_failures": 0,
                "bulk_get_attempts": 12,
                "bulk_get_docs": 12,
                "docs_read": 12,
                "docs_written": 12,
                "missing_revisions_found": 12,
                "revisions_checked": 12,
                "source_seq": "12-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE1lzgQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSexgk4yMkhITjS0wdWUBADfEJBg",
                "through_seq": "12-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE1lzgQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSexgk4yMkhITjS0wdWUBADfEJBg"
            },
            "last_updated": "2017-04-29T05:01:37Z",
            "node": "node1@127.0.0.1",
            "source_proxy": null,
            "target_proxy": null,
            "source": "http://myserver.com/foo",
            "start_time": "2017-04-29T05:01:37Z",
            "state": "running",
            "target": "http://adm:*****@localhost:15984/cdyno-0000003/"
        }
    ],
    "offset": 0,
    "total_rows": 2
}
```

GET /_scheduler/docs/{replicator_db}

Get information about replication documents from a replicator database. The default replicator database is `_replicator` but other replicator databases can exist if their name ends with the suffix `/_replicator`.

Note: As a convenience slashes (/) in replicator db names do not have to be escaped. So `/_scheduler/docs/other/_replicator` is valid and equivalent to `/_scheduler/docs/other%2f_replicator`

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Query Parameters

- **limit** (*number*) – How many results to return
- **skip** (*number*) – How many result to skip starting at the beginning, if ordered by document ID

Response JSON Object

- **offset** (*number*) – How many results were skipped
- **total_rows** (*number*) – Total number of replication documents.

- **id** (*string*) – Replication ID, or `null` if state is `completed` or `failed`
- **state** (*string*) – One of following states (see *Replication states* for descriptions): `initializing`, `running`, `completed`, `pending`, `crashing`, `error`, `failed`
- **database** (*string*) – Database where replication document came from
- **doc_id** (*string*) – Replication document ID
- **node** (*string*) – Cluster node where the job is running
- **source** (*string*) – Replication source
- **target** (*string*) – Replication target
- **start_time** (*string*) – Timestamp of when the replication was started
- **last_update** (*string*) – Timestamp of last state update
- **info** (*object*) – Will contain additional information about the state. For errors, this will be an object with an `"error"` field and string value. For success states, see below.
- **error_count** (*number*) – Consecutive errors count. Indicates how many times in a row this replication has crashed. Replication will be retried with an exponential backoff based on this number. As soon as the replication succeeds this count is reset to 0. To can be used to get an idea why a particular replication is not making progress.

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

The `info` field of a scheduler doc:

JSON Parameters

- **revisions_checked** (*number*) – The count of revisions which have been checked since this replication began.
- **missing_revisions_found** (*number*) – The count of revisions which were found on the source, but missing from the target.
- **docs_read** (*number*) – The count of docs which have been read from the source.
- **docs_written** (*number*) – The count of docs which have been written to the target.
- **bulk_get_attempts** (*number*) – The total count of attempted doc revisions fetched with `_bulk_get`.
- **bulk_get_docs** (*number*) – The total count of successful docs fetched with `_bulk_get`.
- **changes_pending** (*number*) – The count of changes not yet replicated.
- **doc_write_failures** (*number*) – The count of docs which failed to be written to the target.
- **checkpointed_source_seq** (*object*) – The source sequence id which was last successfully replicated.

Request:

```
GET /_scheduler/docs/other/_replicator HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sat, 29 Apr 2017 05:10:08 GMT
Server: Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "docs": [
        {
            "database": "other/_replicator",
            "doc_id": "cdyno-0000001-0000002",
            "error_count": 0,
            "id": "e327d79214831ca4c11550b4a453c9ba+continuous",
            "info": {
                "changes_pending": 0,
                "checkpointed_source_seq": "60-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYEyVygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSSpgk4yMkhITjS0wdWUBAENCJEg",
                "doc_write_failures": 0,
                "docs_read": 67,
                "bulk_get_attempts": 67,
                "bulk_get_docs": 67,
                "docs_written": 67,
                "missing_revisions_found": 67,
                "revisions_checked": 67,
                "source_seq": "67-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE2VygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSepgk4yMkhITjS0wdWUBAEVKJE8",
                "through_seq": "67-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE2VygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSepgk4yMkhITjS0wdWUBAEVKJE8"
            },
            "last_updated": "2017-04-29T05:01:37Z",
            "node": "node2@127.0.0.1",
            "source_proxy": null,
            "target_proxy": null,
            "source": "http://myserver.com/foo",
            "start_time": "2017-04-29T05:01:37Z",
            "state": "running",
            "target": "http://adm:*****@localhost:15984/cdyno-0000002/"
        }
    ],
    "offset": 0,
    "total_rows": 1
}
```

GET /_scheduler/docs/{replicator_db}/{docid}

> **Note:** As a convenience slashes (/) in replicator db names do not have to be escaped. So `/_scheduler/docs/other/_replicator` is valid and equivalent to `/_scheduler/docs/other%2f_replicator`

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Response JSON Object

- **id** (*string*) – Replication ID, or `null` if state is `completed` or `failed`
- **state** (*string*) – One of following states (see *Replication states* for descriptions): `initializing`, `running`, `completed`, `pending`, `crashing`, `error`, `failed`
- **database** (*string*) – Database where replication document came from
- **doc_id** (*string*) – Replication document ID
- **node** (*string*) – Cluster node where the job is running
- **source** (*string*) – Replication source
- **target** (*string*) – Replication target
- **start_time** (*string*) – Timestamp of when the replication was started
- **last_update** (*string*) – Timestamp of last state update
- **info** (*object*) – Will contain additional information about the state. For errors, this will be an object with an `"error"` field and string value. For success states, see below.
- **error_count** (*number*) – Consecutive errors count. Indicates how many times in a row this replication has crashed. Replication will be retried with an exponential backoff based on this number. As soon as the replication succeeds this count is reset to 0. To can be used to get an idea why a particular replication is not making progress.

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

The `info` field of a scheduler doc:

JSON Parameters

- **revisions_checked** (*number*) – The count of revisions which have been checked since this replication began.
- **missing_revisions_found** (*number*) – The count of revisions which were found on the source, but missing from the target.
- **docs_read** (*number*) – The count of docs which have been read from the source.
- **docs_written** (*number*) – The count of docs which have been written to the target.
- **bulk_get_attempts** (*number*) – The total count of attempted doc revisions fetched with `_bulk_get`.
- **bulk_get_docs** (*number*) – The total count of successful docs fetched with `_bulk_get`.
- **changes_pending** (*number*) – The count of changes not yet replicated.
- **doc_write_failures** (*number*) – The count of docs which failed to be written to the target.
- **checkpointed_source_seq** (*object*) –

  The source sequence id which was last
  successfully replicated.

  Request:

```
GET /_scheduler/docs/other/_replicator/cdyno-0000001-0000002 HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sat, 29 Apr 2017 05:10:08 GMT
Server: Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "database": "other/_replicator",
    "doc_id": "cdyno-0000001-0000002",
    "error_count": 0,
    "id": "e327d79214831ca4c11550b4a453c9ba+continuous",
    "info": {
        "changes_pending": 0,
        "checkpointed_source_seq": "60-
↪g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYEyVygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSSpgk4yMkhITjS0wdWUBAENCJEg",
        "doc_write_failures": 0,
        "docs_read": 67,
        "bulk_get_attempts": 67,
        "bulk_get_docs": 67,
        "docs_written": 67,
        "missing_revisions_found": 67,
        "revisions_checked": 67,
        "source_seq": "67-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9___
↪_szKYE2VygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSepgk4yMkhITjS0wdWUBAEVKJE8",
        "through_seq": "67-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9__
↪__szKYE2VygQLsBsZm5pZJJpjKcRqRxwIkGRqA1H-oSepgk4yMkhITjS0wdWUBAEVKJE8"
    },
    "last_updated": "2017-04-29T05:01:37Z",
    "node": "node2@127.0.0.1",
    "source_proxy": null,
    "target_proxy": null,
    "source": "http://myserver.com/foo",
    "start_time": "2017-04-29T05:01:37Z",
    "state": "running",
    "target": "http://adm:*****@localhost:15984/cdyno-0000002/"
}
```

12.2.11 /_node/{node-name}

GET /_node/{node-name}

The /_node/{node-name} endpoint can be used to confirm the Erlang node name of the server that processes the request. This is most useful when accessing /_node/_local to retrieve this information. Repeatedly retrieving this information for a CouchDB endpoint can be useful to determine if a CouchDB cluster is correctly proxied through a reverse load balancer.

Request Headers

- Accept –
  - *application/json*

- *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /_node/_local HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 27
Content-Type: application/json
Date: Tue, 28 Jan 2020 19:25:51 GMT
Server: CouchDB (Erlang OTP)
X-Couch-Request-ID: 5b8db6c677
X-CouchDB-Body-Time: 0

{"name":"node1@127.0.0.1"}
```

12.2.12 /_node/{node-name}/_stats

GET /_node/{node-name}/_stats

The `_stats` resource returns a JSON object containing the statistics for the running server. The object is structured with top-level sections collating the statistics for a range of entries, with each individual statistic being easily identified, and the content of each statistic is self-describing.

Statistics are sampled internally on a *configurable interval*. When monitoring the `_stats` endpoint, you need to use a polling frequency of at least twice this to observe accurate results. For example, if the *interval* is 10 seconds, poll `_stats` at least every 5 seconds.

The literal string `_local` serves as an alias for the local node name, so for all stats URLs, `{node-name}` may be replaced with `_local`, to interact with the local node's statistics.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /_node/_local/_stats/couchdb/request_time HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 187
Content-Type: application/json
Date: Sat, 10 Aug 2013 11:41:11 GMT
Server: CouchDB (Erlang/OTP)

{
  "value": {
    "min": 0,
    "max": 0,
    "arithmetic_mean": 0,
    "geometric_mean": 0,
    "harmonic_mean": 0,
    "median": 0,
    "variance": 0,
    "standard_deviation": 0,
    "skewness": 0,
    "kurtosis": 0,
    "percentile": [
      [
        50,
        0
      ],
      [
        75,
        0
      ],
      [
        90,
        0
      ],
      [
        95,
        0
      ],
      [
        99,
        0
      ],
      [
        999,
        0
      ]
    ],
    "histogram": [
      [
        0,
        0
      ]
```

(continues on next page)

(continued from previous page)

```
    ],
    "n": 0
  },
  "type": "histogram",
  "desc": "length of a request inside CouchDB without MochiWeb"
}
```

The fields provide the current, minimum and maximum, and a collection of statistical means and quantities. The quantity in each case is not defined, but the descriptions below provide sufficient detail to determine units.

Statistics are reported by 'group'. The statistics are divided into the following top-level sections:

- `couch_log`: Logging subsystem
- `couch_replicator`: Replication scheduler and subsystem
- `couchdb`: Primary CouchDB database operations
- `fabric`: Cluster-related operations
- `global_changes`: Global changes feed
- `mem3`: Node membership-related statistics
- `pread`: CouchDB file-related exceptions
- `rexi`: Cluster internal RPC-related statistics

The type of the statistic is included in the `type` field, and is one of the following:

- `counter`: Monotonically increasing counter, resets on restart
- `histogram`: Binned set of values with meaningful subdivisions. Scoped to the current *collection interval*.
- `gauge`: Single numerical value that can go up and down

You can also access individual statistics by quoting the statistics sections and statistic ID as part of the URL path. For example, to get the `request_time` statistics within the `couchdb` section for the target node, you can use:

```
GET /_node/_local/_stats/couchdb/request_time HTTP/1.1
```

This returns an entire statistics object, as with the full request, but containing only the requested individual statistic.

12.2.13 /_node/{node-name}/_prometheus

GET /_node/{node-name}/_prometheus

The `_prometheus` resource returns a text/plain response that consolidates our */_node/{node-name}/_stats*, and */_node/{node-name}/_system* endpoints. The format is determined by Prometheus. The format version is 2.0.

Request:

```
GET /_node/_local/_prometheus HTTP/1.1
Accept: text/plain
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 187
Content-Type: text/plain; version=2.0
Date: Sat, 10 May 2020 11:41:11 GMT
```

(continues on next page)

(continued from previous page)

```
Server: CouchDB (Erlang/OTP)

# TYPE couchdb_couch_log_requests_total counter
couchdb_couch_log_requests_total{level="alert"} 0
couchdb_couch_log_requests_total{level="critical"} 0
couchdb_couch_log_requests_total{level="debug"} 0
couchdb_couch_log_requests_total{level="emergency"} 0
couchdb_couch_log_requests_total{level="error"} 0
couchdb_couch_log_requests_total{level="info"} 8
couchdb_couch_log_requests_total{level="notice"} 51
couchdb_couch_log_requests_total{level="warning"} 0
# TYPE couchdb_couch_replicator_changes_manager_deaths_total counter
couchdb_couch_replicator_changes_manager_deaths_total 0
# TYPE couchdb_couch_replicator_changes_queue_deaths_total counter
couchdb_couch_replicator_changes_queue_deaths_total 0
# TYPE couchdb_couch_replicator_changes_read_failures_total counter
couchdb_couch_replicator_changes_read_failures_total 0
# TYPE couchdb_couch_replicator_changes_reader_deaths_total counter
couchdb_couch_replicator_changes_reader_deaths_total 0
# TYPE couchdb_couch_replicator_checkpoints_failure_total counter
couchdb_couch_replicator_checkpoints_failure_total 0
# TYPE couchdb_couch_replicator_checkpoints_total counter
couchdb_couch_replicator_checkpoints_total 0
# TYPE couchdb_couch_replicator_cluster_is_stable gauge
couchdb_couch_replicator_cluster_is_stable 1
# TYPE couchdb_couch_replicator_connection_acquires_total counter
couchdb_couch_replicator_connection_acquires_total 0
# TYPE couchdb_couch_replicator_connection_closes_total counter
couchdb_couch_replicator_connection_closes_total 0
# TYPE couchdb_couch_replicator_connection_creates_total counter
couchdb_couch_replicator_connection_creates_total 0
# TYPE couchdb_couch_replicator_connection_owner_crashes_total counter
couchdb_couch_replicator_connection_owner_crashes_total 0
# TYPE couchdb_couch_replicator_connection_releases_total counter
couchdb_couch_replicator_connection_releases_total 0
# TYPE couchdb_couch_replicator_connection_worker_crashes_total counter
couchdb_couch_replicator_connection_worker_crashes_total 0
# TYPE couchdb_couch_replicator_db_scans_total counter
couchdb_couch_replicator_db_scans_total 1
# TYPE couchdb_couch_replicator_docs_completed_state_updates_total counter
couchdb_couch_replicator_docs_completed_state_updates_total 0
# TYPE couchdb_couch_replicator_docs_db_changes_total counter
couchdb_couch_replicator_docs_db_changes_total 0
# TYPE couchdb_couch_replicator_docs_dbs_deleted_total counter
couchdb_couch_replicator_docs_dbs_deleted_total 0
# TYPE couchdb_couch_replicator_docs_dbs_found_total counter
couchdb_couch_replicator_docs_dbs_found_total 2
# TYPE couchdb_couch_replicator_docs_failed_state_updates_total counter
couchdb_couch_replicator_docs_failed_state_updates_total 0
# TYPE couchdb_couch_replicator_failed_starts_total counter
couchdb_couch_replicator_failed_starts_total 0
# TYPE couchdb_couch_replicator_jobs_adds_total counter
couchdb_couch_replicator_jobs_adds_total 0
# TYPE couchdb_couch_replicator_jobs_crashed gauge
couchdb_couch_replicator_jobs_crashed 0
# TYPE couchdb_couch_replicator_jobs_crashes_total counter
```

(continues on next page)

(continued from previous page)

```
couchdb_couch_replicator_jobs_crashes_total 0
# TYPE couchdb_couch_replicator_jobs_duplicate_adds_total counter
couchdb_couch_replicator_jobs_duplicate_adds_total 0
# TYPE couchdb_couch_replicator_jobs_pending gauge
couchdb_couch_replicator_jobs_pending 0
# TYPE couchdb_couch_replicator_jobs_removes_total counter
couchdb_couch_replicator_jobs_removes_total 0
# TYPE couchdb_couch_replicator_jobs_running gauge
couchdb_couch_replicator_jobs_running 0
# TYPE couchdb_couch_replicator_jobs_starts_total counter
couchdb_couch_replicator_jobs_starts_total 0
# TYPE couchdb_couch_replicator_jobs_stops_total counter
couchdb_couch_replicator_jobs_stops_total 0
# TYPE couchdb_couch_replicator_jobs_total gauge
couchdb_couch_replicator_jobs_total 0
# TYPE couchdb_couch_replicator_requests_total counter
couchdb_couch_replicator_requests_total 0
# TYPE couchdb_couch_replicator_responses_failure_total counter
couchdb_couch_replicator_responses_failure_total 0
# TYPE couchdb_couch_replicator_responses_total counter
couchdb_couch_replicator_responses_total 0
# TYPE couchdb_couch_replicator_stream_responses_failure_total counter
couchdb_couch_replicator_stream_responses_failure_total 0
# TYPE couchdb_couch_replicator_stream_responses_total counter
couchdb_couch_replicator_stream_responses_total 0
# TYPE couchdb_couch_replicator_worker_deaths_total counter
couchdb_couch_replicator_worker_deaths_total 0
# TYPE couchdb_couch_replicator_workers_started_total counter
couchdb_couch_replicator_workers_started_total 0
# TYPE couchdb_auth_cache_requests_total counter
couchdb_auth_cache_requests_total 0
# TYPE couchdb_auth_cache_misses_total counter
couchdb_auth_cache_misses_total 0
# TYPE couchdb_collect_results_time_seconds summary
couchdb_collect_results_time_seconds{quantile="0.5"} 0.0
couchdb_collect_results_time_seconds{quantile="0.75"} 0.0
couchdb_collect_results_time_seconds{quantile="0.9"} 0.0
couchdb_collect_results_time_seconds{quantile="0.95"} 0.0
couchdb_collect_results_time_seconds{quantile="0.99"} 0.0
couchdb_collect_results_time_seconds{quantile="0.999"} 0.0
couchdb_collect_results_time_seconds_sum 0.0
couchdb_collect_results_time_seconds_count 0
# TYPE couchdb_couch_server_lru_skip_total counter
couchdb_couch_server_lru_skip_total 0
# TYPE couchdb_database_purges_total counter
couchdb_database_purges_total 0
# TYPE couchdb_database_reads_total counter
couchdb_database_reads_total 0
# TYPE couchdb_database_writes_total counter
couchdb_database_writes_total 0
# TYPE couchdb_db_open_time_seconds summary
couchdb_db_open_time_seconds{quantile="0.5"} 0.0
couchdb_db_open_time_seconds{quantile="0.75"} 0.0
couchdb_db_open_time_seconds{quantile="0.9"} 0.0
couchdb_db_open_time_seconds{quantile="0.95"} 0.0
couchdb_db_open_time_seconds{quantile="0.99"} 0.0
```

(continues on next page)

(continued from previous page)

```
couchdb_db_open_time_seconds{quantile="0.999"} 0.0
couchdb_db_open_time_seconds_sum 0.0
couchdb_db_open_time_seconds_count 0
# TYPE couchdb_dbinfo_seconds summary
couchdb_dbinfo_seconds{quantile="0.5"} 0.0
couchdb_dbinfo_seconds{quantile="0.75"} 0.0
couchdb_dbinfo_seconds{quantile="0.9"} 0.0
couchdb_dbinfo_seconds{quantile="0.95"} 0.0
couchdb_dbinfo_seconds{quantile="0.99"} 0.0
couchdb_dbinfo_seconds{quantile="0.999"} 0.0
couchdb_dbinfo_seconds_sum 0.0
couchdb_dbinfo_seconds_count 0
# TYPE couchdb_document_inserts_total counter
couchdb_document_inserts_total 0
# TYPE couchdb_document_purges_failure_total counter
couchdb_document_purges_failure_total 0
# TYPE couchdb_document_purges_success_total counter
couchdb_document_purges_success_total 0
# TYPE couchdb_document_purges_total_total counter
couchdb_document_purges_total_total 0
# TYPE couchdb_document_writes_total counter
couchdb_document_writes_total 0
# TYPE couchdb_httpd_aborted_requests_total counter
couchdb_httpd_aborted_requests_total 0
# TYPE couchdb_httpd_all_docs_timeouts_total counter
couchdb_httpd_all_docs_timeouts_total 0
# TYPE couchdb_httpd_bulk_docs_seconds summary
couchdb_httpd_bulk_docs_seconds{quantile="0.5"} 0.0
couchdb_httpd_bulk_docs_seconds{quantile="0.75"} 0.0
couchdb_httpd_bulk_docs_seconds{quantile="0.9"} 0.0
couchdb_httpd_bulk_docs_seconds{quantile="0.95"} 0.0
couchdb_httpd_bulk_docs_seconds{quantile="0.99"} 0.0
couchdb_httpd_bulk_docs_seconds{quantile="0.999"} 0.0
couchdb_httpd_bulk_docs_seconds_sum 0.0
couchdb_httpd_bulk_docs_seconds_count 0
...remaining couchdb metrics from _stats and _system
```

If an additional port config option is specified, then a client can call this API using that port which does not require authentication. This option is `false` (OFF) by default. When the option is `true` (ON), the default ports for a 3 node cluster are `17986`, `27986`, `37986`. See *Configuration of Prometheus Endpoint* for details.

```
GET /_node/_local/_prometheus HTTP/1.1
Accept: text/plain
Host: localhost:17986
```

12.2.14 /_node/{node-name}/_smoosh/status

New in version 3.4.

GET /_node/{node-name}/_smoosh/status

This prints the state of each channel, how many jobs they are currently running and how many jobs are enqueued (as well as the lowest and highest priority of those enqueued items). The idea is to provide, at a glance, sufficient insight into `smoosh` that an operator can assess whether `smoosh` is adequately targeting the reclaimable space in the cluster.

In general, a healthy status output will have items in the `ratio_dbs` and `ratio_views` channels. Owing to the default settings, the `slack_dbs` and `slack_views` will almost certainly have items in them. Historically,

we've not found that the slack channels, on their own, are particularly adept at keeping things well compacted.

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_node/_local/_smoosh/status HTTP/1.1
Host: 127.0.0.1:5984
Accept: */*
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "channels": {
        "slack_dbs": {
            "starting": 0,
            "waiting": {
                "size": 0,
                "min": 0,
                "max": 0
            },
            "active": 0
        },
        "ratio_dbs": {
            "starting": 0,
            "waiting": {
                "size": 56,
                "min": 1.125,
                "max": 11.0625
            },
            "active": 0
        },
        "ratio_views": {
            "starting": 0,
            "waiting": {
                "size": 0,
                "min": 0,
                "max": 0
            },
            "active": 0
        },
        "upgrade_dbs": {
            "starting": 0,
            "waiting": {
                "size": 0,
                "min": 0,
                "max": 0
            },
            "active": 0
        },
        "slack_views": {
            "starting": 0,
            "waiting": {
```

(continues on next page)

(continued from previous page)

```
                "size": 0,
                "min": 0,
                "max": 0
            },
            "active": 0
        },
        "upgrade_views": {
            "starting": 0,
            "waiting": {
                "size": 0,
                "min": 0,
                "max": 0
            },
            "active": 0
        },
        "index_cleanup": {
            "starting": 0,
            "waiting": {
                "size": 0,
                "min": 0,
                "max": 0
            },
            "active": 0
        }
    }
}
```

12.2.15 /_node/{node-name}/_system

GET /_node/{node-name}/_system

The `_system` resource returns a JSON object containing various system-level statistics for the running server. The object is structured with top-level sections collating the statistics for a range of entries, with each individual statistic being easily identified, and the content of each statistic is self-describing.

The literal string `_local` serves as an alias for the local node name, so for all stats URLs, `{node-name}` may be replaced with `_local`, to interact with the local node's statistics.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /_node/_local/_system HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 187
Content-Type: application/json
Date: Sat, 10 Aug 2013 11:41:11 GMT
Server: CouchDB (Erlang/OTP)

{
  "uptime": 259,
  "memory": {}
}
```

These statistics are generally intended for CouchDB developers only.

12.2.16 /_node/{node-name}/_restart

POST /_node/{node-name}/_restart

This API is to facilitate integration testing only it is not meant to be used in production

Status Codes

- 200 OK – Request completed successfully

12.2.17 /_node/{node-name}/_versions

GET /_node/{node-name}/_versions

The `_versions` resource returns a JSON object containing various system-level informations for the running server.

The literal string `_local` serves as an alias for the local node name, so for all stats URLs, `{node-name}` may be replaced with `_local`, to interact with the local node's informations.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /_node/_local/_versions HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 368
Content-Type: application/json
Date: Sat, 03 Sep 2022 08:12:12 GMT
Server: CouchDB/3.2.2-ea382cf (Erlang OTP/25)

{
    "javascript_engine": {
        "version": "91",
        "name": "spidermonkey"
    },
    "erlang": {
        "version": "25.0.4",
        "supported_hashes": [
            "sha",
            "sha224",
            "sha256",
        ]
    },
    "collation_driver": {
        "name": "libicu",
        "library_version": "70.1",
        "collator_version": "153.112",
        "collation_algorithm_version": "14"
    }
}
```

12.2.18 /_search_analyze

> **Warning:** Search endpoints require a running search plugin connected to each cluster node. See *Search Plugin Installation* for details.

New in version 3.0.

POST /_search_analyze

Tests the results of Lucene analyzer tokenization on sample text.

Parameters

- **analyzer** – Type of analyzer
- **text** – Analyzer token you want to test

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Request body is wrong (malformed or missing one of the mandatory fields)
- 500 Internal Server Error – A server error (or other kind of error) occurred

Request:

```
POST /_search_analyze HTTP/1.1
Host: localhost:5984
```

(continues on next page)

(continued from previous page)

```
Content-Type: application/json

{"analyzer":"english", "text":"running"}
```

Response:

```
{
    "tokens": [
        "run"
    ]
}
```

12.2.19 /_nouveau_analyze

> **Warning:** Nouveau is an experimental feature. Future releases might change how the endpoints work and might invalidate existing indexes.

> **Warning:** Nouveau endpoints require a running nouveau server. See *Nouveau Server Installation* for details.

New in version 4.0.

POST /_nouveau_analyze

Tests the results of Lucene analyzer tokenization on sample text.

Parameters

- **analyzer** – Name of analyzer
- **text** – Analyzer token you want to test

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Request body is wrong (malformed or missing one of the mandatory fields)
- 500 Internal Server Error – A server error (or other kind of error) occurred

Request:

```
POST /_nouveau_analyze HTTP/1.1
Host: localhost:5984
Content-Type: application/json

{"analyzer":"english", "text":"running"}
```

Response:

```
{
    "tokens": [
        "run"
    ]
}
```

12.2.20 /_utils

GET /_utils

Accesses the built-in Fauxton administration interface for CouchDB.

Response Headers

- Location – New URI location

Status Codes

- 301 Moved Permanently – Redirects to *GET /_utils/*

GET /_utils/

Response Headers

- Content-Type – *text/html*
- Last-Modified – Static files modification timestamp

Status Codes

- 200 OK – Request completed successfully

12.2.21 /_up

New in version 2.0.

GET /_up

Confirms that the server is up, running, and ready to respond to requests. If *maintenance_mode* is `true` or `nolb`, the endpoint will return a 404 response.

Response Headers

- Content-Type – *application/json*

Status Codes

- 200 OK – Request completed successfully
- 404 Not Found – The server is unavailable for requests at this time.

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 16
Content-Type: application/json
Date: Sat, 17 Mar 2018 04:46:26 GMT
Server: CouchDB/2.2.0-f999071ec (Erlang OTP/19)
X-Couch-Request-ID: c57a3b2787
X-CouchDB-Body-Time: 0

{"status":"ok"}
```

12.2.22 /_uuids

Changed in version 2.0.0.

GET /_uuids

Requests one or more Universally Unique Identifiers (UUIDs) from the CouchDB instance. The response is a JSON object providing a list of UUIDs.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **count** (*number*) – Number of UUIDs to return. Default is 1.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- ETag – Response hash

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Requested more UUIDs than is *allowed* to retrieve

Request:

```
GET /_uuids?count=10 HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 362
Content-Type: application/json
Date: Sat, 10 Aug 2013 11:46:25 GMT
ETag: "DGRWWQFLUDWN5MRKSLKQ425XV"
Expires: Fri, 01 Jan 1990 00:00:00 GMT
Pragma: no-cache
Server: CouchDB (Erlang/OTP)

{
    "uuids": [
        "75480ca477454894678e22eec6002413",
        "75480ca477454894678e22eec600250b",
        "75480ca477454894678e22eec6002c41",
        "75480ca477454894678e22eec6003b90",
        "75480ca477454894678e22eec6003fca",
        "75480ca477454894678e22eec6004bef",
        "75480ca477454894678e22eec600528f",
        "75480ca477454894678e22eec6005e0b",
        "75480ca477454894678e22eec6006158",
        "75480ca477454894678e22eec6006161"
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

The UUID type is determined by the *UUID algorithm* setting in the CouchDB configuration.

The UUID type may be changed at any time through the *Configuration API*. For example, the UUID type could be changed to `random` by sending this HTTP request:

```
PUT http://couchdb:5984/_node/nonode@nohost/_config/uuids/algorithm HTTP/1.1
Content-Type: application/json
Accept: */*

"random"
```

You can verify the change by obtaining a list of UUIDs:

```
{
    "uuids" : [
        "031aad7b469956cf2826fcb2a9260492",
        "6ec875e15e6b385120938df18ee8e496",
        "cff9e881516483911aa2f0e98949092d",
        "b89d37509d39dd712546f9510d4a9271",
        "2e0dbf7f6c4ad716f21938a016e4e59f"
    ]
}
```

12.2.23 /favicon.ico

GET /favicon.ico

Binary content for the *favicon.ico* site icon.

Response Headers

- Content-Type – *image/x-icon*

Status Codes

- 200 OK – Request completed successfully
- 404 Not Found – The requested content could not be found

12.2.24 /_reshard

New in version 2.4.

GET /_reshard

Returns a count of completed, failed, running, stopped, and total jobs along with the state of resharding on the cluster.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Response JSON Object

- **state** (*string*) – stopped or running
- **state_reason** (*string*) – null or string describing additional information or reason associated with the state
- **completed** (*number*) – Count of completed resharding jobs
- **failed** (*number*) – Count of failed resharding jobs
- **running** (*number*) – Count of running resharding jobs
- **stopped** (*number*) – Count of stopped resharding jobs
- **total** (*number*) – Total count of resharding jobs

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_reshard HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "completed": 21,
    "failed": 0,
    "running": 3,
    "state": "running",
    "state_reason": null,
    "stopped": 0,
    "total": 24
}
```

`GET /_reshard/state`

Returns the resharding state and optional information about the state.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Response JSON Object

- **state** (*string*) – stopped or running
- **state_reason** (*string*) – Additional information or reason associated with the state

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_reshard/state HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "reason": null,
    "state": "running"
}
```

`PUT /_reshard/state`

Change the resharding state on the cluster. The states are `stopped` or `running`. This starts and stops global resharding on all the nodes of the cluster. If there are any running jobs, they will be stopped when the state changes to `stopped`. When the state changes back to `running` those job will continue running.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Request JSON Object

- **state** (*string*) – `stopped` or `running`
- **state_reason** (*string*) – Optional string describing additional information or reason associated with the state

Response JSON Object

- **ok** (*boolean*) – `true`

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request. Could be a bad or missing state name.
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
PUT /_reshard/state HTTP/1.1
Accept: application/json
Host: localhost:5984

{
    "state": "stopped",
    "reason": "Rebalancing in progress"
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "ok": true
}
```

GET /_reshard/jobs

Note: The shape of the response and the `total_rows` and `offset` field in particular are meant to be consistent with the `_scheduler/jobs` endpoint.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Response JSON Object

- **jobs** (*list*) – Array of json objects, one for each resharding job. For the fields of each job see the /_reshard/job/{jobid} endpoint.
- **offset** (*number*) – Offset in the list of jobs object. Currently hard-coded at `0`.
- **total_rows** (*number*) – Total number of resharding jobs on the cluster.

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_reshard/jobs HTTP/1.1
Accept: application/json
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "jobs": [
        {
            "history": [
                {
                    "detail": null,
                    "timestamp": "2019-03-28T15:28:02Z",
                    "type": "new"
                },
                {
                    "detail": "initial_copy",
                    "timestamp": "2019-03-28T15:28:02Z",
```

(continues on next page)

(continued from previous page)

```
                    "type": "running"
                }
            ],
            "id": "001-
↪171d1211418996ff47bd610b1d1257fc4ca2628868def4a05e63e8f8fe50694a",
            "job_state": "completed",
            "node": "node1@127.0.0.1",
            "source": "shards/00000000-1fffffff/d1.1553786862",
            "split_state": "completed",
            "start_time": "2019-03-28T15:28:02Z",
            "state_info": {},
            "target": [
                "shards/00000000-0fffffff/d1.1553786862",
                "shards/10000000-1fffffff/d1.1553786862"
            ],
            "type": "split",
            "update_time": "2019-03-28T15:28:08Z"
        }
    ],
    "offset": 0,
    "total_rows": 24
}
```

GET /_reshard/jobs/{jobid}

Get information about the resharding job identified by `jobid`.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Response JSON Object

- **id** (*string*) – Job ID.
- **type** (*string*) – Currently only `split` is implemented.
- **job_state** (*string*) – The running state of the job. Could be one of `new`, `running`, `stopped`, `completed` or `failed`.
- **split_state** (*string*) – State detail specific to shard splitting. It indicates how far has shard splitting progressed, and can be one of `new`, `initial_copy`, `topoff1`, `build_indices`, `topoff2`, `copy_local_docs`, `update_shardmap`, `wait_source_close`, `topoff3`, `source_delete` or `completed`.
- **state_info** (*object*) – Optional additional info associated with the current state.
- **source** (*string*) – For `split` jobs this will be the source shard.
- **target** (*list*) – For `split` jobs this will be a list of two or more target shards.
- **history** (*list*) – List of json objects recording a job's state transition history.

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_reshard/jobs/001-
↪171d1211418996ff47bd610b1d1257fc4ca2628868def4a05e63e8f8fe50694a HTTP/1.1
Accept: application/json
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{

    "id": "001-171d1211418996ff47bd610b1d1257fc4ca2628868def4a05e63e8f8fe50694a",
    "job_state": "completed",
    "node": "node1@127.0.0.1",
    "source": "shards/00000000-1fffffff/d1.1553786862",
    "split_state": "completed",
    "start_time": "2019-03-28T15:28:02Z",
    "state_info": {},
    "target": [
        "shards/00000000-0fffffff/d1.1553786862",
        "shards/10000000-1fffffff/d1.1553786862"
    ],
    "type": "split",
    "update_time": "2019-03-28T15:28:08Z",
    "history": [
        {
            "detail": null,
            "timestamp": "2019-03-28T15:28:02Z",
            "type": "new"
        },
        {
            "detail": "initial_copy",
            "timestamp": "2019-03-28T15:28:02Z",
            "type": "running"
        }
    ]
}
```

`POST /_reshard/jobs`

Depending on what fields are specified in the request, one or more resharding jobs will be created. The response is a json array of results. Each result object represents a single resharding job for a particular node and range. Some of the responses could be successful and some could fail. Successful results will have the `"ok": true` key and and value, and failed jobs will have the `"error": "{error_message}"` key and value.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Request JSON Object

- **type** (*string*) – Type of job. Currently only `"split"` is accepted.

- **db** (*string*) – Database to split. This is mutually exclusive with the "shard" field.
- **node** (*string*) – Split shards on a particular node. This is an optional parameter. The value should be one of the nodes returned from the `_membership` endpoint.
- **range** (*string*) – Split shards copies in the given range. The range format is `hhhhhhhh-hhhhhhhh` where `h` is a hexadecimal digit. This format is used since this is how the ranges are represented in the file system. This is parameter is optional and is mutually exclusive with the "shard" field.
- **shard** (*string*) – Split a particular shard. The shard should be specified as "shards/{range}/{db}.{suffix}". Where `range` has the `hhhhhhhh-hhhhhhhh` format, `db` is the database name, and `suffix` is the shard (timestamp) creation suffix.
- **error** (*string*) – Error message if a job could be not be created.
- **node** – Cluster node where the job was created and is running.

Response JSON Object

- **ok** (*boolean*) – `true` if job created successfully.

Status Codes

- 201 Created – One or more jobs were successfully created
- 400 Bad Request – Invalid request. Parameter validation might have failed.
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – Db, node, range or shard was not found

Request:

```
POST /_reshard/jobs HTTP/1.1
Accept: application/json
Content-Type: application/json

{
   "db": "db3",
   "range": "80000000-ffffffff",
   "type": "split"
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

[
    {
        "id": "001-
↪30d7848a6feeb826d5e3ea5bb7773d672af226fd34fd84a8fb1ca736285df557",
        "node": "node1@127.0.0.1",
        "ok": true,
        "shard": "shards/80000000-ffffffff/db3.1554148353"
    },
    {
        "id": "001-
↪c2d734360b4cb3ff8b3feaccb2d787bf81ce2e773489eddd985ddd01d9de8e01",
        "node": "node2@127.0.0.1",
        "ok": true,
        "shard": "shards/80000000-ffffffff/db3.1554148353"
    }
]
```

DELETE /_reshard/jobs/{jobid}

If the job is running, stop the job and then remove it.

Response JSON Object

- **ok** (*boolean*) – `true` if the job was removed successfully.

Status Codes

- 200 OK – The job was removed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – The job was not found

Request:

```
DELETE /_reshard/jobs/001-
↪171d1211418996ff47bd610b1d1257fc4ca2628868def4a05e63e8f8fe50694a HTTP/1.1
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "ok": true
}
```

GET /_reshard/jobs/{jobid}/state

Returns the running state of a resharding job identified by `jobid`.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Request JSON Object

- **state** (*string*) – One of `new`, `running`, `stopped`, `completed` or `failed`.
- **state_reason** (*string*) – Additional information associated with the state

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – The job was not found

Request:

```
GET /_reshard/jobs/001-
↪b3da04f969bbd682faaab5a6c373705cbcca23f732c386bb1a608cfbcfe9faff/state HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "reason": null,
    "state": "running"
}
```

PUT /_reshard/jobs/{jobid}/state

Change the state of a particular resharding job identified by `jobid`. The state can be changed from `stopped` to `running` or from `running` to `stopped`. If an individual job is `stopped` via this API it will stay `stopped` even after the global resharding state is toggled from `stopped` to `running`. If the job is already `completed` its state will stay `completed`.

Request Headers

- Accept –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*

Request JSON Object

- **state** (*string*) – `stopped` or `running`
- **state_reason** (*string*) – Optional string describing additional information or reason associated with the state

Response JSON Object

- **ok** (*boolean*) – `true`

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request. Could be a bad state name, for example.
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – The job was not found

Request:

```
PUT /_reshard/state/001-
↪b3da04f969bbd682faaab5a6c373705cbcca23f732c386bb1a608cfbcfe9faff/state HTTP/1.1
Accept: application/json
Host: localhost:5984

{
    "state": "stopped",
    "reason": "Rebalancing in progress"
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
```

(continues on next page)

(continued from previous page)

```
    "ok": true
}
```

12.2.25 Authentication

Interfaces for obtaining session and authorization data.

Note: We also strongly recommend you *set up SSL* to improve all authentication methods' security.

Basic Authentication

Changed in version 3.4: In order to aid transition to stronger password hashing without causing a performance penalty, CouchDB will send a Set-Cookie header when a request authenticates successfully with Basic authentication. All browsers and many http libraries will automatically send this cookie on subsequent requests. The cost of verifying the cookie is significantly less than PBKDF2 with a high iteration count, for example.

Basic authentication (**RFC 2617**) is a quick and simple way to authenticate with CouchDB. The main drawback is the need to send user credentials with each request which may be insecure and could hurt operation performance (since CouchDB must compute the password hash with every request):

Request:

```
GET / HTTP/1.1
Accept: application/json
Authorization: Basic cm9vdDpyZWxheA==
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 177
Content-Type: application/json
Date: Mon, 03 Dec 2012 00:44:47 GMT
Server: CouchDB (Erlang/OTP)

{
    "couchdb":"Welcome",
    "uuid":"0a959b9b8227188afc2ac26ccdf345a6",
    "version":"1.3.0",
    "vendor": {
        "version":"1.3.0",
        "name":"The Apache Software Foundation"
    }
}
```

Cookie Authentication

For cookie authentication (**RFC 2109**) CouchDB generates a token that the client can use for the next few requests to CouchDB. Tokens are valid until a timeout. When CouchDB sees a valid token in a subsequent request, it will authenticate the user by this token without requesting the password again. By default, cookies are valid for 10 minutes, but it's adjustable via `timeout`. Also it's possible to make cookies `persistent`.

To obtain the first token and thus authenticate a user for the first time, the `username` and `password` must be sent to the *_session API*.

/_session

POST /_session

Initiates new session for specified user credentials by providing *Cookie* value.

Request Headers

- Content-Type –
  - *application/x-www-form-urlencoded*
  - *application/json*

Query Parameters

- **next** (*string*) – Enforces redirect after successful login to the specified location. This location is relative from server root. *Optional*.

Form Parameters

- **name** – User name
- **password** – Password

Response Headers

- Set-Cookie – Authorization token

Response JSON Object

- **ok** (*boolean*) – Operation status
- **name** (*string*) – Username
- **roles** (*array*) – List of user roles

Status Codes

- 200 OK – Successfully authenticated
- 302 Found – Redirect after successful authentication
- 401 Unauthorized – Username or password wasn't recognized

Request:

```
POST /_session HTTP/1.1
Accept: application/json
Content-Length: 24
Content-Type: application/x-www-form-urlencoded
Host: localhost:5984

name=root&password=relax
```

It's also possible to send data as JSON:

```
POST /_session HTTP/1.1
Accept: application/json
Content-Length: 37
Content-Type: application/json
Host: localhost:5984

{
    "name": "root",
    "password": "relax"
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 43
Content-Type: application/json
Date: Mon, 03 Dec 2012 01:23:14 GMT
Server: CouchDB (Erlang/OTP)
Set-Cookie: AuthSession=cm9vdDo1MEJCRkYwMjq0LO0ylOIwShrgt8y-UkhI-c6BGw;␣
↪Version=1; Path=/; HttpOnly

{"ok":true,"name":"root","roles":["_admin"]}
```

If `next` query parameter was provided the response will trigger redirection to the specified location in case of successful authentication:

Request:

```
POST /_session?next=/blog/_design/sofa/_rewrite/recent-posts HTTP/1.1
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Host: localhost:5984

name=root&password=relax
```

Response:

```
HTTP/1.1 302 Moved Temporarily
Cache-Control: must-revalidate
Content-Length: 43
Content-Type: application/json
Date: Mon, 03 Dec 2012 01:32:46 GMT
Location: http://localhost:5984/blog/_design/sofa/_rewrite/recent-posts
Server: CouchDB (Erlang/OTP)
Set-Cookie: AuthSession=cm9vdDo1MEJDMDEzRTp7Vu5GKCkTxTVxwXbpXsBARQWnhQ;␣
↪Version=1; Path=/; HttpOnly

{"ok":true,"name":null,"roles":["_admin"]}
```

`GET /_session`

Returns information about the authenticated user, including a *User Context Object*, the authentication method and database that were used, and a list of configured authentication handlers on the server.

Query Parameters

- **basic** (*boolean*) – Accept *Basic Auth* by requesting this resource. *Optional*.

Response JSON Object

- **ok** (*boolean*) – Operation status

- **userCtx** (*object*) – User context for the current user
- **info** (*object*) – Server authentication configuration

Status Codes

- 200 OK – Successfully authenticated.
- 401 Unauthorized – Username or password wasn't recognized.

Request:

```
GET /_session HTTP/1.1
Host: localhost:5984
Accept: application/json
Cookie: AuthSession=cm9vdDo1MEJDMDQxRDpqb-Ta9QfP9hpdPjHLxNTKg_Hf9w
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 175
Content-Type: application/json
Date: Fri, 09 Aug 2013 20:27:45 GMT
Server: CouchDB (Erlang/OTP)
Set-Cookie: AuthSession=cm9vdDo1MjA1NTBDMTqmX2qKt1KDR--GUC80DQ6-Ew_XIw;␣
↪Version=1; Path=/; HttpOnly

{
    "info": {
        "authenticated": "cookie",
        "authentication_db": "_users",
        "authentication_handlers": [
            "cookie",
            "default"
        ]
    },
    "ok": true,
    "userCtx": {
        "name": "root",
        "roles": [
            "_admin"
        ]
    }
}
```

DELETE /_session

Closes user's session by instructing the browser to clear the cookie. This does not invalidate the session from the server's perspective, as there is no way to do this because CouchDB cookies are stateless. This means calling this endpoint is purely optional from a client perspective, and it does not protect against theft of a session cookie.

Status Codes

- 200 OK – Successfully close session.

Request:

```
DELETE /_session HTTP/1.1
Accept: application/json
Cookie: AuthSession=cm9vdDo1MjA1NEVGMDo1QXNQkqC_0Qmgrk8Fw61_AzDeXw
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Fri, 09 Aug 2013 20:30:12 GMT
Server: CouchDB (Erlang/OTP)
Set-Cookie: AuthSession=; Version=1; Path=/; HttpOnly

{
    "ok": true
}
```

Proxy Authentication

Note: To use this authentication method make sure that the `{chttpd_auth, proxy_authentication_handler}` value is added to the list of the active *chttpd/authentication_handlers*:

```
[chttpd]
authentication_handlers = {chttpd_auth, cookie_authentication_handler}, {chttpd_auth,␣
↪proxy_authentication_handler}, {chttpd_auth, default_authentication_handler}
```

Proxy authentication is very useful in case your application already uses some external authentication service and you don't want to duplicate users and their roles in CouchDB.

This authentication method allows creation of a *User Context Object* for remotely authenticated user. By default, the client just needs to pass specific headers to CouchDB with related requests:

- *X-Auth-CouchDB-UserName*: username
- *X-Auth-CouchDB-Roles*: comma-separated (`,`) list of user roles
- *X-Auth-CouchDB-Token*: authentication token. When *proxy_use_secret* is set (which is strongly recommended!), this header provides an HMAC of the username to authenticate and the secret token to prevent requests from untrusted sources. (Use one of the configured hash algorithms in *chttpd_auth/hash_algorithms* and sign the username with the secret)

Creating the token (example with openssl):

```
echo -n "foo" | openssl dgst -sha256 -hmac "the_secret"
# (stdin)= 3f0786e96b20b0102b77f1a49c041be6977cfb3bf78c41a12adc121cd9b4e68a
```

Request:

```
GET /_session HTTP/1.1
Host: localhost:5984
Accept: application/json
Content-Type: application/json; charset=utf-8
X-Auth-CouchDB-Roles: users,blogger
X-Auth-CouchDB-UserName: foo
X-Auth-CouchDB-Token: 3f0786e96b20b0102b77f1a49c041be6977cfb3bf78c41a12adc121cd9b4e68a
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
```

(continues on next page)

(continued from previous page)

```
Content-Length: 190
Content-Type: application/json
Date: Fri, 14 Jun 2013 10:16:03 GMT
Server: CouchDB (Erlang/OTP)

{
    "info": {
        "authenticated": "proxy",
        "authentication_db": "_users",
        "authentication_handlers": [
            "cookie",
            "proxy",
            "default"
        ]
    },
    "ok": true,
    "userCtx": {
        "name": "foo",
        "roles": [
            "users",
            "blogger"
        ]
    }
}
```

Note that you don't need to request a *session* to be authenticated by this method if all required HTTP headers are provided.

JWT Authentication

Note: To use this authentication method, make sure that the `{chttpd_auth, jwt_authentication_handler}` value is added to the list of the active *chttpd/authentication_handlers*:

```
[chttpd]
authentication_handlers = {chttpd_auth, cookie_authentication_handler}, {chttpd_auth,
↪jwt_authentication_handler}, {chttpd_auth, default_authentication_handler}
```

`JWT authentication` enables CouchDB to use externally-generated JWT tokens instead of defining users or roles in the `_users` database.

The JWT authentication handler requires that all JWT tokens are signed by a key that CouchDB has been configured to trust (there is no support for JWT's "NONE" algorithm).

Additionally, CouchDB can be configured to reject JWT tokens that are missing a configurable set of claims (e.g, a CouchDB administrator could insist on the `exp` claim).

Only claims listed in required checks are validated. Additional claims will be ignored.

Two sections of config exist to configure JWT authentication;

The *required_claims* config setting is a comma-separated list of additional mandatory JWT claims that must be present in any presented JWT token. A 400 Bad Request is sent if any are missing.

The `alg` claim is mandatory as it used to lookup the correct key for verifying the signature.

The `sub` claim is mandatory and is used as the CouchDB user's name if the JWT token is valid.

You can set the user roles claim name through the config setting *roles_claim_name*. If you don't set an explicit value, then `_couchdb.roles` will be set as the default claim name. If presented, it is used as the CouchDB user's roles list as long as the JWT token is valid.

Note: Before CouchDB v3.3.2 it was only possible to define roles as a JSON array of strings. Now you can also use a comma-seperated list to define the user roles in your JWT token. The following declarations are equal:

JSON array of strings:

```
{
    "_couchdb.roles": ["accounting-role", "view-role"]
}
```

JSON comma-seperated strings:

```
{
    "_couchdb.roles": "accounting-role, view-role"
}
```

Warning: `roles_claim_name` is deprecated in CouchDB 3.3, and will be removed later. Please use *roles_claim_path*.

```
; [jwt_keys]
; Configure at least one key here if using the JWT auth handler.
; If your JWT tokens do not include a "kid" attribute, use "_default"
; as the config key, otherwise use the kid as the config key.
; Examples
; hmac:_default = aGVsbG8=
; hmac:foo = aGVsbG8=
; The config values can represent symmetric and asymmetric keys.
; For symmetric keys, the value is base64 encoded;
; hmac:_default = aGVsbG8= # base64-encoded form of "hello"
; For asymmetric keys, the value is the PEM encoding of the public
; key with newlines replaced with the escape sequence \n.
; rsa:foo = -----BEGIN PUBLIC KEY-----\nMIIBIjAN...IDAQAB\n-----END PUBLIC KEY-----\n
; ec:bar = -----BEGIN PUBLIC KEY-----\nMHYwEAYHK...AzztRs\n-----END PUBLIC KEY-----\n
```

The `jwt_keys` section lists all the keys that this CouchDB server trusts. You should ensure that all nodes of your cluster have the same list.

Since version 3.3 it's possible to use = in parameter names, but only when the parameter and value are separated = , i.e. the equal sign is surrounded by at least one space on each side. This might be useful in the `[jwt_keys]` section where base64 encoded keys may contain the = character.

JWT tokens that do not include a `kid` claim will be validated against the `{alg}:_default` key.

It is mandatory to specify the algorithm associated with every key for security reasons (notably presenting a HMAC-signed token using an RSA or EC public key that the server trusts: https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/).

Request:

```
GET /_session HTTP/1.1
Host: localhost:5984
Accept: application/json
Content-Type: application/json; charset=utf-8
Authorization: Bearer <JWT token>
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 188
Content-Type: application/json
Date: Sun, 19 Apr 2020 08:29:15 GMT
Server: CouchDB (Erlang/OTP)

{
    "info": {
        "authenticated": "jwt",
        "authentication_db": "_users",
        "authentication_handlers": [
            "cookie",
            "proxy",
            "default"
        ]
    },
    "ok": true,
    "userCtx": {
        "name": "foo",
        "roles": [
            "users",
            "blogger"
        ]
    }
}
```

Note that you don't need to request *session* to be authenticated by this method if the required HTTP header is provided.

12.2.26 Configuration

The CouchDB Server Configuration API provide an interface to query and update the various configuration values within a running CouchDB instance.

Accessing the local node's configuration

The literal string `_local` serves as an alias for the local node name, so for all configuration URLs, `{node-name}` may be replaced with `_local`, to interact with the local node's configuration.

`/_node/{node-name}/_config`

GET /_node/{node-name}/_config

Returns the entire CouchDB server configuration as a JSON structure. The structure is organized by different configuration sections, with individual values.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –

- *application/json*
- *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request

```
GET /_node/nonode@nohost/_config HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 4148
Content-Type: application/json
Date: Sat, 10 Aug 2013 12:01:42 GMT
Server: CouchDB (Erlang/OTP)

{
    "attachments": {
        "compressible_types": "text/*, application/javascript, application/json,␣
↪ application/xml",
        "compression_level": "8"
    },
    "couchdb": {
        "users_db_suffix": "_users",
        "database_dir": "/var/lib/couchdb",
        "max_attachment_chunk_size": "4294967296",
        "max_dbs_open": "100",
        "os_process_timeout": "5000",
        "uri_file": "/var/lib/couchdb/couch.uri",
        "util_driver_dir": "/usr/lib64/couchdb/erlang/lib/couch-1.5.0/priv/lib",
        "view_index_dir": "/var/lib/couchdb"
    },
    "chttpd": {
        "allow_jsonp": "false",
        "backlog": "512",
        "bind_address": "0.0.0.0",
        "port": "5984",
        "require_valid_user": "false",
        "socket_options": "[{sndbuf, 262144}, {nodelay, true}]",
        "server_options": "[{recbuf, undefined}]",
        "secure_rewrites": "true"
    },
    "httpd": {
        "authentication_handlers": "{couch_httpd_auth, cookie_authentication_
↪handler}, {couch_httpd_auth, default_authentication_handler}",
        "bind_address": "192.168.0.2",
        "max_connections": "2048",
        "port": "5984",
    },
    "log": {
        "writer": "file",
```

(continues on next page)

(continued from previous page)

```
        "file": "/var/log/couchdb/couch.log",
        "include_sasl": "true",
        "level": "info"
    },
    "query_server_config": {
        "reduce_limit": "true"
    },
    "replicator": {
        "max_http_pipeline_size": "10",
        "max_http_sessions": "10"
    },
    "stats": {
        "interval": "10"
    },
    "uuids": {
        "algorithm": "utc_random"
    }
}
```

/_node/{node-name}/_config/{section}

GET /_node/{node-name}/_config/{section}

Gets the configuration structure for a single section.

Parameters

- **section** – Configuration section name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_node/nonode@nohost/_config/httpd HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 444
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
Date: Sat, 10 Aug 2013 12:10:40 GMT
Server: CouchDB (Erlang/OTP)

{
    "authentication_handlers": "{couch_httpd_auth, cookie_authentication_handler}
↪, {couch_httpd_auth, default_authentication_handler}",
    "bind_address": "127.0.0.1",
    "default_handler": "{couch_httpd_db, handle_request}",
    "port": "5984"
}
```

/_node/{node-name}/_config/{section}/{key}

GET /_node/{node-name}/_config/{section}/{key}

Gets a single configuration value from within a specific configuration section.

Parameters

- **section** – Configuration section name
- **key** – Configuration option name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
GET /_node/nonode@nohost/_config/log/level HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 8
Content-Type: application/json
Date: Sat, 10 Aug 2013 12:12:59 GMT
Server: CouchDB (Erlang/OTP)

"debug"
```

Note: The returned value will be the JSON of the value, which may be a string or numeric value, or an array or object. Some client environments may not parse simple strings or numeric values as valid JSON.

`PUT /_node/{node-name}/_config/{section}/{key}`

Updates a configuration value. The new value should be supplied in the request body in the corresponding JSON format. If you are setting a string value, you must supply a valid JSON string. In response CouchDB sends old value for target section key.

Parameters

- **section** – Configuration section name
- **key** – Configuration option name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid JSON request body
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 500 Internal Server Error – Error setting configuration

Request:

```
PUT /_node/nonode@nohost/_config/log/level HTTP/1.1
Accept: application/json
Content-Length: 7
Content-Type: application/json
Host: localhost:5984

"info"
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 8
Content-Type: application/json
Date: Sat, 10 Aug 2013 12:12:59 GMT
Server: CouchDB (Erlang/OTP)

"debug"
```

`DELETE /_node/{node-name}/_config/{section}/{key}`

Deletes a configuration value. The returned JSON will be the value of the configuration parameter before it was deleted.

Parameters

- **section** – Configuration section name
- **key** – Configuration option name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – Specified configuration option not found

Request:

```
DELETE /_node/nonode@nohost/_config/log/level HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 7
Content-Type: application/json
Date: Sat, 10 Aug 2013 12:29:03 GMT
Server: CouchDB (Erlang/OTP)

"info"
```

/_node/{node-name}/_config/_reload

New in version 3.0.

POST /_node/{node-name}/_config/_reload

Reloads the configuration from disk. This has a side effect of flushing any in-memory configuration changes that have not been committed to disk.

Request:

```
POST /_node/nonode@nohost/_config/_reload HTTP/1.1
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 12
```

(continues on next page)

(continued from previous page)

```
Content-Type: application/json
Date: Tues, 21 Jan 2020 11:09:35
Server: CouchDB/3.0.0 (Erlang OTP)

{"ok":true}
```

12.3 Databases

The Database endpoint provides an interface to an entire database with in CouchDB. These are database-level, rather than document-level requests.

For all these requests, the database name within the URL path should be the database name that you wish to perform the operation on. For example, to obtain the meta information for the database `recipes`, you would use the HTTP request:

```
GET /recipes
```

For clarity, the form below is used in the URL paths:

```
GET /{db}
```

Where `{db}` is the name of any database.

12.3.1 /{db}

HEAD /{db}

Returns the HTTP Headers containing a minimal amount of information about the specified database. Since the response body is empty, using the HEAD method is a lightweight way to check if the database exists already or not.

Parameters

- **db** – Database name

Status Codes

- 200 OK – Database exists
- 404 Not Found – Requested database not found

Request:

```
HEAD /test HTTP/1.1
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Mon, 12 Aug 2013 01:27:41 GMT
Server: CouchDB (Erlang/OTP)
```

GET /{db}

Gets information about the specified database.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **cluster.n** (*number*) – Replicas. The number of copies of every document.
- **cluster.q** (*number*) – Shards. The number of range partitions.
- **cluster.r** (*number*) – Read quorum. The number of consistent copies of a document that need to be read before a successful reply.
- **cluster.w** (*number*) – Write quorum. The number of copies of a document that need to be written before a successful reply.
- **compact_running** (*boolean*) – Set to `true` if the database compaction routine is operating on this database.
- **db_name** (*string*) – The name of the database.
- **disk_format_version** (*number*) – The version of the physical format used for the data when it is stored on disk.
- **doc_count** (*number*) – A count of the documents in the specified database.
- **doc_del_count** (*number*) – Number of deleted documents
- **instance_start_time** (*string*) – Always `"0"`. (Returned for legacy reasons.)
- **purge_seq** (*string*) – An opaque string that describes the purge state of the database. Do not rely on this string for counting the number of purge operations.
- **sizes.active** (*number*) – The size of live data inside the database, in bytes.
- **sizes.external** (*number*) – The uncompressed size of database contents in bytes.
- **sizes.file** (*number*) – The size of the database file on disk in bytes. Views indexes are not included in the calculation.
- **update_seq** (*string*) – An opaque string that describes the state of the database. Do not rely on this string for counting the number of updates.
- **props.partitioned** (*boolean*) – (optional) If present and true, this indicates that the database is partitioned.

Status Codes

- 200 OK – Request completed successfully
- 404 Not Found – Requested database not found

Request:

```
GET /receipts HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 258
Content-Type: application/json
Date: Mon, 12 Aug 2013 01:38:57 GMT
Server: CouchDB (Erlang/OTP)

{
    "cluster": {
        "n": 3,
        "q": 8,
        "r": 2,
        "w": 2
    },
    "compact_running": false,
    "db_name": "receipts",
    "disk_format_version": 6,
    "doc_count": 6146,
    "doc_del_count": 64637,
    "instance_start_time": "0",
    "props": {},
    "purge_seq": 0,
    "sizes": {
        "active": 65031503,
        "external": 66982448,
        "file": 137433211
    },
    "update_seq": "292786-g1AAAAF..."
}
```

PUT /{db}

Creates a new database. The database name `{db}` must be composed by following next rules:

- Name must begin with a lowercase letter (`a-z`)
- Lowercase characters (`a-z`)
- Digits (`0-9`)
- Any of the characters `_`, `$`, `(`, `)`, `+`, `-`, and `/`.

If you're familiar with Regular Expressions, the rules above could be written as `^[a-z][a-z0-9_$()+/-]*$`.

Parameters

- **db** – Database name

Query Parameters

- **q** (*integer*) – Shards, aka the number of range partitions. Default is 8, unless overridden in the *cluster config*.
- **n** (*integer*) – Replicas. The number of copies of the database in the cluster. The default is 3, unless overridden in the *cluster config* .
- **partitioned** (*boolean*) – Whether to create a partitioned database. Default is false.

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- Location – Database URI location

Response JSON Object

- **ok** (*boolean*) – Operation status. Available in case of success
- **error** (*string*) – Error type. Available if response code is `4xx`
- **reason** (*string*) – Error description. Available if response code is `4xx`

Status Codes

- 201 Created – Database created successfully (quorum is met)
- 202 Accepted – Accepted (at least by one node)
- 400 Bad Request – Invalid database name
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 412 Precondition Failed – Database already exists

Request:

```
PUT /db HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Mon, 12 Aug 2013 08:01:45 GMT
Location: http://localhost:5984/db
Server: CouchDB (Erlang/OTP)

{
    "ok": true
}
```

If we repeat the same request to CouchDB, it will response with `412` since the database already exists:

Request:

```
PUT /db HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 412 Precondition Failed
Cache-Control: must-revalidate
Content-Length: 95
Content-Type: application/json
Date: Mon, 12 Aug 2013 08:01:16 GMT
Server: CouchDB (Erlang/OTP)
```

(continues on next page)

(continued from previous page)

```
{
    "error": "file_exists",
    "reason": "The database could not be created, the file already exists."
}
```

If an invalid database name is supplied, CouchDB returns response with `400`:

Request:

```
PUT /_db HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Request:

```
HTTP/1.1 400 Bad Request
Cache-Control: must-revalidate
Content-Length: 194
Content-Type: application/json
Date: Mon, 12 Aug 2013 08:02:10 GMT
Server: CouchDB (Erlang/OTP)

{
    "error": "illegal_database_name",
    "reason": "Name: '_db'. Only lowercase characters (a-z), digits (0-9), and␣
↪any of the characters _, $, (, ), +, -, and / are allowed. Must begin with a␣
↪letter."
}
```

`DELETE /{db}`

Deletes the specified database, and all the documents and attachments contained within it.

Note: To avoid deleting a database, CouchDB will respond with the HTTP status code 400 when the request URL includes a ?rev= parameter. This suggests that one wants to delete a document but forgot to add the document id to the URL.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **ok** (*boolean*) – Operation status

Status Codes

- 200 OK – Database removed successfully (quorum is met and database is deleted by at least one node)
- 202 Accepted – Accepted (deleted by at least one of the nodes, quorum is not met yet)
- 400 Bad Request – Invalid database name or forgotten document id by accident
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – Database doesn't exist or invalid database name

Request:

```
DELETE /db HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Mon, 12 Aug 2013 08:54:00 GMT
Server: CouchDB (Erlang/OTP)

{
    "ok": true
}
```

`POST /{db}`

Creates a new document in the specified database, using the supplied JSON document structure.

If the JSON structure includes the `_id` field, then the document will be created with the specified document ID.

If the `_id` field is not specified, a new unique ID will be generated, following whatever UUID algorithm is configured for that server.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Query Parameters

- **batch** (*string*) – Stores document in *batch mode* Possible values: `ok`. *Optional*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- Location – Document's URI

Response JSON Object

- **id** (*string*) – Document ID
- **ok** (*boolean*) – Operation status
- **rev** (*string*) – Revision info

Status Codes

- 201 Created – Document created and stored on disk
- 202 Accepted – Document data accepted, but not yet stored on disk
- 400 Bad Request – Invalid database name
- 401 Unauthorized – Write privileges required
- 404 Not Found – Database doesn't exist
- 409 Conflict – A Conflicting Document with same ID already exists

Request:

```
POST /db HTTP/1.1
Accept: application/json
Content-Length: 81
Content-Type: application/json

{
    "servings": 4,
    "subtitle": "Delicious with fresh bread",
    "title": "Fish Stew"
}
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 95
Content-Type: application/json
Date: Tue, 13 Aug 2013 15:19:25 GMT
Location: http://localhost:5984/db/ab39fe0993049b84cfa81acd6ebad09d
Server: CouchDB (Erlang/OTP)

{
    "id": "ab39fe0993049b84cfa81acd6ebad09d",
    "ok": true,
    "rev": "1-9c65296036141e575d32ba9c034dd3ee"
}
```

Specifying the Document ID

The document ID can be specified by including the `_id` field in the JSON of the submitted record. The following request will create the same document with the ID `FishStew`.

Request:

```
POST /db HTTP/1.1
Accept: application/json
Content-Length: 98
Content-Type: application/json

{
```

(continues on next page)

(continued from previous page)

```
    "_id": "FishStew",
    "servings": 4,
    "subtitle": "Delicious with fresh bread",
    "title": "Fish Stew"
}
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 71
Content-Type: application/json
Date: Tue, 13 Aug 2013 15:19:25 GMT
ETag: "1-9c65296036141e575d32ba9c034dd3ee"
Location: http://localhost:5984/db/FishStew
Server: CouchDB (Erlang/OTP)

{
    "id": "FishStew",
    "ok": true,
    "rev": "1-9c65296036141e575d32ba9c034dd3ee"
}
```

Batch Mode Writes

You can write documents to the database at a higher rate by using the batch option. This collects document writes together in memory (on a per-user basis) before they are committed to disk. This increases the risk of the documents not being stored in the event of a failure, since the documents are not written to disk immediately.

Batch mode is not suitable for critical data, but may be ideal for applications such as log data, when the risk of some data loss due to a crash is acceptable.

To use batch mode, append the `batch=ok` query argument to the URL of a *POST /{db}*, *PUT /{db}/{docid}*, or *DELETE /{db}/{docid}* request. The CouchDB server will respond with an HTTP 202 Accepted response code immediately.

Note: Creating or updating documents with batch mode doesn't guarantee that all documents will be successfully stored on disk. For example, individual documents may not be saved due to conflicts, rejection by *validation function* or by other reasons, even if overall the batch was successfully submitted.

Request:

```
POST /db?batch=ok HTTP/1.1
Accept: application/json
Content-Length: 98
Content-Type: application/json

{
    "_id": "FishStew",
    "servings": 4,
    "subtitle": "Delicious with fresh bread",
    "title": "Fish Stew"
}
```

Response:

```
HTTP/1.1 202 Accepted
Cache-Control: must-revalidate
Content-Length: 28
Content-Type: application/json
Date: Tue, 13 Aug 2013 15:19:25 GMT
Location: http://localhost:5984/db/FishStew
Server: CouchDB (Erlang/OTP)

{
    "id": "FishStew",
    "ok": true
}
```

12.3.2 /{db}/_all_docs

GET /{db}/_all_docs

Executes the built-in *_all_docs view*, returning all of the documents in the database. With the exception of the URL parameters (described below), this endpoint works identically to any other view. Refer to the *view endpoint* documentation for a complete description of the available query parameters and the format of the returned data.

Parameters

- **db** – Database name

Request Headers

- Content-Type – *application/json*

Response Headers

- Content-Type –
  - *application/json*

Status Codes

- 200 OK – Request completed successfully
- 404 Not Found – Requested database not found

Request:

```
GET /db/_all_docs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 10 Aug 2013 16:22:56 GMT
ETag: "1W2DJUZFZSZD9K78UFA3GZWB4"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "offset": 0,
    "rows": [
        {
```

(continues on next page)

(continued from previous page)

```
            "id": "16e458537602f5ef2a710089dffd9453",
            "key": "16e458537602f5ef2a710089dffd9453",
            "value": {
                "rev": "1-967a00dff5e02add41819138abb3284d"
            }
        },
        {
            "id": "a4c51cdfa2069f3e905c431114001aff",
            "key": "a4c51cdfa2069f3e905c431114001aff",
            "value": {
                "rev": "1-967a00dff5e02add41819138abb3284d"
            }
        },
        {
            "id": "a4c51cdfa2069f3e905c4311140034aa",
            "key": "a4c51cdfa2069f3e905c4311140034aa",
            "value": {
                "rev": "5-6182c9c954200ab5e3c6bd5e76a1549f"
            }
        },
        {
            "id": "a4c51cdfa2069f3e905c431114003597",
            "key": "a4c51cdfa2069f3e905c431114003597",
            "value": {
                "rev": "2-7051cbe5c8faecd085a3fa619e6e6337"
            }
        },
        {
            "id": "f4ca7773ddea715afebc4b4b15d4f0b3",
            "key": "f4ca7773ddea715afebc4b4b15d4f0b3",
            "value": {
                "rev": "2-7051cbe5c8faecd085a3fa619e6e6337"
            }
        }
    ],
    "total_rows": 5
}
```

POST /{db}/_all_docs

POST *_all_docs* functionality supports identical parameters and behavior as specified in the *GET /{db}/_all_docs* API but allows for the query string parameters to be supplied as keys in a JSON object in the body of the *POST* request.

Request:

```
POST /db/_all_docs HTTP/1.1
Accept: application/json
Content-Length: 70
Content-Type: application/json
Host: localhost:5984

{
    "keys" : [
        "Zingylemontart",
        "Yogurtraita"
    ]
}
```

Response:

```
{
    "total_rows" : 2666,
    "rows" : [
        {
            "value" : {
                "rev" : "1-a3544d296de19e6f5b932ea77d886942"
            },
            "id" : "Zingylemontart",
            "key" : "Zingylemontart"
        },
        {
            "value" : {
                "rev" : "1-91635098bfe7d40197a1b98d7ee085fc"
            },
            "id" : "Yogurtraita",
            "key" : "Yogurtraita"
        }
    ],
    "offset" : 0
}
```

12.3.3 /{db}/_design_docs

New in version 2.2.

GET /{db}/_design_docs

Returns a JSON structure of all of the design documents in a given database. The information is returned as a JSON structure containing meta information about the return structure, including a list of all design documents and basic contents, consisting the ID, revision and key. The key is the design document's `_id`.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **conflicts** (*boolean*) – Includes *conflicts* information in response. Ignored if *include_docs* isn't `true`. Default is `false`.
- **descending** (*boolean*) – Return the design documents in descending by key order. Default is `false`.
- **endkey** (*string*) – Stop returning records when the specified key is reached. *Optional*.
- **end_key** (*string*) – Alias for *endkey* param.
- **endkey_docid** (*string*) – Stop returning records when the specified design document ID is reached. *Optional*.
- **end_key_doc_id** (*string*) – Alias for *endkey_docid* param.
- **include_docs** (*boolean*) – Include the full content of the design documents in the return. Default is `false`.

- **inclusive_end** (*boolean*) – Specifies whether the specified end key should be included in the result. Default is `true`.
- **key** (*string*) – Return only design documents that match the specified key. *Optional*.
- **keys** (*string*) – Return only design documents that match the specified keys. *Optional*.
- **limit** (*number*) – Limit the number of the returned design documents to the specified number. *Optional*.
- **skip** (*number*) – Skip this number of records before starting to return the results. Default is `0`.
- **startkey** (*string*) – Return records starting with the specified key. *Optional*.
- **start_key** (*string*) – Alias for *startkey* param.
- **startkey_docid** (*string*) – Return records starting with the specified design document ID. *Optional*.
- **start_key_doc_id** (*string*) – Alias for *startkey_docid* param.
- **update_seq** (*boolean*) – Response includes an `update_seq` value indicating which sequence id of the underlying database the view reflects. Default is `false`.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- ETag – Response signature

Response JSON Object

- **offset** (*number*) – Offset where the design document list started
- **rows** (*array*) – Array of view row objects. By default the information returned contains only the design document ID and revision.
- **total_rows** (*number*) – Number of design documents in the database. Note that this is not the number of rows returned in the actual query.
- **update_seq** (*number*) – Current update sequence for the database

Status Codes

- 200 OK – Request completed successfully
- 404 Not Found – Requested database not found

Request:

```
GET /db/_design_docs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 23 Dec 2017 16:22:56 GMT
ETag: "1W2DJUZFZSZD9K78UFA3GZWB4"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked
```

(continues on next page)

(continued from previous page)

```
{
    "offset": 0,
    "rows": [
        {
            "id": "_design/ddoc01",
            "key": "_design/ddoc01",
            "value": {
                "rev": "1-7407569d54af5bc94c266e70cbf8a180"
            }
        },
        {
            "id": "_design/ddoc02",
            "key": "_design/ddoc02",
            "value": {
                "rev": "1-d942f0ce01647aa0f46518b213b5628e"
            }
        },
        {
            "id": "_design/ddoc03",
            "key": "_design/ddoc03",
            "value": {
                "rev": "1-721fead6e6c8d811a225d5a62d08dfd0"
            }
        },
        {
            "id": "_design/ddoc04",
            "key": "_design/ddoc04",
            "value": {
                "rev": "1-32c76b46ca61351c75a84fbcbceece2f"
            }
        },
        {
            "id": "_design/ddoc05",
            "key": "_design/ddoc05",
            "value": {
                "rev": "1-af856babf9cf746b48ae999645f9541e"
            }
        }
    ],
    "total_rows": 5
}
```

POST /{db}/_design_docs

POST *_design_docs* functionality supports identical parameters and behavior as specified in the *GET /{db}/_design_docs* API but allows for the query string parameters to be supplied as keys in a JSON object in the body of the *POST* request.

Request:

```
POST /db/_design_docs HTTP/1.1
Accept: application/json
Content-Length: 70
Content-Type: application/json
Host: localhost:5984

{
    "keys" : [
```

(continues on next page)

(continued from previous page)

```
        "_design/ddoc02",
        "_design/ddoc05"
    ]
}
```

The returned JSON is the all documents structure, but with only the selected keys in the output:

```
{
    "total_rows" : 5,
    "rows" : [
        {
            "value" : {
                "rev" : "1-d942f0ce01647aa0f46518b213b5628e"
            },
            "id" : "_design/ddoc02",
            "key" : "_design/ddoc02"
        },
        {
            "value" : {
                "rev" : "1-af856babf9cf746b48ae999645f9541e"
            },
            "id" : "_design/ddoc05",
            "key" : "_design/ddoc05"
        }
    ],
    "offset" : 0
}
```

Sending multiple queries to a database

`/{db}/_all_docs/queries`

New in version 2.2.

POST /{db}/_all_docs/queries

Executes multiple specified built-in view queries of all documents in this database. This enables you to request multiple queries in a single request, in place of multiple *POST /{db}/_all_docs* requests.

Parameters

- **db** – Database name

Request Headers

- Content-Type –
  - *application/json*
- Accept –
  - *application/json*

Request JSON Object

- **queries** – An array of query objects with fields for the parameters of each individual view query to be executed. The field names and their meaning are the same as the query parameters of a regular *_all_docs request*.

Response Headers

- Content-Type –

- *application/json*
- *text/plain; charset=utf-8*

- ETag – Response signature
- Transfer-Encoding – chunked

Response JSON Object

- **results** (*array*) – An array of result objects - one for each query. Each result object contains the same fields as the response to a regular *_all_docs request*.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 404 Not Found – Specified database is missing
- 500 Internal Server Error – Query execution error

Request:

```
POST /db/_all_docs/queries HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: localhost:5984

{
    "queries": [
        {
            "keys": [
                "meatballs",
                "spaghetti"
            ]
        },
        {
            "limit": 3,
            "skip": 2
        }
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 20 Dec 2017 11:17:07 GMT
ETag: "1H8RGBCK3ABY6ACDM7ZSC30QK"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "results" : [
        {
            "rows": [
                {
                    "id": "meatballs",
                    "key": "meatballs",
```

(continues on next page)

(continued from previous page)

```
                    "value": 1
                },
                {
                    "id": "spaghetti",
                    "key": "spaghetti",
                    "value": 1
                }
            ],
            "total_rows": 3
        },
        {
            "offset" : 2,
            "rows" : [
                {
                    "id" : "Adukiandorangecasserole-microwave",
                    "key" : "Aduki and orange casserole - microwave",
                    "value" : [
                        null,
                        "Aduki and orange casserole - microwave"
                    ]
                },
                {
                    "id" : "Aioli-garlicmayonnaise",
                    "key" : "Aioli - garlic mayonnaise",
                    "value" : [
                        null,
                        "Aioli - garlic mayonnaise"
                    ]
                },
                {
                    "id" : "Alabamapeanutchicken",
                    "key" : "Alabama peanut chicken",
                    "value" : [
                        null,
                        "Alabama peanut chicken"
                    ]
                }
            ],
            "total_rows" : 2667
        }
    ]
}
```

Note: The multiple queries are also supported in */{db}/_local_docs/queries* and */{db}/_design_docs/queries* (similar to */{db}/_all_docs/queries*).

/{db}/_design_docs/queries

POST /{db}/_design_docs/queries

Querying with specified `keys` will return design documents only. You can also combine `keys` with other query parameters, such as `limit` and `skip`.

Parameters

- **db** – Database name

Request Headers

- Content-Type –
  - *application/json*
- Accept –
  - *application/json*

Request JSON Object

- **queries** – An array of query objects with fields for the parameters of each individual view query to be executed. The field names and their meaning are the same as the query parameters of a regular *_design_docs request*.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- Transfer-Encoding – `chunked`

Response JSON Object

- **results** (*array*) – An array of result objects - one for each query. Each result object contains the same fields as the response to a regular *_design_docs request*.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 404 Not Found – Specified database is missing
- 500 Internal Server Error – Query execution error

Request:

```
POST /db/_design_docs/queries HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: localhost:5984

{
    "queries": [
        {
            "keys": [
                "_design/recipe",
                "_design/not-exist",
                "spaghetti"
            ]
```

(continues on next page)

(continued from previous page)

```
        }
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Thu, 20 Jul 2023 20:06:44 GMT
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "results": [
        {
            "total_rows": 1,
            "offset": null,
            "rows": [
                {
                    "id": "_design/recipe",
                    "key": "_design/recipe",
                    "value": {
                        "rev": "1-ad0e29fe6b473658514742a7c2317766"
                    }
                },
                {
                    "key": "_design/not-exist",
                    "error": "not_found"
                }
            ]
        }
    ]
}
```

Note: /{db}/_design_docs/queries with keys will only return design documents, or `"error": "not_found"` if the design document doesn't exist. If `key` is not a design document id, it will not be included in the response.

12.3.4 /{db}/_bulk_get

POST /{db}/_bulk_get

This method can be called to query several documents in bulk. It is well suited for fetching a specific revision of documents, as replicators do for example, or for getting revision history.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *multipart/related*
  - *multipart/mixed*

- Content-Type – *application/json*

Query Parameters

- **revs** (*boolean*) – Give the revisions history

Request JSON Object

- **docs** (*array*) – List of document objects, with `id`, and optionally `rev` and `atts_since`

Response Headers

- Content-Type –
  - *application/json*

Response JSON Object

- **results** (*object*) – an array of results for each requested document/rev pair. `id` key lists the requested document ID, `docs` contains a single-item array of objects, each of which has either an `error` key and value describing the error, or `ok` key and associated value of the requested document, with the additional `_revisions` property that lists the parent revisions if `revs=true`.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – The request provided invalid JSON data or invalid query parameter
- 401 Unauthorized – Read permission required
- 404 Not Found – Invalid database name
- 415 Unsupported Media Type – Bad Content-Type value

Request:

```
POST /db/_bulk_get HTTP/1.1
Accept: application/json
Content-Type:application/json
Host: localhost:5984

{
    "docs": [
        {
            "id": "foo"
            "rev": "4-753875d51501a6b1883a9d62b4d33f91",
        },
        {
            "id": "foo"
            "rev": "1-4a7e4ae49c4366eaed8edeaea8f784ad",
        },
        {
            "id": "bar",
        }
        {
            "id": "baz",
        }
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Mon, 19 Mar 2018 15:27:34 GMT
Server: CouchDB (Erlang/OTP)

{
  "results": [
    {
      "id": "foo",
      "docs": [
        {
          "ok": {
            "_id": "foo",
            "_rev": "4-753875d51501a6b1883a9d62b4d33f91",
            "value": "this is foo",
            "_revisions": {
              "start": 4,
              "ids": [
                "753875d51501a6b1883a9d62b4d33f91",
                "efc54218773c6acd910e2e97fea2a608",
                "2ee767305024673cfb3f5af037cd2729",
                "4a7e4ae49c4366eaed8edeaea8f784ad"
              ]
            }
          }
        }
      ]
    },
    {
      "id": "foo",
      "docs": [
        {
          "ok": {
            "_id": "foo",
            "_rev": "1-4a7e4ae49c4366eaed8edeaea8f784ad",
            "value": "this is the first revision of foo",
            "_revisions": {
              "start": 1,
              "ids": [
                "4a7e4ae49c4366eaed8edeaea8f784ad"
              ]
            }
          }
        }
      ]
    },
    {
      "id": "bar",
      "docs": [
        {
          "ok": {
            "_id": "bar",
            "_rev": "2-9b71d36dfdd9b4815388eb91cc8fb61d",
            "baz": true,
            "_revisions": {
              "start": 2,
```

(continues on next page)

(continued from previous page)

```
              "ids": [
                "9b71d36dfdd9b4815388eb91cc8fb61d",
                "309651b95df56d52658650fb64257b97"
              ]
            }
          }
        }
      ]
    },
    {
      "id": "baz",
      "docs": [
        {
          "error": {
            "id": "baz",
            "rev": "undefined",
            "error": "not_found",
            "reason": "missing"
          }
        }
      ]
    }
  ]
}
```

Example response with a conflicted document:

Request:

```
POST /db/_bulk_get HTTP/1.1
Accept: application/json
Content-Type:application/json
Host: localhost:5984

{
    "docs": [
        {
            "id": "a"
        }
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Mon, 19 Mar 2018 15:27:34 GMT
Server: CouchDB (Erlang/OTP)

{
  "results": [
    {
      "id": "a",
      "docs": [
        {
          "ok": {
```

(continues on next page)

(continued from previous page)

```
            "_id": "a",
            "_rev": "1-23202479633c2b380f79507a776743d5",
            "a": 1
          }
        },
        {
          "ok": {
            "_id": "a",
            "_rev": "1-967a00dff5e02add41819138abb3284d"
          }
        }
      ]
    }
  ]
}
```

12.3.5 /{db}/_bulk_docs

POST /{db}/_bulk_docs

The bulk document API allows you to create and update multiple documents at the same time within a single request. The basic operation is similar to creating or updating a single document, except that you batch the document structure and information.

When creating new documents the document ID (`_id`) is optional.

For updating existing documents, you must provide the document ID, revision information (`_rev`), and new document values.

In case of batch deleting documents all fields as document ID, revision information and deletion status (`_deleted`) are required.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Request JSON Object

- **docs** (*array*) – List of documents objects
- **new_edits** (*boolean*) – If `false`, prevents the database from assigning them new revision IDs. Default is `true`. *Optional*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Array of Objects

- **id** (*string*) – Document ID

- **rev** (*string*) – New document revision token. Available if document has saved without errors. *Optional*
- **error** (*string*) – Error type. *Optional*
- **reason** (*string*) – Error reason. *Optional*

Status Codes

- 201 Created – Document(s) have been created or updated
- 400 Bad Request – The request provided invalid JSON data
- 404 Not Found – Requested database not found

Request:

```
POST /db/_bulk_docs HTTP/1.1
Accept: application/json
Content-Length: 109
Content-Type:application/json
Host: localhost:5984

{
    "docs": [
        {
            "_id": "FishStew"
        },
        {
            "_id": "LambStew",
            "_rev": "2-0786321986194c92dd3b57dfbfc741ce",
            "_deleted": true
        }
    ]
}
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 144
Content-Type: application/json
Date: Mon, 12 Aug 2013 00:15:05 GMT
Server: CouchDB (Erlang/OTP)

[
    {
        "ok": true,
        "id": "FishStew",
        "rev":" 1-967a00dff5e02add41819138abb3284d"
    },
    {
        "ok": true,
        "id": "LambStew",
        "rev": "3-f9c62b2169d0999103e9f41949090807"
    }
]
```

Inserting Documents in Bulk

Each time a document is stored or updated in CouchDB, the internal B-tree is updated. Bulk insertion provides efficiency gains in both storage space, and time, by consolidating many of the updates to intermediate B-tree nodes.

It is not intended as a way to perform `ACID`-like transactions in CouchDB, the only transaction boundary within CouchDB is a single update to a single database. The constraints are detailed in *Bulk Documents Transaction Semantics*.

To insert documents in bulk into a database you need to supply a JSON structure with the array of documents that you want to add to the database. You can either include a document ID, or allow the document ID to be automatically generated.

For example, the following update inserts three new documents, two with the supplied document IDs, and one which will have a document ID generated:

```
POST /source/_bulk_docs HTTP/1.1
Accept: application/json
Content-Length: 323
Content-Type: application/json
Host: localhost:5984

{
    "docs": [
        {
            "_id": "FishStew",
            "servings": 4,
            "subtitle": "Delicious with freshly baked bread",
            "title": "FishStew"
        },
        {
            "_id": "LambStew",
            "servings": 6,
            "subtitle": "Serve with a whole meal scone topping",
            "title": "LambStew"
        },
        {
            "servings": 8,
            "subtitle": "Hand-made dumplings make a great accompaniment",
            "title": "BeefStew"
        }
    ]
}
```

The return type from a bulk insertion will be 201 Created, with the content of the returned structure indicating specific success or otherwise messages on a per-document basis.

The return structure from the example above contains a list of the documents created, here with the combination and their revision IDs:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 215
Content-Type: application/json
Date: Sat, 26 Oct 2013 00:10:39 GMT
Server: CouchDB (Erlang OTP)

[
    {
        "id": "FishStew",
```

(continues on next page)

(continued from previous page)

```
        "ok": true,
        "rev": "1-6a466d5dfda05e613ba97bd737829d67"
    },
    {
        "id": "LambStew",
        "ok": true,
        "rev": "1-648f1b989d52b8e43f05aa877092cc7c"
    },
    {
        "id": "00a271787f89c0ef2e10e88a0c0003f0",
        "ok": true,
        "rev": "1-e4602845fc4c99674f50b1d5a804fdfa"
    }
]
```

For details of the semantic content and structure of the returned JSON see *Bulk Documents Transaction Semantics*. Conflicts and validation errors when updating documents in bulk must be handled separately; see *Bulk Document Validation and Conflict Errors*.

Updating Documents in Bulk

The bulk document update procedure is similar to the insertion procedure, except that you must specify the document ID and current revision for every document in the bulk update JSON string.

For example, you could send the following request:

```
POST /recipes/_bulk_docs HTTP/1.1
Accept: application/json
Content-Length: 464
Content-Type: application/json
Host: localhost:5984

{
    "docs": [
        {
            "_id": "FishStew",
            "_rev": "1-6a466d5dfda05e613ba97bd737829d67",
            "servings": 4,
            "subtitle": "Delicious with freshly baked bread",
            "title": "FishStew"
        },
        {
            "_id": "LambStew",
            "_rev": "1-648f1b989d52b8e43f05aa877092cc7c",
            "servings": 6,
            "subtitle": "Serve with a whole meal scone topping",
            "title": "LambStew"
        },
        {
            "_id": "BeefStew",
            "_rev": "1-e4602845fc4c99674f50b1d5a804fdfa",
            "servings": 8,
            "subtitle": "Hand-made dumplings make a great accompaniment",
            "title": "BeefStew"
        }
    ]
```

(continues on next page)

(continued from previous page)

```
}
```

The return structure is the JSON of the updated documents, with the new revision and ID information:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 215
Content-Type: application/json
Date: Sat, 26 Oct 2013 00:10:39 GMT
Server: CouchDB (Erlang OTP)

[
    {
        "id": "FishStew",
        "ok": true,
        "rev": "2-2bff94179917f1dec7cd7f0209066fb8"
    },
    {
        "id": "LambStew",
        "ok": true,
        "rev": "2-6a7aae7ac481aa98a2042718d09843c4"
    },
    {
        "id": "BeefStew",
        "ok": true,
        "rev": "2-9801936a42f06a16f16c30027980d96f"
    }
]
```

You can optionally delete documents during a bulk update by adding the `_deleted` field with a value of `true` to each document ID/revision combination within the submitted JSON structure.

The return type from a bulk insertion will be 201 Created, with the content of the returned structure indicating specific success or otherwise messages on a per-document basis.

The content and structure of the returned JSON will depend on the transaction semantics being used for the bulk update; see *Bulk Documents Transaction Semantics* for more information. Conflicts and validation errors when updating documents in bulk must be handled separately; see *Bulk Document Validation and Conflict Errors*.

Bulk Documents Transaction Semantics

Bulk document operations are **non-atomic**. This means that CouchDB does not guarantee that any individual document included in the bulk update (or insert) will be saved when you send the request. The response will contain the list of documents successfully inserted or updated during the process. In the event of a crash, some of the documents may have been successfully saved, while others lost.

The response structure will indicate whether the document was updated by supplying the new `_rev` parameter indicating a new document revision was created. If the update failed, you will get an `error` of type `conflict`. For example:

```
[
    {
        "id" : "FishStew",
        "error" : "conflict",
        "reason" : "Document update conflict."
    },
    {
```

(continues on next page)

(continued from previous page)

```
        "id" : "LambStew",
        "error" : "conflict",
        "reason" : "Document update conflict."
    },
    {
        "id" : "BeefStew",
        "error" : "conflict",
        "reason" : "Document update conflict."
    }
]
```

In this case no new revision has been created and you will need to submit the document update, with the correct revision tag, to update the document.

Replication of documents is independent of the type of insert or update. The documents and revisions created during a bulk insert or update are replicated in the same way as any other document.

Bulk Document Validation and Conflict Errors

The JSON returned by the `_bulk_docs` operation consists of an array of JSON structures, one for each document in the original submission. The returned JSON structure should be examined to ensure that all of the documents submitted in the original request were successfully added to the database.

When a document (or document revision) is not correctly committed to the database because of an error, you should check the `error` field to determine error type and course of action. Errors will be one of the following type:

- **conflict**

  The document as submitted is in conflict. The new revision will not have been created and you will need to re-submit the document to the database.

  Conflict resolution of documents added using the bulk docs interface is identical to the resolution procedures used when resolving conflict errors during replication.

- **forbidden**

  Entries with this error type indicate that the validation routine applied to the document during submission has returned an error.

  For example, if your *validation routine* includes the following:

```
throw({forbidden: 'invalid recipe ingredient'});
```

  The error response returned will be:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 80
Content-Type: application/json
Date: Sat, 26 Oct 2013 00:05:17 GMT
Server: CouchDB (Erlang OTP)

[
    {
        "id": "LambStew",
        "error": "forbidden",
        "reason": "invalid recipe ingredient"
    }
]
```

12.3.6 /{db}/_find

POST /{db}/_find

Find documents using a declarative JSON querying syntax. Queries will use custom indexes, specified using the *_index* endpoint, if available. Otherwise, they use the built-in *_all_docs* index, which can be arbitrarily slow.

Parameters

- **db** – Database name

Request Headers

- Content-Type –
  - *application/json*

Request JSON Object

- **selector** (*object*) – JSON object describing criteria used to select documents. More information provided in the section on *selector syntax*. *Required*
- **limit** (*number*) – Maximum number of results returned. Default is 25. *Optional*
- **skip** (*number*) – Skip the first 'n' results, where 'n' is the value specified. *Optional*
- **sort** (*array*) – JSON array following *sort syntax*. *Optional*
- **fields** (*array*) – JSON array specifying which fields of each object should be returned. If it is omitted, the entire object is returned. More information provided in the section on *filtering fields*. *Optional*
- **use_index** (*string|array*) – Request a query to use a specific index. Specified either as "<design_document>" or ["<design_document>", "<index_name>"]. It is not guaranteed that the index will be actually used because if the index is not valid for the selector, fallback to a valid index is attempted. Therefore that is more like a hint. When fallback occurs, the details are given in the warning field of the response. *Optional*
- **conflicts** (*boolean*) – Include conflicted documents if true. Intended use is to easily find conflicted documents, without an index or view. Default is false. *Optional*
- **r** (*number*) – Read quorum needed for the result. This defaults to 1, in which case the document found in the index is returned. If set to a higher value, each document is read from at least that many replicas before it is returned in the results. This is likely to take more time than using only the document stored locally with the index. *Optional, default: 1*
- **bookmark** (*string*) – A string that enables you to specify which page of results you require. Used for paging through result sets. Every query returns an opaque string under the bookmark key that can then be passed back in a query to get the next page of results. If any part of the selector query changes between requests, the results are undefined. *Optional, default: null*
- **update** (*boolean*) – Whether to update the index prior to returning the result. Default is true. *Optional*
- **stable** (*boolean*) – Whether or not the view results should be returned from a "stable" set of shards. *Optional*
- **stale** (*string*) – Combination of update=false and stable=true options. Possible options: "ok", false (default). *Optional* Note that this parameter is deprecated. Use stable and update instead. See *Views Generation* for more details.
- **execution_stats** (*boolean*) – Include *execution statistics* in the query response. *Optional, default:* false

Response Headers

- Content-Type –
  - *application/json*
- Transfer-Encoding – chunked

Response JSON Object

- **docs** (*object*) – Array of documents matching the search. In each matching document, the fields specified in the `fields` part of the request body are listed, along with their values.
- **warning** (*string*) – Execution warnings
- **execution_stats** (*object*) – Execution statistics
- **bookmark** (*string*) – An opaque string used for paging. See the `bookmark` field in the request (above) for usage details.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 404 Not Found – Requested database not found
- 500 Internal Server Error – Query execution error

The `limit` and `skip` values are exactly as you would expect. While `skip` exists, it is not intended to be used for paging. The reason is that the `bookmark` feature is more efficient.

Request:

Example request body for finding documents using an index:

```
POST /movies/_find HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 168
Host: localhost:5984

{
    "selector": {
        "year": {"$gt": 2010}
    },
    "fields": ["_id", "_rev", "year", "title"],
    "sort": [{"year": "asc"}],
    "limit": 2,
    "skip": 0,
    "execution_stats": true
}
```

Response:

Example response when finding documents using an index:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Thu, 01 Sep 2016 15:41:53 GMT
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked
```

(continues on next page)

(continued from previous page)

```
{
    "docs": [
        {
            "_id": "176694",
            "_rev": "1-54f8e950cc338d2385d9b0cda2fd918e",
            "year": 2011,
            "title": "The Tragedy of Man"
        },
        {
            "_id": "780504",
            "_rev": "1-5f14bab1a1e9ac3ebdf85905f47fb084",
            "year": 2011,
            "title": "Drive"
        }
    ],
    "execution_stats": {
        "total_keys_examined": 200,
        "total_docs_examined": 200,
        "total_quorum_docs_examined": 0,
        "results_returned": 2,
        "execution_time_ms": 5.52
    }
}
```

Selector Syntax

Selectors are expressed as a JSON object describing documents of interest. Within this structure, you can apply conditional logic using specially named fields.

Whilst selectors have some similarities with MongoDB query documents, these arise from a similarity of purpose and do not necessarily extend to commonality of function or result.

Selector Basics

Elementary selector syntax requires you to specify one or more fields, and the corresponding values required for those fields. This selector matches all documents whose `"director"` field has the value `"Lars von Trier"`.

```
{
    "director": "Lars von Trier"
}
```

A simple selector, inspecting specific fields:

```
"selector": {
    "title": "Live And Let Die"
},
"fields": [
    "title",
    "cast"
]
```

You can create more complex selector expressions by combining operators. For best performance, it is best to combine 'combination' or 'array logical' operators, such as `$regex`, with an operator that defines a contiguous range of keys such as `$eq`, `$gt`, `$gte`, `$lt`, `$lte`, and `$beginsWith` (but not `$ne`). For more information about creating complex selector expressions, see *creating selector expressions*.

Selector with 2 fields

This selector matches any document with a name field containing "Paul", and that also has a location field with the value "Boston".

```
{
    "name": "Paul",
    "location": "Boston"
}
```

Subfields

A more complex selector enables you to specify the values for field of nested objects, or subfields. For example, you might use a standard JSON structure for specifying a field and subfield.

Example of a field and subfield selector, using a standard JSON structure:

```
{
    "imdb": {
        "rating": 8
    }
}
```

An abbreviated equivalent uses a dot notation to combine the field and subfield names into a single name.

```
{
    "imdb.rating": 8
}
```

Operators

Operators are identified by the use of a dollar sign ($) prefix in the name field.

There are two core types of operators in the selector syntax:

- Combination operators
- Condition operators

In general, combination operators are applied at the topmost level of selection. They are used to combine conditions, or to create combinations of conditions, into one selector.

Every explicit operator has the form:

```
{
    "$operator": argument
}
```

A selector without an explicit operator is considered to have an implicit operator. The exact implicit operator is determined by the structure of the selector expression.

Implicit Operators

There are two implicit operators:

- Equality
- And

In a selector, any field containing a JSON value, but that has no operators in it, is considered to be an equality condition. The implicit equality test applies also for fields and subfields.

Any JSON object that is not the argument to a condition operator is an implicit `$and` operator on each field.

In the below example, we use an operator to match any document, where the `"year"` field has a value greater than `2010`:

```
{
    "year": {
        "$gt": 2010
    }
}
```

In this next example, there must be a field `"director"` in a matching document, and the field must have a value exactly equal to `"Lars von Trier"`.

```
{
    "director": "Lars von Trier"
}
```

You can also make the equality operator explicit.

```
{
    "director": {
        "$eq": "Lars von Trier"
    }
}
```

In the next example using subfields, the required field `"imdb"` in a matching document must also have a subfield `"rating"` and the subfield must have a value equal to `8`.

Example of implicit operator applied to a subfield test:

```
{
    "imdb": {
        "rating": 8
    }
}
```

Again, you can make the equality operator explicit.

```
{
    "imdb": {
        "rating": { "$eq": 8 }
    }
}
```

An example of the `$eq` operator used with full text indexing:

```
{
  "selector": {
    "year": {
```

(continues on next page)

(continued from previous page)

```
      "$eq": 2001
    }
  },
  "sort": [
    "title:string"
  ],
  "fields": [
    "title"
  ]
}
```

An example of the `$eq` operator used with database indexed on the field `"year"`:

```
{
  "selector": {
    "year": {
      "$eq": 2001
    }
  },
  "sort": [
    "year"
  ],
  "fields": [
    "year"
  ]
}
```

In this example, the field `"director"` must be present and contain the value `"Lars von Trier"` and the field `"year"` must exist and have the value `2003`.

```
{
    "director": "Lars von Trier",
    "year": 2003
}
```

You can make both the `$and` operator and the equality operator explicit.

Example of using explicit `$and` and `$eq` operators:

```
{
    "$and": [
        {
            "director": {
                "$eq": "Lars von Trier"
            }
        },
        {
            "year": {
                "$eq": 2003
            }
        }
    ]
}
```

Explicit Operators

All operators, apart from 'Equality' and 'And', must be stated explicitly.

Combination Operators

Combination operators are used to combine selectors. In addition to the common boolean operators found in most programming languages, there are three combination operators (`$all`, `$elemMatch`, and `$allMatch`) that help you work with JSON arrays and one that works with JSON maps (`$keyMapMatch`).

A combination operator takes a single argument. The argument is either another selector, or an array of selectors.

The list of combination operators:

Operator	Argument	Purpose
`$and`	Array	Matches if all the selectors in the array match.
`$or`	Array	Matches if any of the selectors in the array match. All selectors must use the same index.
`$not`	Selector	Matches if the given selector does not match.
`$nor`	Array	Matches if none of the selectors in the array match.
`$all`	Array	Matches an array value if it contains all the elements of the argument array.
`$elemMatc`	Selector	Matches and returns all documents that contain an array field with at least one element that matches all the specified query criteria.
`$allMatch`	Selector	Matches and returns all documents that contain an array field with all its elements matching all the specified query criteria.
`$keyMapMa`	Selector	Matches and returns all documents that contain a map that contains at least one key that matches all the specified query criteria.

The `$and` operator

`$and` operator used with two fields:

```
{
  "selector": {
    "$and": [
      {
        "title": "Total Recall"
      },
      {
        "year": {
          "$in": [1984, 1991]
        }
      }
    ]
  },
  "fields": [
      "year",
      "title",
      "cast"
  ]
}
```

The `$and` operator matches if all the selectors in the array match. Below is an example using the primary index (`_all_docs`):

```
{
    "$and": [
        {
            "_id": { "$gt": null }
        },
        {
            "year": {
                "$in": [2014, 2015]
            }
        }
    ]
}
```

The $or operator

The $or operator matches if any of the selectors in the array match. Below is an example used with an index on the field "year":

```
{
    "year": 1977,
    "$or": [
        { "director": "George Lucas" },
        { "director": "Steven Spielberg" }
    ]
}
```

The $not operator

The $not operator matches if the given selector does not match. Below is an example used with an index on the field "year":

```
{
    "year": {
        "$gte": 1900
    },
    "year": {
        "$lte": 1903
    },
    "$not": {
        "year": 1901
    }
}
```

The $nor operator

The $nor operator matches if the given selector does not match. Below is an example used with an index on the field "year":

```
{
    "year": {
        "$gte": 1900
    },
    "year": {
        "$lte": 1910
    },
    "$nor": [
        { "year": 1901 },
        { "year": 1905 },
        {  "year": 1907 }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

The `$all` operator

The `$all` operator matches an array value if it contains all the elements of the argument array. Below is an example used with the primary index (`_all_docs`):

```
{
    "_id": {
        "$gt": null
    },
    "genre": {
        "$all": ["Comedy","Short"]
    }
}
```

The `$elemMatch` operator

The `$elemMatch` operator matches and returns all documents that contain an array field with at least one element matching the supplied query criteria. Below is an example used with the primary index (`_all_docs`):

```
{
    "_id": { "$gt": null },
    "genre": {
        "$elemMatch": {
            "$eq": "Horror"
        }
    }
}
```

The `$allMatch` operator

The `$allMatch` operator matches and returns all documents that contain an array field with all its elements matching the supplied query criteria. Below is an example used with the primary index (`_all_docs`):

```
{
    "_id": { "$gt": null },
    "genre": {
        "$allMatch": {
            "$eq": "Horror"
        }
    }
}
```

The `$keyMapMatch` operator

The `$keyMapMatch` operator matches and returns all documents that contain a map that contains at least one key that matches all the specified query criteria. Below is an example used with the primary index (`_all_docs`):

```
{
    "_id": { "$gt": null },
    "cameras": {
        "$keyMapMatch": {
            "$eq": "secondary"
        }
    }
}
```

Condition Operators

Condition operators are specific to a field, and are used to evaluate the value stored in that field. For instance, the basic `$eq` operator matches when the specified field contains a value that is equal to the supplied argument.

Note: For a condition operator to function correctly, the field **must exist** in the document for the selector to match. As an example, `$ne` means the specified field must exist, and is not equal to the value of the argument.

The basic equality and inequality operators common to most programming languages are supported. Strict type matching is used.

In addition, some 'meta' condition operators are available. Some condition operators accept any valid JSON content as the argument. Other condition operators require the argument to be in a specific JSON format.

Operator type	Operator	Argument	Purpose
(In)ec	`$lt`	Any JSON	The field is less than the argument.
	`$lte`	Any JSON	The field is less than or equal to the argument.
	`$eq`	Any JSON	The field is equal to the argument.
	`$ne`	Any JSON	The field is not equal to the argument.
	`$gte`	Any JSON	The field is greater than or equal to the argument.
	`$gt`	Any JSON	The field is greater than the argument.
Object	`$exi`	Boolea	Check whether the field exists or not, regardless of its value.
	`$typ`	String	Check the document field's type. Valid values are `"null"`, `"boolean"`, `"number"`, `"string"`, `"array"`, and `"object"`.
Array	`$in`	Array of JSON values	The document field must exist in the list provided.
	`$nin`	Array of JSON values	The document field not must exist in the list provided.
	`$siz`	Integer	Special condition to match the length of an array field in a document. Non-array fields cannot match this condition.
Miscellaneous	`$mod`	[Divisor, Remainder]	Divisor is a non-zero integer, Remainder is any integer. Non-integer values result in a 404. Matches documents where `field % Divisor == Remainder` is true, and only when the document field is an integer.
	`$reg`	String	A regular expression pattern to match against the document field. Only matches when the field is a string value and matches the supplied regular expression. The matching algorithms are based on the Perl Compatible Regular Expression (PCRE) library. For more information about what is implemented, see the Erlang Regular Expression.
	`$beg`	String	Matches where the document field begins with the specified prefix (case-sensitive). If the document field contains a non-string value, the document is not matched.

Warning: Regular expressions do not work with indexes, so they should not be used to filter large data sets. They can, however, be used to restrict a *partial index*.

Creating Selector Expressions

We have seen examples of combining selector expressions, such as *using explicit $and and $eq operators*.

In general, whenever you have an operator that takes an argument, that argument can itself be another operator with arguments of its own. This enables us to build up more complex selector expressions.

However, only operators that define a contiguous range of values such as `$eq`, `$gt`, `$gte`, `$lt`, `$lte`, and `$beginsWith` (but not `$ne`) can be used as the basis of a query that can make efficient use of a `json` index. You should include at least one of these in a selector, or consider using a `text` index if greater flexibility is required.

For example, if you try to perform a query that attempts to match all documents that have a field called *afieldname* containing a value that begins with the letter *A*, this will trigger a warning because no index could be used and the database performs a full scan of the primary index:

Request

```
POST /movies/_find HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 112
Host: localhost:5984

{
    "selector": {
        "afieldname": {"$regex": "^A"}
    }
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Thu, 01 Sep 2016 17:25:51 GMT
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked

{
    "warning":"no matching index found, create an index to optimize query␣
↪time",
    "docs":[
    ]
}
```

Warning: It is always recommended that you create an appropriate index when deploying in production.

Most selector expressions work exactly as you would expect for the given operator. But it is not always the case: for example, comparison of strings is done with ICU and can can give surprising results if you were expecting ASCII ordering. See *Views Collation* for more details.

Sort Syntax

The `sort` field contains a list of field name and direction pairs, expressed as a basic array. The first field name and direction pair is the topmost level of sort. The second pair, if provided, is the next level of sort.

The field can be any field, using dotted notation if desired for sub-document fields.

The direction value is `"asc"` for ascending, and `"desc"` for descending. If you omit the direction value, the default `"asc"` is used.

Example, sorting by 2 fields:

```
[{"fieldName1": "desc"}, {"fieldName2": "desc"}]
```

Example, sorting by 2 fields, assuming default direction for both :

```
["fieldNameA", "fieldNameB"]
```

A typical requirement is to search for some content using a selector, then to sort the results according to the specified field, in the required direction.

To use sorting, ensure that:

- At least one of the sort fields is included in the selector.
- There is an index already defined, with all the sort fields in the same order.
- Each object in the sort array has a single key.

If an object in the sort array does not have a single key, the resulting sort order is implementation specific and might change.

Find does not support multiple fields with different sort orders, so the directions must be either all ascending or all descending.

For field names in text search sorts, it is sometimes necessary for a field type to be specified, for example:

```
{
    "<fieldname>:string": "asc"
}
```

If possible, an attempt is made to discover the field type based on the selector. In ambiguous cases the field type must be provided explicitly.

The sorting order is undefined when fields contain different data types. This is an important difference between text and view indexes. Sorting behavior for fields with different data types might change in future versions.

A simple query, using sorting:

```
{
    "selector": {"Actor_name": "Robert De Niro"},
    "sort": [{"Actor_name": "asc"}, {"Movie_runtime": "asc"}]
}
```

Filtering Fields

It is possible to specify exactly which fields are returned for a document when selecting from a database. The two advantages are:

- Your results are limited to only those parts of the document that are required for your application.
- A reduction in the size of the response.

The fields returned are specified as an array.

Only the specified filter fields are included, in the response. There is no automatic inclusion of the `_id` or other metadata fields when a field list is included.

Example of selective retrieval of fields from matching documents:

```
{
    "selector": { "Actor_name": "Robert De Niro" },
    "fields": ["Actor_name", "Movie_year", "_id", "_rev"]
}
```

Pagination

Mango queries support pagination via the bookmark field. Every `_find` response contains a bookmark - a token that CouchDB uses to determine where to resume from when subsequent queries are made. To get the next set of query results, add the bookmark that was received in the previous response to your next request. Remember to keep the *selector* the same, otherwise you will receive unexpected results. To paginate backwards, you can use a previous bookmark to return the previous set of results.

Note that the presence of a bookmark does not guarantee that there are more results. You can to test whether you have reached the end of the result set by comparing the number of results returned with the page size requested - if results returned < *limit*, there are no more.

Execution Statistics

Find can return basic execution statistics for a specific request. Combined with the *_explain* endpoint, this should provide some insight as to whether indexes are being used effectively.

The execution statistics currently include:

Field	Description
`total_keys_ex`	Number of index keys examined.
`total_docs_ex`	Number of documents fetched from the database / index, equivalent to using `include_docs=true` in a view. These may then be filtered in-memory to further narrow down the result set based on the selector.
`total_quorum_`	Number of documents fetched from the database using an out-of-band document fetch. This is only non-zero when read quorum > 1 is specified in the query parameters.
`results_retur`	Number of results returned from the query. Ideally this should not be significantly lower than the total documents / keys examined.
`execution_tim`	Total execution time in milliseconds as measured by the database.

12.3.7 /{db}/_index

Mango is a declarative JSON querying language for CouchDB databases. Mango wraps several index types, starting with the Primary Index out-of-the-box. Mango indexes, with index type *json*, are built using MapReduce Views.

POST /{db}/_index

Create a new index on a database

Parameters

- **db** – Database name

Request Headers

- Content-Type –
  - *application/json*

Query Parameters

- **index** (*object*) – JSON object describing the index to create.
- **ddoc** (*string*) – Name of the design document in which the index will be created. By default, each index will be created in its own design document. Indexes can be grouped into design documents for efficiency. However, a change to one index in a design document will invalidate all other indexes in the same document (similar to views). *Optional*
- **name** (*string*) – Name of the index. If no name is provided, a name will be generated automatically. *Optional*
- **type** (*string*) – Can be `"json"` or `"text"`. Defaults to `"json"`. Geospatial indexes will be supported in the future. *Optional* Text indexes are supported via a third-party library. *Optional*
- **partitioned** (*boolean*) – Determines whether a JSON index is partitioned or global. The default value of `partitioned` is the `partitioned` property of the database. To create a global index on a partitioned database, specify `false` for the `"partitioned"` field. If you specify `true` for the `"partitioned"` field on an unpartitioned database, an error occurs.

Response Headers

- Content-Type –
  - *application/json*
- Transfer-Encoding – `chunked`

Response JSON Object

- **result** (*string*) – Flag to show whether the index was created or one already exists. Can be `"created"` or `"exists"`.
- **id** (*string*) – Id of the design document the index was created in.
- **name** (*string*) – Name of the index created.

Status Codes

- 200 OK – Index created successfully or already exists
- 400 Bad Request – Invalid request
- 401 Unauthorized – Admin permission required
- 404 Not Found – Database not found
- 500 Internal Server Error – Execution error

The *index* parameter is a JSON object with the following fields:

- **fields** (*array*): Array of field names following the *sort syntax*. Nested fields are also allowed, e.g. *"person.name"*.
- **partial_filter_selector** (*object*): A *selector* to apply to documents at indexing time, creating a *partial index*. *Optional*

Example of creating a new index for a field called `foo`:

Request:

```
POST /db/_index HTTP/1.1
Content-Type: application/json
Content-Length: 116
Host: localhost:5984

{
    "index": {
        "fields": ["foo"]
    },
    "name" : "foo-index",
    "type" : "json"
}
```

The returned JSON confirms the index has been created:

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 96
Content-Type: application/json
Date: Thu, 01 Sep 2016 18:17:48 GMT
Server: CouchDB (Erlang OTP/18)

{
    "result":"created",
    "id":"_design/a5f4711fc9448864a13c81dc71e660b524d7410c",
    "name":"foo-index"
}
```

Example index creation using all available query parameters:

Request:

```
POST /db/_index HTTP/1.1
Content-Type: application/json
Content-Length: 396
Host: localhost:5984

{
    "index": {
        "partial_filter_selector": {
            "year": {
                "$gt": 2010
            },
            "limit": 10,
            "skip": 0
        },
        "fields": [
            "_id",
```

(continues on next page)

(continued from previous page)

```
            "_rev",
            "year",
            "title"
        ]
    },
    "ddoc": "example-ddoc",
    "name": "example-index",
    "type": "json",
    "partitioned": false
}
```

By default, a JSON index will include all documents that have the indexed fields present, including those which have `null` values.

Partial Indexes

Partial indexes allow documents to be filtered at indexing time, potentially offering significant performance improvements for query selectors that do not map cleanly to a range query on an index.

Let's look at an example query:

```
{
    "selector": {
        "status": {
            "$ne": "archived"
        },
        "type": "user"
    }
}
```

Without a partial index, this requires a full index scan to find all the documents of `"type":"user"` that do not have a status of `"archived"`. This is because a normal index can only be used to match contiguous rows, and the `"$ne"` operator cannot guarantee that.

To improve response times, we can create an index which excludes documents where `"status": { "$ne": "archived" }` at index time using the `"partial_filter_selector"` field:

```
POST /db/_index HTTP/1.1
Content-Type: application/json
Content-Length: 144
Host: localhost:5984

{
  "index": {
    "partial_filter_selector": {
      "status": {
        "$ne": "archived"
      }
    },
    "fields": ["type"]
  },
  "ddoc" : "type-not-archived",
  "type" : "json"
}
```

Partial indexes are not currently used by the query planner unless specified by a `"use_index"` field, so we need to modify the original query:

```
{
    "selector": {
        "status": {
            "$ne": "archived"
        },
        "type": "user"
    },
    "use_index": "type-not-archived"
}
```

Technically, we do not need to include the filter on the `"status"` field in the query selector - the partial index ensures this is always true - but including it makes the intent of the selector clearer and will make it easier to take advantage of future improvements to query planning (e.g. automatic selection of partial indexes).

Note: An index with fields is only used, when the selector includes all of the fields indexed. For instance, if an index contains `["a". "b"]` but the selector only requires field `["a"]` to exist in the matching documents, the index would not be valid for the query. All indexes, however, can be treated as if they include the special fields `_id` and `_rev`. They **never** need to be specified in the query selector.

GET /{db}/_index

When you make a `GET` request to `/{db}/_index`, you get a list of all indexes in the database. In addition to the information available through this API, indexes are also stored in design documents as *views*. Design documents are regular documents that have an ID starting with `_design/`. Design documents can be retrieved and modified in the same way as any other document, although this is not necessary when using Mango.

Parameters

- **db** – Database name.

Response Headers

- Content-Type –
  - *application/json*
- Transfer-Encoding – `chunked`

Response JSON Object

- **total_rows** (*number*) – Number of indexes.
- **indexes** (*array*) – Array of index definitions (see below).

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 500 Internal Server Error – Execution error

Index definitions are JSON objects with the following fields:

- **ddoc** (*string*): ID of the design document the index belongs to. This ID can be used to retrieve the design document containing the index, by making a `GET` request to `/{db}/ddoc`, where `ddoc` is the value of this field.
- **name** (*string*): Name of the index.
- **partitioned** (*boolean*): Partitioned (`true`) or global (`false`) index.
- **type** (*string*): Type of the index. Currently `"json"` is the only supported type.

- **def** (*object*): Definition of the index, containing the indexed fields and the sort order: ascending or descending.

Request:

```
GET /db/_index HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 238
Content-Type: application/json
Date: Thu, 01 Sep 2016 18:17:48 GMT
Server: CouchDB (Erlang OTP/18)

{
    "total_rows": 2,
    "indexes": [
    {
        "ddoc": null,
        "name": "_all_docs",
        "type": "special",
        "def": {
            "fields": [
                {
                    "_id": "asc"
                }
            ]
        }
    },
    {
        "ddoc": "_design/a5f4711fc9448864a13c81dc71e660b524d7410c",
        "name": "foo-index",
        "partitioned": false,
        "type": "json",
        "def": {
            "fields": [
                {
                    "foo": "asc"
                }
            ]
        }
    }
  ]
}
```

`DELETE /{db}/_index/{design_doc}/json/{name}`

Parameters

- **db** – Database name.
- **design_doc** – Design document name. The `_design/` prefix is not required.
- **name** – Index name.

Response Headers

- Content-Type –

– *application/json*

Response JSON Object

- **ok** (*string*) – *"true"* if successful.

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid request
- 401 Unauthorized – Writer permission required
- 404 Not Found – Index not found
- 500 Internal Server Error – Execution error

Request:

```
DELETE /db/_index/_design/a5f4711fc9448864a13c81dc71e660b524d7410c/json/foo-
↪index HTTP/1.1
Accept: */*
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Thu, 01 Sep 2016 19:21:40 GMT
Server: CouchDB (Erlang OTP/18)

{
    "ok": true
}
```

`POST /{db}/_index/_bulk_delete`

Parameters

- **db** – Database name

Request Headers

- Content-Type –

  – *application/json*

Request JSON Object

- **docids** (*array*) – List of names for indexes to be deleted.
- **w** (*number*) – Write quorum for each of the deletions. Default is 2. *Optional*

Response Headers

- Content-Type –

  – *application/json*

Response JSON Object

- **success** (*array*) – An array of objects that represent successful deletions per index. The `id` key contains the name of the index, and `ok` reports if the operation has completed
- **fail** (*array*) – An array of object that describe failed deletions per index. The `id` key names the corresponding index, and `error` describes the reason for the failure

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid request
- 404 Not Found – Requested database not found
- 500 Internal Server Error – Execution error

Request:

```
POST /db/_index/_bulk_delete HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: localhost:5984

{
    "docids": [
        "_design/example-ddoc",
        "foo-index",
        "nonexistent-index"
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 94
Content-Type: application/json
Date: Thu, 01 Sep 2016 19:26:59 GMT
Server: CouchDB (Erlang OTP/18)

{
    "success": [
        {
            "id": "_design/example-ddoc",
            "ok": true
        },
        {
            "id": "foo-index",
            "ok": true
        }
    ],
    "fail": [
        {
            "id": "nonexistent-index",
            "error": "not_found"
        }
    ]
}
```

12.3.8 /{db}/_explain

POST /{db}/_explain

Shows which index is being used by the query. Parameters are the same as *_find*.

Parameters

- **db** – Database name

Request Headers

- Content-Type –
  - *application/json*

Response Headers

- Content-Type –
  - *application/json*
- Transfer-Encoding – chunked

Response JSON Object

- **covering** (*boolean*) – Tell if the query could be answered only by relying on the data stored in the index. When true, no documents are fetched, which results in a faster response.
- **dbname** (*string*) – Name of database.
- **index** (*object*) – Index used to fulfill the query.
- **selector** (*object*) – Query selector used.
- **opts** (*object*) – Query options used.
- **mrargs** (*object*) – Arguments passed to the underlying view.
- **limit** (*number*) – Limit parameter used.
- **skip** (*number*) – Skip parameter used.
- **fields** (*array*) – Fields to be returned by the query. The *[]* value here means all the fields, since there is no projection happening in that case.
- **partitioned** (*boolean*) – The database is partitioned or not.
- **index_candidates** (*array*) – The list of all indexes that were found but not selected for serving the query. See *the section on index selection* below for the details.
- **selector_hints** (*object*) – Extra information on the selector to provide insights about its usability.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 500 Internal Server Error – Execution error

Request:

```
POST /movies/_explain HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 168
Host: localhost:5984
```

(continues on next page)

(continued from previous page)

```
{
    "selector": {
        "year": {"$gt": 2010}
    },
    "fields": ["_id", "_rev", "year", "title"],
    "sort": [{"year": "asc"}],
    "limit": 2,
    "skip": 0
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Thu, 01 Sep 2016 15:41:53 GMT
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked

{
    "dbname": "movies",
    "index": {
        "ddoc": "_design/0d61d9177426b1e2aa8d0fe732ec6e506f5d443c",
        "name": "0d61d9177426b1e2aa8d0fe732ec6e506f5d443c",
        "type": "json",
        "partitioned": false,
        "def": {
            "fields": [
                {
                    "year": "asc"
                }
            ]
        }
    },
    "partitioned": false,
    "selector": {
        "year": {
            "$gt": 2010
        }
    },
    "opts": {
        "use_index": [],
        "bookmark": "nil",
        "limit": 2,
        "skip": 0,
        "sort": {},
        "fields": [
            "_id",
            "_rev",
            "year",
            "title"
        ],
        "partition": "",
        "r": 1,
        "conflicts": false,
```

(continues on next page)

(continued from previous page)

```
        "stale": false,
        "update": true,
        "stable": false,
        "execution_stats": false
    },
    "limit": 2,
    "skip": 0,
    "fields": [
        "_id",
        "_rev",
        "year",
        "title"
    ],
    "mrargs": {
        "include_docs": true,
        "view_type": "map",
        "reduce": false,
        "partition": null,
        "start_key": [
            2010
        ],
        "end_key": [
            "<MAX>"
        ],
        "direction": "fwd",
        "stable": false,
        "update": true,
        "conflicts": "undefined"
    },
    "covering": false
    "index_candidates": [
        {
            "index": {
                "ddoc": null,
                "name": "_all_docs",
                "type": "special",
                "def": {
                    "fields": [
                        {
                            "_id": "asc"
                        }
                    ]
                }
            },
            "analysis": {
                "usable": true,
                "reasons": [
                    {
                        "name": "unfavored_type"
                    }
                ],
                "ranking": 1,
                "covering": null
            }
        }
    ],
```

(continues on next page)

(continued from previous page)

```
    "selector_hints": [
        {
            "type": "json",
            "indexable_fields": [
                "year"
            ],
            "unindexable_fields": []
        }
    ]
}
```

Index selection

_find chooses which index to use for responding to a query, unless you specify an index at query time. In this section, a brief overview of the index selection process is presented.

Note: It is good practice to specify indexes explicitly in your queries. This prevents existing queries being affected by new indexes that might get added in a production environment.

Note: Both the *_explain* and *_find* endpoints rely on the same index selection logic. But *_explain* is a bit more elaborate, therefore it could be used for simulation and exploration. In the output, details for discarding indexes are placed in the `analysis` field of the JSON objects under `index_candidates`. Under `analysis` the exact reason is listed in the `reasons` field. Each reason has a specific code, which will be mentioned at the relevant subsections below.

The index selection happens in multiple rounds.

Fig. 1: Steps of index selection

First, all the indexes for the database are collected. The result always includes the special entity called *all docs* which is the primary index on the `_id` field. This is reserved as a catch-all answer when no other suitable indexes could be found, but its use of discouraged for performance reasons.

In the next round, *partial indexes* are eliminated unless specified in the `use_index` field of the query object.

After that, indexes are filtered according whether a global or partitioned query was issued. Indexes that do not match the query scope are assigned a `scope_mismatch` reason code.

The remaining indexes are filtered by a series of usability checks.

Each usability check is supplied with its own reason code. That is `field_mismatch` for the cases when the fields in the index do not match with that of the selector. The code `sort_order_mismatch` means that the requested sorting does not align with the index. These checks depend on the type of index.

- `"special"`: Usable if no `sort` is specified in the query or `sort` is specified on `_id` only.
- `"json"`: The selector must not request a free-form text search via the `$text` operator. The `needs_text_search` reason code is returned otherwise.

  All the fields in the index must be referenced by the `selector` or `sort` in the query.

  Any `sort` specified in the query must match the order of the fields in the index.
- `"text"`: The index must contain fields that are referenced by the query `"selector"` or `"sort"`.

  The `"text"` indexes do not work empty selectors, and they return a `empty_selector` reason code in response to that.

After the usable indexes having gathered, the user-specified index is verified next. If this is a valid, usable index, then every other usable index is excluded with the `excluded_by_user` code. Otherwise, it is ignored an the process continues with the rest of the usable indexes.

There is a natural order of preference among the various index types: `"json"`, `"text"`, and then `"special"`. The usable indexes are grouped by their types in this order and the search is narrowed down to the elements of the first group. That is, even if there is a `"text"` index present that could match with the selector, it might be discarded if a `"json"` index with the suitable fields could be identified. Indexes dropped in this round are all tagged with the `unfavored_type` reason code.

There could be only a single `"text"` and `"special"` index per database, hence the selection ends in this phase for thoses cases. For `"json"` indexes, an additional round is run to find the ideal index.

The query planner looks at the selector section and finds the index with the closest match to operators and fields used in the query. This is described by the `less_overlap` reason code. If there are two or more `"json"`-type indexes that match, the index with the least number of fields in the index is preferred. This is marked by the `too_many_fields` reason code. If there are still two or more candidate indexes, the index with the first alphabetical name is chosen. This is reflected by the `alphabetically_comes_after` reason code.

Reason Code	Index Type	Description
`alphabetically_comes_a`	json	There is another suitable index whose name comes before that of this index.
`empty_selector`	text	`"text"` indexes do not support queries with empty selectors.
`excluded_by_user`	any	`use_index` was used to manually specify the index.
`field_mismatch`	any	Fields in `"selector"` of the query do not match with the fields available in the index.
`is_partial`	json, text	Partial indexes can be selected only manually.
`less_overlap`	json	There is a better match of fields available within the indexes for the query.
`needs_text_search`	json	The use of the `$text` operator requires a `"text"` index.
`scope_mismatch`	json	The scope of the query and the index is not the same.
`sort_order_mismatch`	json, special	Fields in `"sort"` of the query do not match with the fields available in the index.
`too_many_fields`	json	The index has more fields than the chosen one.
`unfavored_type`	any	The type of the index is not preferred.

In the *_explain* output, some additional information on the candidate indexes could be found too as part of the `analysis` object.

- The `ranking` (*number*) attribute defines a loose ordering on the items of the list, which might be used to order them. This is a positive integer which is the greater the index is farther down in the queue. Virtually, the selected index is of rank `0` always, everything else must come after that one. The rank reflects the final position of the given index candidate in the tournament described above.
- The `usable` (*Boolean*) attribute tells if the index is usable at all. This could be used to partition the index candidates by their usability in relation to the selector.
- The `covering` (*Boolean*) attribute tells if the index is a covering index or not. This property is calculated for `"json"` indexes only and it is `null` in every other case.

12.3.9 /{db}/_shards

New in version 2.0.

GET /{db}/_shards

The response will contain a list of database shards. Each shard will have its internal database range, and the nodes on which replicas of those shards are stored.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **shards** (*object*) – Mapping of shard ranges to individual shard replicas on each node in the cluster

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid database name
- 401 Unauthorized – Read privilege required
- 415 Unsupported Media Type – Bad Content-Type value
- 500 Internal Server Error – Internal server error or timeout

Request:

```
GET /db/_shards HTTP/1.1
Accept: */*
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 621
Content-Type: application/json
Date: Fri, 18 Jan 2019 19:55:14 GMT
Server: CouchDB/2.4.0 (Erlang OTP/19)

{
  "shards": {
    "00000000-1fffffff": [
      "couchdb@node1.example.com",
      "couchdb@node2.example.com",
      "couchdb@node3.example.com"
    ],
```

(continues on next page)

(continued from previous page)

```
    "20000000-3fffffff": [
      "couchdb@node1.example.com",
      "couchdb@node2.example.com",
      "couchdb@node3.example.com"
    ],
    "40000000-5fffffff": [
      "couchdb@node1.example.com",
      "couchdb@node2.example.com",
      "couchdb@node3.example.com"
    ],
    "60000000-7fffffff": [
      "couchdb@node1.example.com",
      "couchdb@node2.example.com",
      "couchdb@node3.example.com"
    ],
    "80000000-9fffffff": [
      "couchdb@node1.example.com",
      "couchdb@node2.example.com",
      "couchdb@node3.example.com"
    ],
    "a0000000-bfffffff": [
      "couchdb@node1.example.com",
      "couchdb@node2.example.com",
      "couchdb@node3.example.com"
    ],
    "c0000000-dfffffff": [
      "couchdb@node1.example.com",
      "couchdb@node2.example.com",
      "couchdb@node3.example.com"
    ],
    "e0000000-ffffffff": [
      "couchdb@node1.example.com",
      "couchdb@node2.example.com",
      "couchdb@node3.example.com"
    ]
  }
}
```

12.3.10 /{db}/_shards/{docid}

GET /{db}/_shards/{docid}

Returns information about the specific shard into which a given document has been stored, along with information about the nodes on which that shard has a replica.

Parameters

- **db** – Database name
- **docid** – Document ID

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **range** (*string*) – The shard range in which the document is stored
- **nodes** (*array*) – List of nodes serving a replica of the shard

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – Read privilege required
- 404 Not Found – Database or document not found
- 500 Internal Server Error – Internal server error or timeout

Request:

```
GET /db/_shards/docid HTTP/1.1
Accept: */*
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 94
Content-Type: application/json
Date: Fri, 18 Jan 2019 20:26:33 GMT
Server: CouchDB/2.3.0-9d4cb03c2 (Erlang OTP/19)

{
  "range": "e0000000-ffffffff",
  "nodes": [
    "node1@127.0.0.1",
    "node2@127.0.0.1",
    "node3@127.0.0.1"
  ]
}
```

12.3.11 /{db}/_sync_shards

New in version 2.3.1.

POST /{db}/_sync_shards

For the given database, force-starts internal shard synchronization for all replicas of all database shards.

This is typically only used when performing cluster maintenance, such as *moving a shard*.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **ok** (*boolean*) – Operation status. Available in case of success
- **error** (*string*) – Error type. Available if response code is `4xx`
- **reason** (*string*) – Error description. Available if response code is `4xx`

Status Codes

- 202 Accepted – Request accepted
- 400 Bad Request – Invalid database name
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – Database not found
- 500 Internal Server Error – Internal server error or timeout

Request:

```
POST /db/_sync_shards HTTP/1.1
Host: localhost:5984
Accept: */*
```

Response:

```
HTTP/1.1 202 Accepted
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Fri, 18 Jan 2019 20:19:23 GMT
Server: CouchDB/2.3.0-9d4cb03c2 (Erlang OTP/19)
X-Couch-Request-ID: 14f0b8d252
X-CouchDB-Body-Time: 0

{
    "ok": true
}
```

Note: Admins may want to bump their `[mem3] sync_concurrency` value to a larger figure for the duration of the shards sync.

12.3.12 /{db}/_changes

GET /{db}/_changes

Returns a sorted list of changes made to documents in the database, in time order of application. Only the most recent change for a given document is guaranteed to be provided, for example if a document has had fields added, and then deleted, an API client checking for changes will not necessarily receive the intermediate state of added documents.

This can be used to listen for update and modifications to the database for post processing or synchronization, and for practical purposes, a continuously connected `_changes` feed is a reasonable approach for generating a real-time log for most applications.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/event-stream*
  - *text/plain*
- Last-Event-ID – ID of the last events received by the server on a previous connection. Overrides `since` query parameter.

Query Parameters

- **doc_ids** (*array*) – List of document IDs to filter the changes feed as valid JSON array. Used with *_doc_ids* filter. Since length of URL is limited, it is better to use *POST /{db}/_changes* instead.
- **conflicts** (*boolean*) – Includes *conflicts* information in response. Ignored if *include_docs* isn't `true`. Default is `false`.
- **descending** (*boolean*) – Return the change results in descending sequence order (most recent change first). Default is `false`.
- **feed** (*string*) –
  - **normal** Specifies *Normal Polling Mode*. All past changes are returned immediately. *Default.*
  - **longpoll** Specifies *Long Polling Mode*. Waits until at least one change has occurred, sends the change, then closes the connection. Most commonly used in conjunction with `since=now`, to wait for the next change.
  - **continuous** Sets *Continuous Mode*. Sends a line of JSON per event. Keeps the socket open until `timeout`.
  - **eventsource** Sets *Event Source Mode*. Works the same as Continuous Mode, but sends the events in EventSource format.
- **filter** (*string*) –
  - **design_doc/filter_name** Reference to a *filter function* from a design document that will filter whole stream emitting only filtered events. See the section Change Notifications in the book CouchDB The Definitive Guide for more information.
  - **_doc_ids** *doc_ids filter*
  - **_view** *view filter*
  - **_design** *design filter*
- **heartbeat** (*number*) – Period in *milliseconds* after which an empty line is sent in the results. Only applicable for *longpoll*, *continuous*, and *eventsource* feeds. Overrides any timeout to keep the feed alive indefinitely. Default is `60000`. May be `true` to use default value.
- **include_docs** (*boolean*) – Include the associated document with each result. If there are conflicts, only the winning revision is returned. Default is `false`. When used with `all_docs` style and a filter, return the document body even if does not pass the filtering criteria. In other words, filtering applies only to the list of `"changes"` revision list not the returned document body in the `"doc"` field.
- **attachments** (*boolean*) – Include the Base64-encoded content of *attachments* in the documents that are included if `include_docs` is `true`. Ignored if `include_docs` isn't `true`. Default is `false`.

- **att_encoding_info** (*boolean*) – Include encoding information in attachment stubs if `include_docs` is `true` and the particular attachment is compressed. Ignored if `include_docs` isn't `true`. Default is `false`.
- **last-event-id** (*number*) – Alias of *Last-Event-ID* header.
- **limit** (*number*) – Limit number of result rows to the specified value (note that using `0` here has the same effect as `1`).
- **since** – Start the results from the change immediately after the given update sequence. Can be valid update sequence or `now` value. Default is `0`.
- **style** (*string*) – Specifies how many revisions are returned in the changes array. The default, `main_only`, will only return the current "winning" revision; `all_docs` will return all leaf revisions (including conflicts and deleted former conflicts). When using a filter with `all_docs` style, if none of the revisions match the filter, the changes row is skipped. If at least one revision matches, the changes row is returned with all matching revision. If `all_docs` style is used with `include_docs=true` and at least one revision matches the filter, the winning doc body is returned, even if it doesn't not pass the filtering criteria.
- **timeout** (*number*) – Maximum period in *milliseconds* to wait for a change before the response is sent, even if there are no results. Only applicable for *longpoll* or *continuous* feeds. Default value is specified by `chttpd/changes_timeout` configuration option. Note that `60000` value is also the default maximum timeout to prevent undetected dead connections.
- **view** (*string*) – Allows to use view functions as filters. Documents counted as "passed" for view filter in case if map function emits at least one record for them. See *_view* for more info.
- **seq_interval** (*number*) – When fetching changes in a batch, setting the *seq_interval* parameter tells CouchDB to only calculate the update seq with every Nth result returned. By setting **seq_interval=<batch size>** , where <`batch size`> is the number of results requested per batch, load can be reduced on the source CouchDB database; computing the seq value across many shards (esp. in highly-sharded databases) is expensive in a heavily loaded CouchDB cluster.

Response Headers

- Cache-Control – `no-cache` if changes feed is *eventsource*
- Content-Type –
  - *application/json*
  - *text/event-stream*
  - *text/plain; charset=utf-8*
- ETag – Response hash if changes feed is *normal*
- Transfer-Encoding – `chunked`

Response JSON Object

- **last_seq** (*json*) – Last change update sequence
- **pending** (*number*) – Count of remaining items in the feed
- **results** (*array*) – Changes made to a database

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Bad request

The `results` field of database changes:

JSON Parameters

- **changes** (*array*) – List of document's leaves with single field `rev`.
- **id** (*string*) – Document ID.
- **seq** (*json*) – Update sequence.
- **deleted** (*bool*) – `true` if the document is deleted.

Request:

```
GET /db/_changes?style=all_docs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Mon, 12 Aug 2013 00:54:58 GMT
ETag: "6ASLEKEMSRABT0O5XY9UPO9Z"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "last_seq": "5-g1AAAAIreJyVkEsKwjAURZ-
↪toI5cgq5A0sQ0OrI70XyppcaRY92J7kR3ojupaSPUUgotgRd4yTlwbw4A0zRUMLdnpaMkwmyF3Ily9xBwEIuiKLI05KOT
↪MPJX9tpEAYx4TQASns2E24ucuJ7rXJSL1BbEgf3vTwpmedCZkYa7Pulck7Xt7x_
↪usFU2aIHOD4eEfVTVA5KMGUkqhNZV-8_o5i",
    "pending": 0,
    "results": [
        {
            "changes": [
                {
                    "rev": "2-7051cbe5c8faecd085a3fa619e6e6337"
                }
            ],
            "id": "6478c2ae800dfc387396d14e1fc39626",
            "seq": "3-g1AAAAG3eJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MSGXAqSVIAkkn2IFUZzIkMuUAee5pRqnGiuXkKA2dpXkpqWmZeagpu_Q4g_
↪fGEbEkAqaqH2sIItsXAyMjM2NgUUwdOU_
↪JYgCRDA5ACGjQfn30QlQsgKvcjfGaQZmaUmmZClM8gZhyAmHGfsG0PICrBPmQC22ZqbGRqamyIqSsLAAArcXo
↪"
        },
        {
            "changes": [
                {
                    "rev": "3-7379b9e515b161226c6559d90c4dc49f"
                }
            ],
            "deleted": true,
            "id": "5bbc9ca465f1b0fcd62362168a7c8831",
            "seq": "4-g1AAAAHXeJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBMZc4EC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HqQ_
↪kQG3qgSQqnoUtxoYGZkZG5uS4NY8FiDJ0ACkgAbNx2cfROUCiMr9CJ8ZpJkZpaaZEOUziBkHIGbcJ2zbA4hKsA-
↪ZwLaZGhuZmhobYurKAgCz33kh"
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "changes": [
                {
                    "rev": "6-460637e73a6288cb24d532bf91f32969"
                },
                {
                    "rev": "5-eeaa298781f60b7bcae0c91bdedd1b87"
                }
            ],
            "id": "729eb57437745e506b333068fff665ae",
            "seq": "5-g1AAAAIReJyVkE0OgjAQRkcwUVceQU9g-
↪mOpruQm2tI2SLCuXOtN9CZ6E70JFmpCCCFCmkyTdt6bfJMDwDQNFcztWWkcY8JXyB2cu49AgFwURZGloRid3MMkEUoJHb
↪SxQWQzRVHCuYHaxSpuj1aqbj0t-3-AlSrZakn78oeSvjRSIkIhSNiCFHbsKN3c50b02mURvEB-
↪yD296eNOzzoRMRLRZ98rkHS_veGcC_nR-fGe1gaCaxihhjOI2lX0BhniHaA"
        }
    ]
}
```

Changed in version 0.11.0: added `include_docs` parameter

Changed in version 1.2.0: added `view` parameter and special value *_view* for `filter` one

Changed in version 1.3.0: `since` parameter could take *now* value to start listen changes since current seq number.

Changed in version 1.3.0: `eventsource` feed type added.

Changed in version 1.4.0: Support `Last-Event-ID` header.

Changed in version 1.6.0: added `attachments` and `att_encoding_info` parameters

Changed in version 2.0.0: update sequences can be any valid json object, added `seq_interval`

Note: If the specified replicas of the shards in any given since value are unavailable, alternative replicas are selected, and the last known checkpoint between them is used. If this happens, you might see changes again that you have previously seen. Therefore, an application making use of the `_changes` feed should be 'idempotent', that is, able to receive the same data multiple times, safely.

Note: Cloudant Sync and PouchDB already optimize the replication process by setting `seq_interval` parameter to the number of results expected per batch. This parameter increases throughput by reducing latency between sequential requests in bulk document transfers. This has resulted in up to a 20% replication performance improvement in highly-sharded databases.

Warning: Using the `attachments` parameter to include attachments in the changes feed is not recommended for large attachment sizes. Also note that the Base64-encoding that is used leads to a 33% overhead (i.e. one third) in transfer size for attachments.

Warning: The results returned by *_changes* are partially ordered. In other words, the order is not guaranteed to be preserved for multiple calls.

`POST /{db}/_changes`

Requests the database changes feed in the same way as *GET /{db}/_changes* does, but is widely used with `?filter=_doc_ids` or `?filter=_selector` query parameters and allows one to pass a larger list of document IDs or the body of the selector to filter.

Parameters

- **db** – Database name

Query Parameters

- **filter** (*string*) –
  - **_doc_ids** *doc_ids filter*
  - **_selector** *selector filter*

Request:

```
POST /recipes/_changes?filter=_doc_ids HTTP/1.1
Accept: application/json
Content-Length: 40
Content-Type: application/json
Host: localhost:5984

{
    "doc_ids": [
        "SpaghettiWithMeatballs"
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 28 Sep 2013 07:23:09 GMT
ETag: "ARIHFWL3I7PIS0SPVTFU6TLR2"
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked

{
    "last_seq": "5-g1AAAAIreJyVkEsKwjAURZ-
↪toI5cgq5A0sQ0OrI70XyppcaRY92J7kR3ojupaSPUUgotgRd4yTlwbw4A0zRUMLdnpaMkwmyF3Ily9xBwEIuiKLI05KOTW
↪MPJX9tpEAYx4TQASns2E24ucuJ7rXJSL1BbEgf3vTwpmedCZkYa7Pulck7Xt7x_
↪usFU2aIHOD4eEfVTVA5KMGUkqhNZV8_o5i",
    "pending": 0,
    "results": [
        {
            "changes": [
                {
                    "rev": "13-bcb9d6388b60fd1e960d9ec4e8e3f29e"
                }
            ],
            "id": "SpaghettiWithMeatballs",
            "seq":  "5-g1AAAAIReJyVkE0OgjAQRkcwUVceQU9g-
↪mOpruQm2tI2SLCuXOtN9CZ6E70JFmpCCCFCmkyTdt6bfJMDwDQNFcztWWkcY8JXyB2cu49AgFwURZGloRid3MMkEUoJHbX
↪SxQWQzRVHCuYHaxSpuj1aqbj0t-3-AlSrZakn78oeSvjRSIkIhSNiCFHbsKN3c50b02mURvEB-
↪yD296eNOzzoRMRLRZ98rkHS_veGcC_nR-fGe1gaCaxihhjOI2lX0BhniHaA"
        }
    ]
}
```

Request:

```
POST /db/_changes?filter=_selector HTTP/1.1
Accept: application/json
```

(continues on next page)

(continued from previous page)

```
Accept-Encoding: gzip, deflate
Content-Length: 25
Content-Type: application/json
Host: 127.0.0.1:5984

{
    "selector": {
        "data": 1
    }
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Fri, 05 Jan 2024 18:08:46 GMT
ETag: "9UTJJV90GMV3XQKBM9RNAS0IK"
Server: CouchDB/3.3.3-42c2484 (Erlang OTP/24)
Transfer-Encoding: chunked

{
    "last_seq": "4-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE5lzgQLshqkGSWmGyZjKcRqRxwIkGRqA1H-oSYxgk0ySLSxSEi0wdWUBAGlCJKQ",
    "pending": 0,
    "results": [
        {
            "changes": [
                {
                    "rev": "3-fc9d7a5cf38c9f062aa246cb072eae68"
                }
            ],
            "id": "d1",
            "seq": "4-g1AAAACTeJzLYWBgYMpgTmHgz8tPSTV0MDQy1zMAQsMckEQiQ1L9____
↪szKYE5lzgQLshqkGSWmGyZjKcRqRxwIkGRqA1H-oSYxgk0ySLSxSEi0wdWUBAGlCJKQ"
        }
    ]
}
```

Changes Feeds

Polling

By default all changes are immediately returned within the JSON body:

```
GET /somedatabase/_changes HTTP/1.1
```

```
{"results":[
{"seq":"1-g1AAAAF9eJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P__
↪7MSGXAqSVIAkkn2IFUZzIkMuUAee5pRqnGiuXkKA2dpXkpqWmZeagpu_Q4g_
↪fGEbEkAqaqH2sIItsXAyMjM2NgUUwdOU_JYgCRDA5ACGjQfn30QlQsgKvcTVnkAovI-
↪YZUPICpBvs0CAN1eY_c","id":"fresh","changes":[{"rev":"1-
↪967a00dff5e02add41819138abb3284d"}]},
{"seq":"3-g1AAAAG3eJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MSGXAqSVIAkkn2IFUZzIkMuUAee5pRqnGiuXkKA2dpXkpqWmZeagpu_Q4g_
```

(continues on next page)

(continued from previous page)

```
↪fGEbEkAqaqH2sIItsXAyMjM2NgUUwdOU_
↪JYgCRDA5ACGjQfn30QlQsgKvcjfGaQZmaUmmZClM8gZhyAmHGfsG0PICrBPmQC22ZqbGRqamyIqSsLAAArcXo
↪","id":"updated","changes":[{"rev":"2-
↪7051cbe5c8faecd085a3fa619e6e6337CFCmkyTdt6bfJMDwDQNFcztWWkcY8JXyB2cu49AgFwURZGloRid3MMkEUoJHbXbOxV
↪SxQWQzRVHCuYHaxSpuj1aqbj0t-3-AlSrZakn78oeSvjRSIkIhSNiCFHbsKN3c50b02mURvEB-
↪yD296eNOzzoRMRLRZ98rkHS_veGcC_nR-fGe1gaCaxihhjOI2lX0BhniHaA","id":"deleted","changes
↪":[{"rev":"2-eec205a9d413992850a6e32678485900"}],"deleted":true}
],
"last_seq":"5-g1AAAAIreJyVkEsKwjAURZ-
↪toI5cgq5A0sQ0OrI70XyppcaRY92J7kR3ojupaSPUUgotgRd4yTlwbw4A0zRUMLdnpaMkwmyF3Ily9xBwEIuiKLI05KOTW0wkV
↪MPJX9tpEAYx4TQASns2E24ucuJ7rXJSL1BbEgf3vTwpmedCZkYa7Pulck7Xt7x_
↪usFU2aIHOD4eEfVTVA5KMGUkqhNZV-8_o5i",
"pending": 0}
```

`results` is the list of changes in sequential order. New and changed documents only differ in the value of the rev; deleted documents include the `"deleted": true` attribute. (In the `style=all_docs` mode, deleted applies only to the current/winning revision. The other revisions listed might be deleted even if there is no deleted property; you have to `GET` them individually to make sure.)

`last_seq` is the update sequence of the last update returned (Equivalent to the last item in the results).

Sending a `since` param in the query string skips all changes up to and including the given update sequence:

```
GET /somedatabase/_changes?since=4-
↪g1AAAAHXeJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBMZc4EC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HqQ_
↪kQG3qgSQqnoUtxoYGZkZG5uS4NY8FiDJ0ACkgAbNx2cfROUCiMr9CJ8ZpJkZpaaZEOUziBkHIGbcJ2zbA4hKsA-
↪ZwLaZGhuZmhobYurKAgCz33kh HTTP/1.1
```

The return structure for `normal` and `longpoll` modes is a JSON array of changes objects, and the last update sequence.

In the return format for `continuous` mode, the server sends a `CRLF` (carriage-return, linefeed) delimited line for each change. Each line contains the *JSON object* described above.

You can also request the full contents of each document change (instead of just the change notification) by using the `include_docs` parameter.

```
{
    "last_seq": "5-g1AAAAIreJyVkEsKwjAURZ-
↪toI5cgq5A0sQ0OrI70XyppcaRY92J7kR3ojupaSPUUgotgRd4yTlwbw4A0zRUMLdnpaMkwmyF3Ily9xBwEIuiKLI05KOTW0wkV
↪MPJX9tpEAYx4TQASns2E24ucuJ7rXJSL1BbEgf3vTwpmedCZkYa7Pulck7Xt7x_
↪usFU2aIHOD4eEfVTVA5KMGUkqhNZV-8_o5i",
    "pending": 0,
    "results": [
        {
            "changes": [
                {
                    "rev": "2-eec205a9d413992850a6e32678485900"
                }
            ],
            "deleted": true,
            "id": "deleted",
            "seq":  "5-g1AAAAIReJyVkE0OgjAQRkcwUVceQU9g-
↪mOpruQm2tI2SLCuXOtN9CZ6E70JFmpCCCFCmkyTdt6bfJMDwDQNFcztWWkcY8JXyB2cu49AgFwURZGloRid3MMkEUoJHbXbOxV
↪SxQWQzRVHCuYHaxSpuj1aqbj0t-3-
↪AlSrZakn78oeSvjRSIkIhSNiCFHbsKN3c50b02mURvEByD296eNOzzoRMRLRZ98rkHS_veGcC_nR-
↪fGe1gaCaxihhjOI2lX0BhniHaA",
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Long Polling

The *longpoll* feed, probably most applicable for a browser, is a more efficient form of polling that waits for a change to occur before the response is sent. *longpoll* avoids the need to frequently poll CouchDB to discover nothing has changed!

The request to the server will remain open until a change is made on the database and is subsequently transferred, and then the connection will close. This is low load for both server and client.

The response is basically the same JSON as is sent for the *normal* feed.

Because the wait for a change can be significant you can set a timeout before the connection is automatically closed (the `timeout` argument). You can also set a heartbeat interval (using the `heartbeat` query argument), which sends a newline to keep the connection active.

Keep in mind that `heartbeat` means "Send a linefeed every `x` ms if no change arrives, and hold the connection indefinitely" while `timeout` means "Hold this connection open for `x` ms, and if no change arrives in that time, close the socket." `heartbeat` overrides `timeout`.

Continuous

Continually polling the CouchDB server is not ideal - setting up new HTTP connections just to tell the client that nothing happened puts unnecessary strain on CouchDB.

A continuous feed stays open and connected to the database until explicitly closed and changes are sent to the client as they happen, i.e. in near real-time.

As with the *longpoll* feed type you can set both the timeout and heartbeat intervals to ensure that the connection is kept open for new changes and updates.

Keep in mind that `heartbeat` means "Send a linefeed every `x` ms if no change arrives, and hold the connection indefinitely" while `timeout` means "Hold this connection open for `x` ms, and if no change arrives in that time, close the socket." `heartbeat` overrides `timeout`.

The continuous feed's response is a little different than the other feed types to simplify the job of the client - each line of the response is either empty or a JSON object representing a single change, as found in the normal feed's results.

If *limit* has been specified the feed will end with a *{ last_seq }* object.

```
GET /somedatabase/_changes?feed=continuous HTTP/1.1
```

```
{"seq":"1-g1AAAAF9eJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MSGXAqSVIAkkn2IFUZzIkMuUAee5pRqnGiuXkKA2dpXkpqWmZeagpu_Q4g_
↪fGEbEkAqaqH2sIItsXAyMjM2NgUUwdOU_JYgCRDA5ACGjQfn30QlQsgKvcTVnkAovI-
↪YZUPICpBvs0CAN1eY_c","id":"fresh","changes":[{"rev":"5-
↪g1AAAAHxeJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBOZcoEC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HkV_
↪kkGyZWqSEXH6E0D666H6GcH6DYyMzIyNTUnwRR4LkGRoAFJAg-
↪YjwiMtOdXCwJyU8ICYtABi0n6EnwzSzIxS00yI8hPEjAMQM-5nJTIQUPkAovI_UGUWAA0SgOI","id":
↪"updated","changes":[{"rev":"2-7051cbe5c8faecd085a3fa619e6e6337"}]}
{"seq":"3-g1AAAAHReJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBOZcoEC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HkV_
↪kkGyZWqSEXH6E0D660H6Exlwqspj AZIMDUAKqHA-
↪yCZGiEuTUy0MzEnxL8SkBRCT9iPcbJBmZpSaZkKUmyFmHICYcZ-wux9AVIJ8mAUABgp6XQ","id":
```

(continues on next page)

(continued from previous page)

```
↪"deleted","changes":[{"rev":"2-eec205a9d413992850a6e32678485900"}],"deleted":true}
... tum tee tum ...
{"seq":"6-
↪g1AAAAIreJyVkEsKwjAURWMrqCOXoCuQ9MU0OrI70XyppcaRY92J7kR3ojupaVNopRQsgRd4yTlwb44QmqahQnN7VjpKImAr7E
↪_-EFllst4D_-UPLXmh9VPAaICaEDUtixm-jmLie6N30YqTeYDenDmx7e9GwyYRODNuu_
↪MnnHyzverV6AMkPkAMfHO1rdUAKUkqhLZV-_0o5j","id":"updated","changes":[{"rev":"3-
↪825cb35de44c433bfb2df415563a19de"}]}
```

Obviously, … *tum tee tum* … does not appear in the actual response, but represents a long pause before the change with seq 6 occurred.

Event Source

The *eventsource* feed provides push notifications that can be consumed in the form of DOM events in the browser. Refer to the W3C eventsource specification for further details. CouchDB also honours the `Last-Event-ID` parameter.

```
GET /somedatabase/_changes?feed=eventsource HTTP/1.1
```

```
// define the event handling function
if (window.EventSource) {

    var source = new EventSource("/somedatabase/_changes?feed=eventsource");
    source.onerror = function(e) {
        alert('EventSource failed.');
    };

    var results = [];
    var sourceListener = function(e) {
        var data = JSON.parse(e.data);
        results.push(data);
    };

    // start listening for events
    source.addEventListener('message', sourceListener, false);

    // stop listening for events
    source.removeEventListener('message', sourceListener, false);

}
```

If you set a heartbeat interval (using the `heartbeat` query argument), CouchDB will send a `hearbeat` event that you can subscribe to with:

```
source.addEventListener('heartbeat', function () {}, false);
```

This can be monitored by the client application to restart the EventSource connection if needed (i.e. if the TCP connection gets stuck in a half-open state).

Note: EventSource connections are subject to cross-origin resource sharing restrictions. You might need to configure *CORS support* to get the EventSource to work in your application.

Filtering

You can filter the contents of the changes feed in a number of ways. The most basic way is to specify one or more document IDs to the query. This causes the returned structure value to only contain changes for the specified IDs. Note that the value of this query argument should be a JSON formatted array.

You can also filter the `_changes` feed by defining a filter function within a design document. The specification for the filter is the same as for replication filters. You specify the name of the filter function to the `filter` parameter, specifying the design document name and *filter name*. For example:

```
GET /db/_changes?filter=design_doc/filtername HTTP/1.1
```

Additionally, a couple of built-in filters are available and described below.

_doc_ids

This filter accepts only changes for documents which ID in specified in `doc_ids` query parameter or payload's object array. See *POST /{db}/_changes* for an example.

_selector

New in version 2.0.

This filter accepts only changes for documents which match a specified selector, defined using the same *selector syntax* used for *_find*.

This is significantly more efficient than using a JavaScript filter function and is the recommended option if filtering on document attributes only.

Note that, unlike JavaScript filters, selectors do not have access to the request object.

Request:

```
POST /recipes/_changes?filter=_selector HTTP/1.1
Content-Type: application/json
Host: localhost:5984

{
    "selector": { "_id": { "$regex": "^_design/" } }
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Tue, 06 Sep 2016 20:03:23 GMT
Etag: "1H8RGBCK3ABY6ACDM7ZSC30QK"
Server: CouchDB (Erlang OTP/18)
Transfer-Encoding: chunked

{
    "last_seq": "11-g1AAAAIreJyVkEEKwjAQRUOrqCuPoCeQZGIaXdmbaNIk1FLjyrXeRG-
↪iN9Gb1LQRaimFlsAEJnkP_s8RQtM0VGhuz0qTmABfYXdI7h4CgeSiKIosDUVwcotJIpQSOmp_
↪71TIpZty97OgymJAU8G5QrOLVdocrVbdfFzy-wYvcbLVEvrxh5K_
↪NlJggIhSNiCFHbmJbu5yonttMoneYD6kD296eNOzzoRNBNqse2Xyjpd3vP96AcYNTQY4Pt5RdTOuHIwCY5S0qewLwY6OaA
↪",
    "pending": 0,
```

(continues on next page)

(continued from previous page)


```
    "results": [
        {
            "changes": [
                {
                    "rev": "10-304cae84fd862832ea9814f02920d4b2"
                }
            ],
            "id": "_design/ingredients",
            "seq": "8-g1AAAAHxeJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBOZcoEC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HkV_
↪kkGyZWqSEXH6E0D666H6GcH6DYyMzIyNTUnwRR4LkGRoAFJAg-ZnJTIQULkAonI_
↪ws0GaWZGqWkmRLkZYsYBiBn3Cdv2AKIS7ENWsG2mxkampsaGmLqyAOYpgEo"
        },
        {
            "changes": [
                {
                    "rev": "123-6f7c1b7c97a9e4f0d22bdf130e8fd817"
                }
            ],
            "deleted": true,
            "id": "_design/cookbook",
            "seq": "9-g1AAAAHxeJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBOZcoEC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HkV_
↪kkGyZWqSEXH6E0D661F8YWBkZGZsbEqCL_JYgCRDA5ACGjQ_
↪K5GBgMoFEJX7EW42SDMzSk0zIcrNEDMOQMy4T9i2BxCVYB-ygm0zNTYyNTU2xNSVBQDnK4BL"
        },
        {
            "changes": [
                {
                    "rev": "6-5b8a52c22580e922e792047cff3618f3"
                }
            ],
            "deleted": true,
            "id": "_design/meta",
            "seq": "11-g1AAAAIReJyVkE0OgjAQRiegUVceQU9g-
↪mOpruQm2tI2SLCuXOtN9CZ6E70JFmpCCCFCmkyTdt6bfJMDwDQNFcztWWkcY8JXyB2cu49AgFwURZGloQhO7mGSCKWEjtrtnQq
↪b_ASJVstST_-UPLXRgpESEQpG5DCjlyFm7uc6F6bTKI3iA_Zhzc9vOlZZ0ImItqse2Xyjpd3vDMBfzo_
↪vrPawLiaxihhjOI2lX0BirqHbg"
        }
    ]
}
```


Missing selector

If the selector object is missing from the request body, the error message is similar to the following example:

```
{
   "error": "bad request",
   "reason": "Selector must be specified in POST payload"
}
```

Not a valid JSON object

If the selector object is not a well-formed JSON object, the error message is similar to the following example:

```
{
   "error": "bad request",
   "reason": "Selector error: expected a JSON object"
}
```

Not a valid selector

If the selector object does not contain a valid selection expression, the error message is similar to the following example:

```
{
   "error": "bad request",
   "reason": "Selector error: expected a JSON object"
}
```

_design

The `_design` filter accepts only changes for any design document within the requested database.

Request:

```
GET /recipes/_changes?filter=_design HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Tue, 06 Sep 2016 12:55:12 GMT
ETag: "ARIHFWL3I7PIS0SPVTFU6TLR2"
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked

{
    "last_seq": "11-g1AAAAIreJyVkEEKwjAQRUOrqCuPoCeQZGIaXdmbaNIk1FLjyrXeRG-
↪iN9Gb1LQRaimFlsAEJnkP_s8RQtM0VGhuz0qTmABfYXdI7h4CgeSiKIosDUVwcotJIpQSOmp_
↪71TIpZty97OgymJAU8G5QrOLVdocrVbdfFzy-wYvcbLVEvrxh5K_
↪NlJggIhSNiCFHbmJbu5yonttMoneYD6kD296eNOzzoRNBNqse2Xyjpd3vP96AcYNTQY4Pt5RdTOuHIwCY5S0qewLwY6OaA
↪",
    "pending": 0,
    "results": [
        {
            "changes": [
                {
                    "rev": "10-304cae84fd862832ea9814f02920d4b2"
                }
            ],
            "id": "_design/ingredients",
            "seq": "8-g1AAAAHxeJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
```

(continues on next page)

(continued from previous page)

```
↪7MymBOZcoEC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HkV_
↪kkGyZWqSEXH6E0D666H6GcH6DYyMzIyNTUnwRR4LkGRoAFJAg-ZnJTIQULkAonI_
↪ws0GaWZGqWkmRLkZYsYBiBn3Cdv2AKIS7ENWsG2mxkampsaGmLqyAOYpgEo"
        },
        {
            "changes": [
                {
                    "rev": "123-6f7c1b7c97a9e4f0d22bdf130e8fd817"
                }
            ],
            "deleted": true,
            "id": "_design/cookbook",
            "seq": "9-g1AAAAHxeJzLYWBg4MhgTmHgz8tPSTV0MDQy1zMAQsMcoARTIkOS_P___
↪7MymBOZcoEC7MmJKSmJqWaYynEakaQAJJPsoaYwgE1JM0o1TjQ3T2HgLM1LSU3LzEtNwa3fAaQ_HkV_
↪kkGyZWqSEXH6E0D661F8YWBkZGZsbEqCL_JYgCRDA5ACGjQ_
↪K5GBgMoFEJX7EW42SDMzSk0zIcrNEDMOQMy4T9i2BxCVYB-ygm0zNTYyNTU2xNSVBQDnK4BL"
        },
        {
            "changes": [
                {
                    "rev": "6-5b8a52c22580e922e792047cff3618f3"
                }
            ],
            "deleted": true,
            "id": "_design/meta",
            "seq": "11-g1AAAAIReJyVkE0OgjAQRiegUVceQU9g-
↪mOpruQm2tI2SLCuXOtN9CZ6E70JFmpCCCFCmkyTdt6bfJMDwDQNFcztWWkcY8JXyB2cu49AgFwURZGloQhO7mGSCKWEjtrtnQq
↪b_ASJVstST_-UPLXRgpESEQpG5DCjlyFm7uc6F6bTKI3iA_Zhzc9vOlZZ0ImItqse2Xyjpd3vDMBfzo_
↪vrPawLiaxihhjOI2lX0BirqHbg"
        }
    ]
}
```

_view

New in version 1.2.

The special filter `_view` allows to use existing *map function* as the *filter*. If the map function emits anything for the processed document it counts as accepted and the changes event emits to the feed. For most use-practice cases *filter* functions are very similar to *map* ones, so this feature helps to reduce amount of duplicated code.

> **Warning:** While *map functions* doesn't process the design documents, using `_view` filter forces them to do this. You need to be sure, that they are ready to handle documents with *alien* structure without panic.

Note: Using `_view` filter doesn't queries the view index files, so you cannot use common *view query parameters* to additionally filter the changes feed by index key. Also, CouchDB doesn't returns the result instantly as it does for views - it really uses the specified map function as filter.

Moreover, you cannot make such filters dynamic e.g. process the request query parameters or handle the *User Context Object* - the map function is only operates with the document.

Request:

```
GET /recipes/_changes?filter=_view&view=ingredients/by_recipe HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Tue, 06 Sep 2016 12:57:56 GMT
ETag: "ARIHFWL3I7PIS0SPVTFU6TLR2"
Server: CouchDB (Erlang OTP)
Transfer-Encoding: chunked

{
    "last_seq": "11-g1AAAAIreJyVkEEKwjAQRUOrqCuPoCeQZGIaXdmbaNIk1FLjyrXeRG-
↪iN9Gb1LQRaimFlsAEJnkP_s8RQtM0VGhuz0qTmABfYXdI7h4CgeSiKIosDUVwcotJIpQSOmp_
↪71TIpZty97OgymJAU8G5QrOLVdocrVbdfFzy-wYvcbLVEvrxh5K_
↪NlJggIhSNiCFHbmJbu5yonttMoneYD6kD296eNOzzoRNBNqse2Xyjpd3vP96AcYNTQY4Pt5RdTOuHIwCY5S0qewLwY6OaA
↪",
    "results": [
        {
            "changes": [
                {
                    "rev": "13-bcb9d6388b60fd1e960d9ec4e8e3f29e"
                }
            ],
            "id": "SpaghettiWithMeatballs",
            "seq": "11-g1AAAAIReJyVkE0OgjAQRiegUVceQU9g-
↪mOpruQm2tI2SLCuXOtN9CZ6E70JFmpCCCFCmkyTdt6bfJMDwDQNFcztWWkcY8JXyB2cu49AgFwURZGloQhO7mGSCKWEjtrtnQq
↪b_ASJVstST_-UPLXRgpESEQpG5DCjlyFm7uc6F6bTKI3iA_Zhzc9vOlZZ0ImItqse2Xyjpd3vDMBfzo_
↪vrPawLiaxihhjOI2lX0BirqHbg"
        }
    ]
}
```

12.3.13 /{db}/_compact

POST /{db}/_compact

Request compaction of the specified database. Compaction compresses the disk database file by performing the following operations:

- Writes a new, optimised, version of the database file, removing any unused sections from the new version during write. Because a new file is temporarily created for this purpose, you may require up to twice the current storage space of the specified database in order for the compaction routine to complete.
- Removes the bodies of any non-leaf revisions of documents from the database.
- Removes old revision history beyond the limit specified by the `_revs_limit` database parameter.

Compaction can only be requested on an individual database; you cannot compact all the databases for a CouchDB instance. The compaction process runs as a background process.

You can determine if the compaction process is operating on a database by obtaining the database meta information, the `compact_running` value of the returned database structure will be set to true. See *GET /{db}*.

You can also obtain a list of running processes to determine whether compaction is currently running. See */_active_tasks*.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **ok** (*boolean*) – Operation status

Status Codes

- 202 Accepted – Compaction request has been accepted
- 400 Bad Request – Invalid database name
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 415 Unsupported Media Type – Bad Content-Type value

Request:

```
POST /db/_compact HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 202 Accepted
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Mon, 12 Aug 2013 09:27:43 GMT
Server: CouchDB (Erlang/OTP)

{
    "ok": true
}
```

12.3.14 /{db}/_compact/{ddoc}

POST /{db}/_compact/{ddoc}

Compacts the view indexes associated with the specified design document. It may be that compacting a large view can return more storage than compacting the actual db. Thus, you can use this in place of the full database compaction if you know a specific set of view indexes have been affected by a recent database change. See *Manual View Compaction* for details.

Parameters

- **db** – Database name
- **ddoc** – Design document name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **ok** (*boolean*) – Operation status

Status Codes

- 202 Accepted – Compaction request has been accepted
- 400 Bad Request – Invalid database name
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 404 Not Found – Design document not found
- 415 Unsupported Media Type – Bad Content-Type value

Request:

```
POST /db/_compact/ddoc HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 202 Accepted
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Mon, 12 Aug 2013 09:36:44 GMT
Server: CouchDB (Erlang/OTP)

{
    "ok": true
}
```

Note: View indexes are stored in a separate `.couch` file based on a hash of the design document's relevant functions, in a sub directory of where the main `.couch` database files are located.

12.3.15 /{db}/_ensure_full_commit

POST /{db}/_ensure_full_commit

Changed in version 3.0.0: Deprecated; endpoint is a no-op.

Before 3.0 this was used to commit recent changes to the database in case the `delayed_commits=true` option was set. That option is always `false` now, so commits are never delayed. However, this endpoint is kept for compatibility with older replicators.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **instance_start_time** (*string*) – Always `"0"`. (Returned for legacy reasons.)
- **ok** (*boolean*) – Operation status

Status Codes

- 201 Created – Commit completed successfully
- 400 Bad Request – Invalid database name
- 415 Unsupported Media Type – Bad Content-Type value

Request:

```
POST /db/_ensure_full_commit HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 53
Content-Type: application/json
Date: Mon, 12 Aug 2013 10:22:19 GMT
Server: CouchDB (Erlang/OTP)
```

(continues on next page)

(continued from previous page)

```
{
    "instance_start_time": "0",
    "ok": true
}
```

12.3.16 /{db}/_view_cleanup

POST /{db}/_view_cleanup

Removes view index files that are no longer required by CouchDB as a result of changed views within design documents. As the view filename is based on a hash of the view functions, over time old views will remain, consuming storage. This call cleans up the cached view output on disk for a given view.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **ok** (*boolean*) – Operation status

Status Codes

- 202 Accepted – Compaction request has been accepted
- 400 Bad Request – Invalid database name
- 401 Unauthorized – CouchDB Server Administrator privileges required
- 415 Unsupported Media Type – Bad Content-Type value

Request:

```
POST /db/_view_cleanup HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 202 Accepted
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Mon, 12 Aug 2013 09:27:43 GMT
Server: CouchDB (Erlang/OTP)
```

(continues on next page)

(continued from previous page)

```
{
    "ok": true
}
```

12.3.17 /{db}/_security

GET /{db}/_security

Returns the current security object from the specified database.

The security object consists of two compulsory elements, `admins` and `members`, which are used to specify the list of users and/or roles that have admin and members rights to the database respectively:

- `members`: they can read all types of documents from the DB, and they can write (and edit) documents to the DB except for design documents.
- `admins`: they have all the privileges of `members` plus the privileges: write (and edit) design documents, add/remove database admins and members and set the *database revisions limit*. They can not create a database nor delete a database.

Both `members` and `admins` objects contain two array-typed fields:

- `names`: List of CouchDB user names
- `roles`: List of users roles

Any additional fields in the security object are optional. The entire security object is made available to validation and other internal functions so that the database can control and limit functionality.

If both the names and roles fields of either the admins or members properties are empty arrays, or are not existent, it means the database has no admins or members.

Having no admins, only server admins (with the reserved `_admin` role) are able to update design documents and make other admin level changes.

Having no members or roles, any user can write regular documents (any non-design document) and read documents from the database.

Since CouchDB 3.x newly created databases have by default the _admin role to prevent unintentional access.

If there are any member names or roles defined for a database, then only authenticated users having a matching name or role are allowed to read documents from the database (or do a *GET /{db}* call).

Note: If the security object for a database has never been set, then the value returned will be empty.

Also note, that security objects are not regular versioned documents (that is, they are not under MVCC rules). This is a design choice to speed up authorization checks (avoids traversing a database's documents B-Tree).

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*

- *text/plain; charset=utf-8*

Response JSON Object

- **admins** (*object*) – Object with two fields as `names` and `roles`. See description above for more info.
- **members** (*object*) – Object with two fields as `names` and `roles`. See description above for more info.

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /db/_security HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 109
Content-Type: application/json
Date: Mon, 12 Aug 2013 19:05:29 GMT
Server: CouchDB (Erlang/OTP)

{
    "admins": {
        "names": [
            "superuser"
        ],
        "roles": [
            "admins"
        ]
    },
    "members": {
        "names": [
            "user1",
            "user2"
        ],
        "roles": [
            "developers"
        ]
    }
}
```

`PUT /{db}/_security`

Sets the security object for the given database.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

- Content-Type – *application/json*

Request JSON Object

- **admins** (*object*) – Object with two fields as `names` and `roles`. *See description above for more info*.
- **members** (*object*) – Object with two fields as `names` and `roles`. *See description above for more info*.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **ok** (*boolean*) – Operation status

Status Codes

- 200 OK – Request completed successfully
- 401 Unauthorized – CouchDB Server Administrator privileges required

Request:

```
shell> curl http://adm:pass@localhost:5984/pineapple/_security -X PUT -H
↪'content-type: application/json' -H 'accept: application/json' -d '{"admins":{
↪"names":["superuser"],"roles":["admins"]},"members":{"names": ["user1","user2
↪"],"roles": ["developers"]}}'
```

```
PUT /db/_security HTTP/1.1
Accept: application/json
Content-Length: 121
Content-Type: application/json
Host: localhost:5984

{
    "admins": {
        "names": [
            "superuser"
        ],
        "roles": [
            "admins"
        ]
    },
    "members": {
        "names": [
            "user1",
            "user2"
        ],
        "roles": [
            "developers"
        ]
    }
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Tue, 13 Aug 2013 11:26:28 GMT
Server: CouchDB (Erlang/OTP)

{
    "ok": true
}
```

12.3.18 /{db}/_purge

POST /{db}/_purge

A database purge permanently removes the references to documents in the database. Normal deletion of a document within CouchDB does not remove the document from the database, instead, the document is marked as `_deleted=true` (and a new revision is created). This is to ensure that deleted documents can be replicated to other databases as having been deleted. This also means that you can check the status of a document and identify that the document has been deleted by its absence.

The purge request must include the document IDs, and for each document ID, one or more revisions that must be purged. Documents can be previously deleted, but it is not necessary. Revisions must be leaf revisions.

The response will contain a list of the document IDs and revisions successfully purged.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Request JSON Object

- **object** – Mapping of document ID to list of revisions to purge

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **purge_seq** (*string*) – Purge sequence string
- **purged** (*object*) – Mapping of document ID to list of purged revisions

Status Codes

- 201 Created – Request completed successfully
- 202 Accepted – Request was accepted, and was completed successfully on at least one replica, but quorum was not reached.
- 400 Bad Request – Invalid database name or JSON payload
- 415 Unsupported Media Type – Bad Content-Type value

- 500 Internal Server Error – Internal server error or timeout

Request:

```
POST /db/_purge HTTP/1.1
Accept: application/json
Content-Length: 76
Content-Type: application/json
Host: localhost:5984

{
    "c6114c65e295552ab1019e2b046b10e": [
        "3-b06fcd1c1c9e0ec7c480ee8aa467bf3b",
        "3-c50a32451890a3f1c3e423334cc92745"
    ]
}
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 107
Content-Type: application/json
Date: Fri, 02 Jun 2017 18:55:54 GMT
Server: CouchDB/2.0.0-2ccd4bf (Erlang OTP/18)

{
  "purge_seq": null,
  "purged": {
    "c6114c65e295552ab1019e2b046b10e": [
        "3-c50a32451890a3f1c3e423334cc92745"
      ]
  }
}
```

Fig. 2: Document Revision Tree 1

For example, given the above purge tree and issuing the above purge request, the whole document will be purged, as it contains only a single branch with a leaf revision *3-c50a32451890a3f1c3e423334cc92745* that will be purged. As a result of this purge operation, a document with *_id:c6114c65e295552ab1019e2b046b10e* will be completely removed from the database's document b+tree, and sequence b+tree. It will not be available through `_all_docs` or `_changes` endpoints, as though this document never existed. Also as a result of purge operation, the database's `purge_seq` and `update_seq` will be increased.

Notice, how revision *3-b06fcd1c1c9e0ec7c480ee8aa467bf3b* was ignored. Revisions that have already been purged and non-leaf revisions are ignored in a purge request.

If a document has two conflict revisions with the following revision history:

the above purge request will purge only one branch, leaving the document's revision tree with only a single branch:

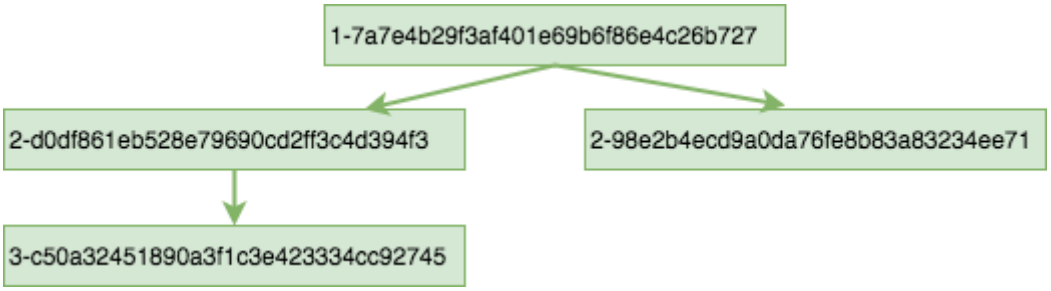

Fig. 3: Document Revision Tree 2

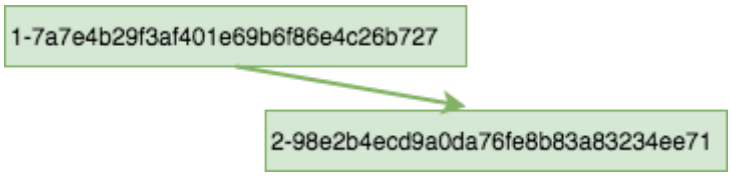

Fig. 4: Document Revision Tree 3

As a result of this purge operation, a new updated version of the document will be available in `_all_docs` and `_changes`, creating a new record in `_changes`. The database's `purge_seq` and `update_seq` will be increased.

Internal Replication

Purges are automatically replicated between replicas of the same database. Each database has an internal purge tree that stores a certain number of the most recent purges. This allows internal synchronization between replicas of the same database.

External Replication

Purge operations are not replicated to other external databases. External replication works by identifying a source's document revisions that are missing on target, and copying these revisions from source to target. A purge operation completely purges revisions from a document's purge tree making external replication of purges impossible.

> **Note:** If you need a purge to be effective across multiple effective databases, you must run the purge separately on each of the databases.

Updating Indexes

The number of purges on a database is tracked using a purge sequence. This is used by the view indexer to optimize the updating of views that contain the purged documents.

Each internal database indexer, including the view indexer, keeps its own purge sequence. The purge sequence stored in the index can be much smaller than the database's purge sequence up to the number of purge requests allowed to be stored in the purge trees of the database. Multiple purge requests can be processed by the indexer without incurring a rebuild of the index. The index will be updated according to these purge requests.

The index of documents is based on the winner of the revision tree. Depending on which revision is specified in the purge request, the index update observes the following behavior:

- If the winner of the revision tree is not specified in the purge request, there is no change to the index record of this document.
- If the winner of the revision tree is specified in the purge request, and there is still a revision left after purging, the index record of the document will be built according to the new winner of the revision tree.

- If all revisions of the document are specified in the purge request, the index record of the document will be deleted. The document will no longer be found in searches.

12.3.19 /{db}/_purged_infos

GET /{db}/_purged_infos

Get a list of purged document IDs and revisions stored in the database.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid database name

Request:

```
GET /db/_purged_infos HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 75
Content-Type: application/json
Date: Thu, 24 Aug 2023 20:56:06 GMT
Server: CouchDB (Erlang/OTP)

{
  "purged_infos": [
    {
      "id": "doc_id",
      "revs": [
        "1-85cfcb946ba8fea03ba81ec38a7a9998",
        "2-c6548393a891f2cec9c7755832ff9d6f"
      ]
    }
  ]
}
```

12.3.20 /{db}/_purged_infos_limit

GET /{db}/_purged_infos_limit

Gets the current `purged_infos_limit` (purged documents limit) setting, the maximum number of historical purges (purged document Ids with their revisions) that can be stored in the database.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /db/_purged_infos_limit HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 5
Content-Type: application/json
Date: Wed, 14 Jun 2017 14:43:42 GMT
Server: CouchDB (Erlang/OTP)

1000
```

PUT /{db}/_purged_infos_limit

Sets the maximum number of purges (requested purged Ids with their revisions) that will be tracked in the database, even after compaction has occurred. You can set the purged documents limit on a database with a scalar integer of the limit that you want to set as the request body.

The default value of historical stored purges is 1000. This means up to 1000 purges can be synchronized between replicas of the same databases in case of one of the replicas was down when purges occurred.

This request sets the soft limit for stored purges. During the compaction CouchDB will try to keep only *_purged_infos_limit* of purges in the database, but occasionally the number of stored purges can exceed this value. If a database has not completed purge synchronization with active indexes or active internal replications, it may temporarily store a higher number of historical purges.

Parameters

- **db** – Database name

Request Headers

- Accept –

- *application/json*
- *text/plain*

- Content-Type – *application/json*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **ok** (*boolean*) – Operation status

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid JSON data

Request:

```
PUT /db/_purged_infos_limit HTTP/1.1
Accept: application/json
Content-Length: 4
Content-Type: application/json
Host: localhost:5984

1500
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Wed, 14 Jun 2017 14:45:34 GMT
Server: CouchDB (Erlang/OTP)

{
    "ok": true
}
```

12.3.21 /{db}/_missing_revs

POST /{db}/_missing_revs

With given a list of document revisions, returns the document revisions that do not exist in the database.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

- Content-Type – *application/json*

Request JSON Object

- **object** – Mapping of document ID to list of revisions to lookup

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **missing_revs** (*object*) – Mapping of document ID to list of missed revisions

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid database name or JSON payload

Request:

```
POST /db/_missing_revs HTTP/1.1
Accept: application/json
Content-Length: 76
Content-Type: application/json
Host: localhost:5984

{
    "c6114c65e295552ab1019e2b046b10e": [
        "3-b06fcd1c1c9e0ec7c480ee8aa467bf3b",
        "3-0e871ef78849b0c206091f1a7af6ec41"
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 64
Content-Type: application/json
Date: Mon, 12 Aug 2013 10:53:24 GMT
Server: CouchDB (Erlang/OTP)

{
    "missing_revs":{
        "c6114c65e295552ab1019e2b046b10e": [
            "3-b06fcd1c1c9e0ec7c480ee8aa467bf3b"
        ]
    }
}
```

12.3.22 /{db}/_revs_diff

POST /{db}/_revs_diff

Given a set of document/revision IDs, returns the subset of those that do not correspond to revisions stored in the database.

Its primary use is by the replicator, as an important optimization: after receiving a set of new revision IDs from the source database, the replicator sends this set to the destination database's `_revs_diff` to find out which of them already exist there. It can then avoid fetching and sending already-known document bodies.

Both the request and response bodies are JSON objects whose keys are document IDs; but the values are structured differently:

- In the request, a value is an array of revision IDs for that document.
- In the response, a value is an object with a `missing`: key, whose value is a list of revision IDs for that document (the ones that are not stored in the database) and optionally a `possible_ancestors` key, whose value is an array of revision IDs that are known that might be ancestors of the missing revisions.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Request JSON Object

- **object** – Mapping of document ID to list of revisions to lookup

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **missing** (*array*) – List of missed revisions for specified document
- **possible_ancestors** (*array*) – List of revisions that *may be* ancestors for specified document and its current revision in requested database

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid database name or JSON payload

Request:

```
POST /db/_revs_diff HTTP/1.1
Accept: application/json
Content-Length: 113
Content-Type: application/json
Host: localhost:5984

{
    "190f721ca3411be7aa9477db5f948bbb": [
```

(continues on next page)

(continued from previous page)

```
        "3-bb72a7682290f94a985f7afac8b27137",
        "4-10265e5a26d807a3cfa459cf1a82ef2e",
        "5-067a00dff5e02add41819138abb3284d"
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 88
Content-Type: application/json
Date: Mon, 12 Aug 2013 16:56:02 GMT
Server: CouchDB (Erlang/OTP)

{
    "190f721ca3411be7aa9477db5f948bbb": {
        "missing": [
            "3-bb72a7682290f94a985f7afac8b27137",
            "5-067a00dff5e02add41819138abb3284d"
        ],
        "possible_ancestors": [
            "4-10265e5a26d807a3cfa459cf1a82ef2e"
        ]
    }
}
```

12.3.23 /{db}/_revs_limit

GET /{db}/_revs_limit

Gets the current `revs_limit` (revision limit) setting.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /db/_revs_limit HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 5
Content-Type: application/json
Date: Mon, 12 Aug 2013 17:27:30 GMT
Server: CouchDB (Erlang/OTP)

1000
```

`PUT /{db}/_revs_limit`

Sets the maximum number of document revisions that will be tracked by CouchDB, even after compaction has occurred. You can set the revision limit on a database with a scalar integer of the limit that you want to set as the request body.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type – *application/json*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **ok** (*boolean*) – Operation status

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid JSON data

Request:

```
PUT /db/_revs_limit HTTP/1.1
Accept: application/json
Content-Length: 5
Content-Type: application/json
Host: localhost:5984

1000
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 12
Content-Type: application/json
Date: Mon, 12 Aug 2013 17:47:52 GMT
Server: CouchDB (Erlang/OTP)
```

(continues on next page)

(continued from previous page)

```
{
    "ok": true
}
```

12.4 Documents

Details on how to create, read, update and delete documents within a database.

12.4.1 /{db}/{docid}

HEAD /{db}/{docid}

Returns the HTTP Headers containing a minimal amount of information about the specified document. The method supports the same query arguments as the *GET /{db}/{docid}* method, but only the header information (including document size, and the revision as an ETag), is returned.

The ETag header shows the current revision for the requested document, and the Content-Length specifies the length of the data, if the document were requested in full.

Adding any of the query arguments (see *GET /{db}/{docid}*), then the resulting HTTP Headers will correspond to what would be returned.

Parameters

- **db** – Database name
- **docid** – Document ID

Request Headers

- If-None-Match – Double quoted document's revision token

Response Headers

- Content-Length – Document size
- ETag – Double quoted document's revision token

Status Codes

- 200 OK – Document exists
- 304 Not Modified – Document wasn't modified since specified revision
- 401 Unauthorized – Read privilege required
- 404 Not Found – Document not found

Request:

```
HEAD /db/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 660
Content-Type: application/json
Date: Tue, 13 Aug 2013 21:35:37 GMT
```

(continues on next page)

(continued from previous page)

```
ETag: "12-151bb8678d45aaa949ec3698ef1c7e78"
Server: CouchDB (Erlang/OTP)
```

GET /{db}/{docid}

Returns document by the specified `docid` from the specified `db`. Unless you request a specific revision, the latest revision of the document will always be returned.

Parameters

- **db** – Database name
- **docid** – Document ID

Request Headers

- Accept –
  - *application/json*
  - *multipart/related*
  - *multipart/mixed*
  - *text/plain*
- If-None-Match – Double quoted document's revision token

Query Parameters

- **attachments** (*boolean*) – Includes attachments bodies in response. Default is `false`
- **att_encoding_info** (*boolean*) – Includes encoding information in attachment stubs if the particular attachment is compressed. Default is `false`.
- **atts_since** (*array*) – Includes attachments only since specified revisions. Doesn't includes attachments for specified revisions. *Optional*
- **conflicts** (*boolean*) – Includes information about conflicts in document. Default is `false`
- **deleted_conflicts** (*boolean*) – Includes information about deleted conflicted revisions. Default is `false`
- **latest** (*boolean*) – Forces retrieving latest "leaf" revision, no matter what *rev* was requested. Default is `false`
- **local_seq** (*boolean*) – Includes last update sequence for the document. Default is `false`
- **meta** (*boolean*) – Acts same as specifying all `conflicts`, `deleted_conflicts` and `revs_info` query parameters. Default is `false`
- **open_revs** (*array*) – Retrieves documents of specified leaf revisions. Additionally, it accepts value as `all` to return all leaf revisions. *Optional*
- **rev** (*string*) – Retrieves document of specified revision. *Optional*
- **revs** (*boolean*) – Includes list of all known document revisions. Default is `false`
- **revs_info** (*boolean*) – Includes detailed information for all known document revisions. Default is `false`

Response Headers

- Content-Type –
  - *application/json*
  - *multipart/related*
  - *multipart/mixed*

– *text/plain; charset=utf-8*

- ETag – Double quoted document's revision token. Not available when retrieving conflicts-related information
- Transfer-Encoding – `chunked`. Available if requested with query parameter `open_revs`

Response JSON Object

- **`_id`** (*string*) – Document ID
- **`_rev`** (*string*) – Revision MVCC token
- **`_deleted`** (*boolean*) – Deletion flag. Available if document was removed
- **`_attachments`** (*object*) – Attachment's stubs. Available if document has any attachments
- **`_conflicts`** (*array*) – List of conflicted revisions. Available if requested with `conflicts=true` query parameter
- **`_deleted_conflicts`** (*array*) – List of deleted conflicted revisions. Available if requested with `deleted_conflicts=true` query parameter
- **`_local_seq`** (*string*) – Document's update sequence in current database. Available if requested with `local_seq=true` query parameter
- **`_revs_info`** (*array*) – List of objects with information about local revisions and their status. Available if requested with `open_revs` query parameter
- **`_revisions`** (*object*) – List of local revision tokens without. Available if requested with `revs=true` query parameter

Status Codes

- 200 OK – Request completed successfully
- 304 Not Modified – Document wasn't modified since specified revision
- 400 Bad Request – The format of the request or revision was invalid
- 401 Unauthorized – Read privilege required
- 404 Not Found – Document not found

Request:

```
GET /recipes/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 660
Content-Type: application/json
Date: Tue, 13 Aug 2013 21:35:37 GMT
ETag: "1-917fa2381192822767f010b95b45325b"
Server: CouchDB (Erlang/OTP)

{
    "_id": "SpaghettiWithMeatballs",
    "_rev": "1-917fa2381192822767f010b95b45325b",
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
```

(continues on next page)

(continued from previous page)

```
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
```

PUT /{db}/{docid}

The PUT method creates a new named document, or creates a new revision of the existing document. Unlike the *POST /{db}*, you must specify the document ID in the request URL.

When updating an existing document, the current document revision must be included in the document (i.e. the request body), as the `rev` query parameter, or in the `If-Match` request header.

Parameters

- **db** – Database name
- **docid** – Document ID

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Content-Type –
  - *application/json*
  - *multipart/related*
- If-Match – Document's revision. Alternative to *rev* query parameter or document key. *Optional*

Query Parameters

- **rev** (*string*) – Document's revision if updating an existing document. Alternative to `If-Match` header or document key. *Optional*
- **batch** (*string*) – Stores document in *batch mode*. Possible values: `ok`. *Optional*
- **new_edits** (*boolean*) – Prevents insertion of a *conflicting document*. Possible values: `true` (default) and `false`. If `false`, a well-formed `_rev` must be included in the document. `new_edits=false` is used by the replicator to insert documents into the target database even if that leads to the creation of conflicts. *Optional*, **The ``false`` value is intended for use only by the replicator.**

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
  - *multipart/related*
- ETag – Quoted document's new revision
- Location – Document URI

Response JSON Object

- **id** (*string*) – Document ID
- **ok** (*boolean*) – Operation status

- **rev** (*string*) – Revision MVCC token

Status Codes

- 201 Created – Document created and stored on disk
- 202 Accepted – Document data accepted, but not yet stored on disk
- 400 Bad Request – Invalid request body or parameters
- 401 Unauthorized – Write privileges required
- 404 Not Found – Specified database or document ID doesn't exists
- 409 Conflict – Document with the specified ID already exists or specified revision is not latest for target document

Request:

```
PUT /recipes/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Content-Length: 196
Content-Type: application/json
Host: localhost:5984

{
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 85
Content-Type: application/json
Date: Wed, 14 Aug 2013 20:31:39 GMT
ETag: "1-917fa2381192822767f010b95b45325b"
Location: http://localhost:5984/recipes/SpaghettiWithMeatballs
Server: CouchDB (Erlang/OTP)

{
    "id": "SpaghettiWithMeatballs",
    "ok": true,
    "rev": "1-917fa2381192822767f010b95b45325b"
}
```

DELETE /{db}/{docid}

Marks the specified document as deleted by adding a field `_deleted` with the value `true`. Documents with this field will not be returned within requests anymore, but stay in the database. You must supply the current (latest) revision, either by using the `rev` parameter or by using the If-Match header to specify the revision.

Note: CouchDB doesn't completely delete the specified document. Instead, it leaves a tombstone with very basic information about the document. The tombstone is required so that the delete action can be replicated across databases.

See also:

Retrieving Deleted Documents

Parameters

- **db** – Database name
- **docid** – Document ID

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- If-Match – Document's revision. Alternative to *rev* query parameter

Query Parameters

- **rev** (*string*) – Actual document's revision
- **batch** (*string*) – Stores document in *batch mode* Possible values: `ok`. *Optional*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- ETag – Double quoted document's new revision

Response JSON Object

- **id** (*string*) – Document ID
- **ok** (*boolean*) – Operation status
- **rev** (*string*) – Revision MVCC token

Status Codes

- 200 OK – Document successfully removed
- 202 Accepted – Request was accepted, but changes are not yet stored on disk
- 400 Bad Request – Invalid request body or parameters
- 401 Unauthorized – Write privileges required
- 404 Not Found – Specified database or document ID doesn't exists
- 409 Conflict – Specified revision is not the latest for target document

Request:

```
DELETE /recipes/FishStew?rev=1-9c65296036141e575d32ba9c034dd3ee HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Alternatively, instead of `rev` query parameter you may use If-Match header:

```
DELETE /recipes/FishStew HTTP/1.1
Accept: application/json
If-Match: 1-9c65296036141e575d32ba9c034dd3ee
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 71
Content-Type: application/json
Date: Wed, 14 Aug 2013 12:23:13 GMT
ETag: "2-056f5f44046ecafc08a2bc2b9c229e20"
Server: CouchDB (Erlang/OTP)

{
    "id": "FishStew",
    "ok": true,
    "rev": "2-056f5f44046ecafc08a2bc2b9c229e20"
}
```

COPY /{db}/{docid}

The COPY (which is non-standard HTTP) copies an existing document to a new or existing document. Copying a document is only possible within the same database.

The source document is specified on the request line, with the Destination header of the request specifying the target document.

Parameters

- **db** – Database name
- **docid** – Document ID

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- Destination – Destination document. Must contain the target document ID, and optionally the target document revision, if copying to an existing document. See *Copying to an Existing Document*.
- If-Match – Source document's revision. Alternative to `rev` query parameter

Query Parameters

- **rev** (*string*) – Revision to copy from. *Optional*
- **batch** (*string*) – Stores document in *batch mode* Possible values: `ok`. *Optional*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- ETag – Double quoted document's new revision
- Location – Document URI

Response JSON Object

- **id** (*string*) – Document document ID
- **ok** (*boolean*) – Operation status
- **rev** (*string*) – Revision MVCC token

Status Codes

- 201 Created – Document successfully created

- 202 Accepted – Request was accepted, but changes are not yet stored on disk
- 400 Bad Request – Invalid request body or parameters
- 401 Unauthorized – Read or write privileges required
- 404 Not Found – Specified database, document ID or revision doesn't exists
- 409 Conflict – Document with the specified ID already exists or specified revision is not latest for target document

Request:

```
COPY /recipes/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Destination: SpaghettiWithMeatballs_Italian
Host: localhost:5984
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 93
Content-Type: application/json
Date: Wed, 14 Aug 2013 14:21:00 GMT
ETag: "1-e86fdf912560c2321a5fcefc6264e6d9"
Location: http://localhost:5984/recipes/SpaghettiWithMeatballs_Italian
Server: CouchDB (Erlang/OTP)

{
    "id": "SpaghettiWithMeatballs_Italian",
    "ok": true,
    "rev": "1-e86fdf912560c2321a5fcefc6264e6d9"
}
```

Attachments

If the document includes attachments, then the returned structure will contain a summary of the attachments associated with the document, but not the attachment data itself.

The JSON for the returned document will include the `_attachments` field, with one or more attachment definitions.

The `_attachments` object keys are attachments names while values are information objects with next structure:

- **content_type** (*string*): Attachment MIME type
- **data** (*string*): Base64-encoded content. Available if attachment content is requested by using the following query parameters:
  - `attachments=true` when querying a document
  - `attachments=true&include_docs=true` when querying a *changes feed* or a *view*
  - `atts_since`.
- **digest** (*string*): Content hash digest. It starts with prefix which announce hash type (`md5-`) and continues with Base64-encoded hash digest
- **encoded_length** (*number*): Compressed attachment size in bytes. Available if `content_type` is in *`list of compressible types`* when the attachment was added and the following query parameters are specified:
  - `att_encoding_info=true` when querying a document
  - `att_encoding_info=true&include_docs=true` when querying a *changes feed* or a *view*

- **encoding** (*string*): Compression codec. Available if `content_type` is in *`list of compressible types`* when the attachment was added and the following query parameters are specified:
  - `att_encoding_info=true` when querying a document
  - `att_encoding_info=true&include_docs=true` when querying a *changes feed* or a *view*
- **length** (*number*): Real attachment size in bytes. Not available if attachment content requested
- **revpos** (*number*): Revision *number* when attachment was added
- **stub** (*boolean*): Has `true` value if object contains stub info and no content. Otherwise omitted in response

Basic Attachments Info

Request:

```
GET /recipes/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 660
Content-Type: application/json
Date: Tue, 13 Aug 2013 21:35:37 GMT
ETag: "5-fd96acb3256302bf0dd2f32713161f2a"
Server: CouchDB (Erlang/OTP)

{
    "_attachments": {
        "grandma_recipe.txt": {
            "content_type": "text/plain",
            "digest": "md5-Ids41vtv725jyrN7iUvMcQ==",
            "length": 1872,
            "revpos": 4,
            "stub": true
        },
        "my_recipe.txt": {
            "content_type": "text/plain",
            "digest": "md5-198BPPNiT5fqlLxoYYbjBA==",
            "length": 85,
            "revpos": 5,
            "stub": true
        },
        "photo.jpg": {
            "content_type": "image/jpeg",
            "digest": "md5-7Pv4HW2822WY1r/3WDbPug==",
            "length": 165504,
            "revpos": 2,
            "stub": true
        }
    },
    "_id": "SpaghettiWithMeatballs",
    "_rev": "5-fd96acb3256302bf0dd2f32713161f2a",
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
```

(continues on next page)

(continued from previous page)

```
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
```

Retrieving Attachments Content

It's possible to retrieve document with all attached files content by using `attachments=true` query parameter:

Request:

```
GET /db/pixel?attachments=true HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 553
Content-Type: application/json
Date: Wed, 14 Aug 2013 11:32:40 GMT
ETag: "4-f1bcae4bf7bbb92310079e632abfe3f4"
Server: CouchDB (Erlang/OTP)

{
    "_attachments": {
        "pixel.gif": {
            "content_type": "image/gif",
            "data": "R0lGODlhAQABAIAAAAAAAP///yH5BAEAAAAALAAAAAABAAEAAAIBRAA7",
            "digest": "md5-2JdGiI2i2VELZKnwMers1Q==",
            "revpos": 2
        },
        "pixel.png": {
            "content_type": "image/png",
            "data":
↪"iVBORw0KGgoAAAANSUhEUgAAAAEAAAABAQMAAAAl21bKAAAAAXNSR0IArs4c6QAAAANQTFRFAAAAp3o92gAAAAF0Uk5TAEDm2
↪",
            "digest": "md5-Dgf5zxgGuchWrve73evvGQ==",
            "revpos": 3
        }
    },
    "_id": "pixel",
    "_rev": "4-f1bcae4bf7bbb92310079e632abfe3f4"
}
```

Or retrieve attached files content since specific revision using `atts_since` query parameter:

Request:

```
GET /recipes/SpaghettiWithMeatballs?atts_since=[%224-874985bc28906155ba0e2e0538f67b05
↪%22]  HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 760
Content-Type: application/json
Date: Tue, 13 Aug 2013 21:35:37 GMT
ETag: "5-fd96acb3256302bf0dd2f32713161f2a"
Server: CouchDB (Erlang/OTP)

{
    "_attachments": {
        "grandma_recipe.txt": {
            "content_type": "text/plain",
            "digest": "md5-Ids41vtv725jyrN7iUvMcQ==",
            "length": 1872,
            "revpos": 4,
            "stub": true
        },
        "my_recipe.txt": {
            "content_type": "text/plain",
            "data":
↪"MS4gQ29vayBzcGFnaGV0dGkKMi4gQ29vayBtZWV0YmFsbHMKMy4gTWl4IHRoZW0KNC4gQWRkIHRvbWF0byBzYXVjZQo1LiAuL
↪",
            "digest": "md5-198BPPNiT5fqlLxoYYbjBA==",
            "revpos": 5
        },
        "photo.jpg": {
            "content_type": "image/jpeg",
            "digest": "md5-7Pv4HW2822WY1r/3WDbPug==",
            "length": 165504,
            "revpos": 2,
            "stub": true
        }
    },
    "_id": "SpaghettiWithMeatballs",
    "_rev": "5-fd96acb3256302bf0dd2f32713161f2a",
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
```

Efficient Multiple Attachments Retrieving

As noted above, retrieving document with `attachments=true` returns a large JSON object with all attachments included. When your document and files are smaller it's ok, but if you have attached something bigger like media files (audio/video), parsing such response might be very expensive.

To solve this problem, CouchDB allows to get documents in *`multipart/related`* format:

Request:

```
GET /recipes/secret?attachments=true HTTP/1.1
Accept: multipart/related
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 538
Content-Type: multipart/related; boundary="e89b3e29388aef23453450d10e5aaed0"
Date: Sat, 28 Sep 2013 08:08:22 GMT
ETag: "2-c1c6c44c4bc3c9344b037c8690468605"
Server: CouchDB (Erlang OTP)

--e89b3e29388aef23453450d10e5aaed0
Content-Type: application/json

{"_id":"secret","_rev":"2-c1c6c44c4bc3c9344b037c8690468605","_attachments":{"recipe.
↪txt":{"content_type":"text/plain","revpos":2,"digest":"md5-HV9aXJdEnu0xnMQYTKgOFA==
↪","length":86,"follows":true}}}
--e89b3e29388aef23453450d10e5aaed0
Content-Disposition: attachment; filename="recipe.txt"
Content-Type: text/plain
Content-Length: 86

1. Take R
2. Take E
3. Mix with L
4. Add some A
5. Serve with X

--e89b3e29388aef23453450d10e5aaed0--
```

In this response the document contains only attachments stub information and quite short while all attachments goes as separate entities which reduces memory footprint and processing overhead (you'd noticed, that attachment content goes as raw data, not in base64 encoding, right?).

Retrieving Attachments Encoding Info

By using `att_encoding_info=true` query parameter you may retrieve information about compressed attachments size and used codec.

Request:

```
GET /recipes/SpaghettiWithMeatballs?att_encoding_info=true HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 736
Content-Type: application/json
Date: Tue, 13 Aug 2013 21:35:37 GMT
ETag: "5-fd96acb3256302bf0dd2f32713161f2a"
Server: CouchDB (Erlang/OTP)

{
    "_attachments": {
        "grandma_recipe.txt": {
            "content_type": "text/plain",
            "digest": "md5-Ids41vtv725jyrN7iUvMcQ==",
            "encoded_length": 693,
            "encoding": "gzip",
            "length": 1872,
            "revpos": 4,
            "stub": true
        },
        "my_recipe.txt": {
            "content_type": "text/plain",
            "digest": "md5-198BPPNiT5fqlLxoYYbjBA==",
            "encoded_length": 100,
            "encoding": "gzip",
            "length": 85,
            "revpos": 5,
            "stub": true
        },
        "photo.jpg": {
            "content_type": "image/jpeg",
            "digest": "md5-7Pv4HW2822WY1r/3WDbPug==",
            "length": 165504,
            "revpos": 2,
            "stub": true
        }
    },
    "_id": "SpaghettiWithMeatballs",
    "_rev": "5-fd96acb3256302bf0dd2f32713161f2a",
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
```

Creating Multiple Attachments

To create a document with multiple attachments with single request you need just inline base64 encoded attachments data into the document body:

```
{
  "_id":"multiple_attachments",
  "_attachments":
  {
    "foo.txt":
    {
      "content_type":"text\/plain",
      "data": "VGhpcyBpcyBhIGJhc2U2NCBlbmNvZGVkIHRleHQ="
    },

   "bar.txt":
    {
      "content_type":"text\/plain",
      "data": "VGhpcyBpcyBhIGJhc2U2NCBlbmNvZGVkIHRleHQ="
    }
  }
}
```

Alternatively, you can upload a document with attachments more efficiently in *multipart/related* format. This avoids having to Base64-encode the attachments, saving CPU and bandwidth. To do this, set the Content-Type header of the *PUT /{db}/{docid}* request to *multipart/related*.

The first MIME body is the document itself, which should have its own Content-Type of *application/json"*. It also should include an `_attachments` metadata object in which each attachment object has a key `follows` with value `true`.

The subsequent MIME bodies are the attachments.

Request:

```
PUT /temp/somedoc HTTP/1.1
Accept: application/json
Content-Length: 372
Content-Type: multipart/related;boundary="abc123"
Host: localhost:5984
User-Agent: HTTPie/0.6.0

--abc123
Content-Type: application/json

{
    "body": "This is a body.",
    "_attachments": {
        "foo.txt": {
            "follows": true,
            "content_type": "text/plain",
            "length": 21
        },
        "bar.txt": {
            "follows": true,
            "content_type": "text/plain",
            "length": 20
        }
    }
```

(continues on next page)

(continued from previous page)

```
}

--abc123

this is 21 chars long
--abc123

this is 20 chars lon
--abc123--
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 72
Content-Type: application/json
Date: Sat, 28 Sep 2013 09:13:24 GMT
ETag: "1-5575e26acdeb1df561bb5b70b26ba151"
Location: http://localhost:5984/temp/somedoc
Server: CouchDB (Erlang OTP)

{
    "id": "somedoc",
    "ok": true,
    "rev": "1-5575e26acdeb1df561bb5b70b26ba151"
}
```

Getting a List of Revisions

You can obtain a list of the revisions for a given document by adding the `revs=true` parameter to the request URL:

Request:

```
GET /recipes/SpaghettiWithMeatballs?revs=true  HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 584
Content-Type: application/json
Date: Wed, 14 Aug 2013 11:38:26 GMT
ETag: "5-fd96acb3256302bf0dd2f32713161f2a"
Server: CouchDB (Erlang/OTP)

{
    "_id": "SpaghettiWithMeatballs",
    "_rev": "8-6f5ad8db0f34af24a6e0984cd1a6cfb9",
    "_revisions": {
        "ids": [
            "6f5ad8db0f34af24a6e0984cd1a6cfb9",
            "77fba3a059497f51ec99b9b478b569d2",
            "136813b440a00a24834f5cb1ddf5b1f1",
            "fd96acb3256302bf0dd2f32713161f2a",
```

(continues on next page)

(continued from previous page)

```
            "874985bc28906155ba0e2e0538f67b05",
            "0de77a37463bf391d14283e626831f2e",
            "d795d1b924777732fdea76538c558b62",
            "917fa2381192822767f010b95b45325b"
        ],
        "start": 8
    },
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
```

The returned JSON structure includes the original document, including a `_revisions` structure that includes the revision information in next form:

- **ids** (*array*): Array of valid revision IDs, in reverse order (latest first)
- **start** (*number*): Prefix number for the latest revision

Obtaining an Extended Revision History

You can get additional information about the revisions for a given document by supplying the `revs_info` argument to the query:

Request:

```
GET /recipes/SpaghettiWithMeatballs?revs_info=true  HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 802
Content-Type: application/json
Date: Wed, 14 Aug 2013 11:40:55 GMT
Server: CouchDB (Erlang/OTP)

{
    "_id": "SpaghettiWithMeatballs",
    "_rev": "8-6f5ad8db0f34af24a6e0984cd1a6cfb9",
    "_revs_info": [
        {
            "rev": "8-6f5ad8db0f34af24a6e0984cd1a6cfb9",
            "status": "available"
        },
        {
            "rev": "7-77fba3a059497f51ec99b9b478b569d2",
            "status": "deleted"
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "rev": "6-136813b440a00a24834f5cb1ddf5b1f1",
            "status": "available"
        },
        {
            "rev": "5-fd96acb3256302bf0dd2f32713161f2a",
            "status": "missing"
        },
        {
            "rev": "4-874985bc28906155ba0e2e0538f67b05",
            "status": "missing"
        },
        {
            "rev": "3-0de77a37463bf391d14283e626831f2e",
            "status": "missing"
        },
        {
            "rev": "2-d795d1b924777732fdea76538c558b62",
            "status": "missing"
        },
        {
            "rev": "1-917fa2381192822767f010b95b45325b",
            "status": "missing"
        }
    ],
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
```

The returned document contains `_revs_info` field with extended revision information, including the availability and status of each revision. This array field contains objects with following structure:

- **rev** (*string*): Full revision string
- **status** (*string*): Status of the revision. Maybe one of:
  - `available`: Revision is available for retrieving with *rev* query parameter
  - `missing`: Revision is not available
  - `deleted`: Revision belongs to deleted document

Obtaining a Specific Revision

To get a specific revision, use the `rev` argument to the request, and specify the full revision number. The specified revision of the document will be returned, including a `_rev` field specifying the revision that was requested.

Request:

```
GET /recipes/SpaghettiWithMeatballs?rev=6-136813b440a00a24834f5cb1ddf5b1f1  HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 271
Content-Type: application/json
Date: Wed, 14 Aug 2013 11:40:55 GMT
Server: CouchDB (Erlang/OTP)

{
    "_id": "SpaghettiWithMeatballs",
    "_rev": "6-136813b440a00a24834f5cb1ddf5b1f1",
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs"
}
```

Retrieving Deleted Documents

CouchDB doesn't actually delete documents via *DELETE /{db}/{docid}*. Instead, it leaves tombstone with very basic information about the document. If you just *GET /{db}/{docid}* CouchDB returns 404 Not Found response:

Request:

```
GET /recipes/FishStew  HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 404 Object Not Found
Cache-Control: must-revalidate
Content-Length: 41
Content-Type: application/json
Date: Wed, 14 Aug 2013 12:23:27 GMT
Server: CouchDB (Erlang/OTP)

{
    "error": "not_found",
    "reason": "deleted"
}
```

However, you may retrieve document's tombstone by using `rev` query parameter with *GET /{db}/{docid}* request:

Request:

```
GET /recipes/FishStew?rev=2-056f5f44046ecafc08a2bc2b9c229e20  HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 79
Content-Type: application/json
Date: Wed, 14 Aug 2013 12:30:22 GMT
ETag: "2-056f5f44046ecafc08a2bc2b9c229e20"
Server: CouchDB (Erlang/OTP)

{
    "_deleted": true,
    "_id": "FishStew",
    "_rev": "2-056f5f44046ecafc08a2bc2b9c229e20"
}
```

Updating an Existing Document

To update an existing document you must specify the current revision number within the `_rev` parameter.

Request:

```
PUT /recipes/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Content-Length: 258
Content-Type: application/json
Host: localhost:5984

{
    "_rev": "1-917fa2381192822767f010b95b45325b",
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs",
    "serving": "hot"
}
```

Alternatively, you can supply the current revision number in the `If-Match` HTTP header of the request:

```
PUT /recipes/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Content-Length: 258
Content-Type: application/json
If-Match: 1-917fa2381192822767f010b95b45325b
Host: localhost:5984

{
    "description": "An Italian-American dish that usually consists of spaghetti,␣
↪tomato sauce and meatballs.",
    "ingredients": [
        "spaghetti",
        "tomato sauce",
        "meatballs"
    ],
    "name": "Spaghetti with meatballs",
```

(continues on next page)

(continued from previous page)

```
    "serving": "hot"
}
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 85
Content-Type: application/json
Date: Wed, 14 Aug 2013 20:33:56 GMT
ETag: "2-790895a73b63fb91dd863388398483dd"
Location: http://localhost:5984/recipes/SpaghettiWithMeatballs
Server: CouchDB (Erlang/OTP)

{
    "id": "SpaghettiWithMeatballs",
    "ok": true,
    "rev": "2-790895a73b63fb91dd863388398483dd"
}
```

Copying from a Specific Revision

To copy *from* a specific version, use the `rev` argument to the query string or If-Match:

Request:

```
COPY /recipes/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Destination: SpaghettiWithMeatballs_Original
If-Match: 1-917fa2381192822767f010b95b45325b
Host: localhost:5984
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 93
Content-Type: application/json
Date: Wed, 14 Aug 2013 14:21:00 GMT
ETag: "1-917fa2381192822767f010b95b45325b"
Location: http://localhost:5984/recipes/SpaghettiWithMeatballs_Original
Server: CouchDB (Erlang/OTP)

{
    "id": "SpaghettiWithMeatballs_Original",
    "ok": true,
    "rev": "1-917fa2381192822767f010b95b45325b"
}
```

Copying to an Existing Document

To copy to an existing document, you must specify the current revision string for the target document by appending the `rev` parameter to the Destination header string.

Request:

```
COPY /recipes/SpaghettiWithMeatballs?rev=8-6f5ad8db0f34af24a6e0984cd1a6cfb9 HTTP/1.1
Accept: application/json
Destination: SpaghettiWithMeatballs_Original?rev=1-917fa2381192822767f010b95b45325b
Host: localhost:5984
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 93
Content-Type: application/json
Date: Wed, 14 Aug 2013 14:21:00 GMT
ETag: "2-62e778c9ec09214dd685a981dcc24074""
Location: http://localhost:5984/recipes/SpaghettiWithMeatballs_Original
Server: CouchDB (Erlang/OTP)

{
    "id": "SpaghettiWithMeatballs_Original",
    "ok": true,
    "rev": "2-62e778c9ec09214dd685a981dcc24074"
}
```

12.4.2 /{db}/{docid}/{attname}

HEAD /{db}/{docid}/{attname}

Returns the HTTP headers containing a minimal amount of information about the specified attachment. The method supports the same query arguments as the *GET /{db}/{docid}/{attname}* method, but only the header information (including attachment size, encoding and the MD5 hash as an ETag), is returned.

Parameters

- **db** – Database name
- **docid** – Document ID
- **attname** – Attachment name

Request Headers

- If-Match – Document's revision. Alternative to `rev` query parameter
- If-None-Match – Attachment's base64 encoded MD5 binary digest. *Optional*

Query Parameters

- **rev** (*string*) – Document's revision. *Optional*

Response Headers

- Accept-Ranges – *Range request aware*. Used for attachments with *application/octet-stream* content type
- Content-Encoding – Used compression codec. Available if attachment's `content_type` is in *list of compressible types*
- Content-Length – Attachment size. If compression codec was used, this value is about compressed size, not actual

- ETag – Double quoted base64 encoded MD5 binary digest

Status Codes

- 200 OK – Attachment exists
- 401 Unauthorized – Read privilege required
- 404 Not Found – Specified database, document or attachment was not found

Request:

```
HEAD /recipes/SpaghettiWithMeatballs/recipe.txt HTTP/1.1
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Accept-Ranges: none
Cache-Control: must-revalidate
Content-Encoding: gzip
Content-Length: 100
Content-Type: text/plain
Date: Thu, 15 Aug 2013 12:42:42 GMT
ETag: "vVa/YgiE1+Gh0WfoFJAcSg=="
Server: CouchDB (Erlang/OTP)
```

GET /{db}/{docid}/{attname}

Returns the file attachment associated with the document. The raw data of the associated attachment is returned (just as if you were accessing a static file. The returned Content-Type will be the same as the content type set when the document attachment was submitted into the database.

Parameters

- **db** – Database name
- **docid** – Document ID
- **attname** – Attachment name

Request Headers

- If-Match – Document's revision. Alternative to `rev` query parameter
- If-None-Match – Attachment's base64 encoded MD5 binary digest. *Optional*

Query Parameters

- **rev** (*string*) – Document's revision. *Optional*

Response Headers

- Accept-Ranges – *Range request aware*. Used for attachments with *application/octet-stream*
- Content-Encoding – Used compression codec. Available if attachment's `content_type` is in *list of compressible types*
- Content-Length – Attachment size. If compression codec is used, this value is about compressed size, not actual
- ETag – Double quoted base64 encoded MD5 binary digest

Response

Stored content

Status Codes

- 200 OK – Attachment exists

- 401 Unauthorized – Read privilege required
- 404 Not Found – Specified database, document or attachment was not found

PUT /{db}/{docid}/{attname}

Uploads the supplied content as an attachment to the specified document. The attachment name provided must be a URL encoded string. You must supply the Content-Type header, and for an existing document you must also supply either the `rev` query argument or the If-Match HTTP header. If the revision is omitted, a new, otherwise empty document will be created with the provided attachment, or a conflict will occur.

If case when uploading an attachment using an existing attachment name, CouchDB will update the corresponding stored content of the database. Since you must supply the revision information to add an attachment to the document, this serves as validation to update the existing attachment.

Note: Uploading an attachment updates the corresponding document revision. Revisions are tracked for the parent document, not individual attachments.

Parameters

- **db** – Database name
- **docid** – Document ID
- **attname** – Attachment name

Request Headers

- Content-Type – Attachment MIME type. Default: *application/octet-stream* *Optional*
- If-Match – Document revision. Alternative to `rev` query parameter

Query Parameters

- **rev** (*string*) – Document revision. *Optional*

Response JSON Object

- **id** (*string*) – Document ID
- **ok** (*boolean*) – Operation status
- **rev** (*string*) – Revision MVCC token

Status Codes

- 201 Created – Attachment created and stored on disk
- 202 Accepted – Request was accepted, but changes are not yet stored on disk
- 400 Bad Request – Invalid request body or parameters
- 401 Unauthorized – Write privileges required
- 404 Not Found – Specified database, document or attachment was not found
- 409 Conflict – Document's revision wasn't specified or it's not the latest

Request:

```
PUT /recipes/SpaghettiWithMeatballs/recipe.txt HTTP/1.1
Accept: application/json
Content-Length: 86
Content-Type: text/plain
Host: localhost:5984
If-Match: 1-917fa2381192822767f010b95b45325b
```

(continues on next page)

(continued from previous page)

```
1. Cook spaghetti
2. Cook meatballs
3. Mix them
4. Add tomato sauce
5. ...
6. PROFIT!
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 85
Content-Type: application/json
Date: Thu, 15 Aug 2013 12:38:04 GMT
ETag: "2-ce91aed0129be8f9b0f650a2edcfd0a4"
Location: http://localhost:5984/recipes/SpaghettiWithMeatballs/recipe.txt
Server: CouchDB (Erlang/OTP)

{
    "id": "SpaghettiWithMeatballs",
    "ok": true,
    "rev": "2-ce91aed0129be8f9b0f650a2edcfd0a4"
}
```

`DELETE /{db}/{docid}/{attname}`

Deletes the attachment with filename `{attname}` of the specified `doc`. You must supply the `rev` query parameter or If-Match with the current revision to delete the attachment.

Note: Deleting an attachment updates the corresponding document revision. Revisions are tracked for the parent document, not individual attachments.

Parameters

- **db** – Database name
- **docid** – Document ID

Request Headers

- Accept –
  - *application/json*
  - *text/plain*
- If-Match – Document revision. Alternative to `rev` query parameter

Query Parameters

- **rev** (*string*) – Document revision. *Required*
- **batch** (*string*) – Store changes in *batch mode* Possible values: `ok`. *Optional*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- ETag – Double quoted document's new revision

Response JSON Object

- **id** (*string*) – Document ID
- **ok** (*boolean*) – Operation status
- **rev** (*string*) – Revision MVCC token

Status Codes

- 200 OK – Attachment successfully removed
- 202 Accepted – Request was accepted, but changes are not yet stored on disk
- 400 Bad Request – Invalid request body or parameters
- 401 Unauthorized – Write privileges required
- 404 Not Found – Specified database, document or attachment was not found
- 409 Conflict – Document's revision wasn't specified or it's not the latest

Request:

```
DELETE /recipes/SpaghettiWithMeatballs?rev=6-440b2dd39c20413045748b42c6aba6e2␣
↪HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Alternatively, instead of `rev` query parameter you may use If-Match header:

```
DELETE /recipes/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
If-Match: 6-440b2dd39c20413045748b42c6aba6e2
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 85
Content-Type: application/json
Date: Wed, 14 Aug 2013 12:23:13 GMT
ETag: "7-05185cf5fcdf4b6da360af939431d466"
Server: CouchDB (Erlang/OTP)

{
    "id": "SpaghettiWithMeatballs",
    "ok": true,
    "rev": "7-05185cf5fcdf4b6da360af939431d466"
}
```

HTTP Range Requests

HTTP allows you to specify byte ranges for requests. This allows the implementation of resumable downloads and skippable audio and video streams alike. This is available for all attachments inside CouchDB.

This is just a real quick run through how this looks under the hood. Usually, you will have larger binary files to serve from CouchDB, like MP3s and videos, but to make things a little more obvious, I use a text file here (Note that I use the *application/octet-stream* :header`Content-Type` instead of *text/plain*).

```
shell> cat file.txt
My hovercraft is full of eels!
```

Now let's store this text file as an attachment in CouchDB. First, we create a database:

```
shell> curl -X PUT http://adm:pass@127.0.0.1:5984/test
{"ok":true}
```

Then we create a new document and the file attachment in one go:

```
shell> curl -X PUT http://adm:pass@127.0.0.1:5984/test/doc/file.txt \
            -H "Content-Type: application/octet-stream" -d@file.txt
{"ok":true,"id":"doc","rev":"1-287a28fa680ae0c7fb4729bf0c6e0cf2"}
```

Now we can request the whole file easily:

```
shell> curl -X GET http://adm:pass@127.0.0.1:5984/test/doc/file.txt
My hovercraft is full of eels!
```

But say we only want the first 13 bytes:

```
shell> curl -X GET http://adm:pass@127.0.0.1:5984/test/doc/file.txt \
            -H "Range: bytes=0-12"
My hovercraft
```

HTTP supports many ways to specify single and even multiple byte ranges. Read all about it in **RFC 2616#section-14.27**.

Note: Databases that have been created with CouchDB 1.0.2 or earlier will support range requests in 3.3, but they are using a less-optimal algorithm. If you plan to make heavy use of this feature, make sure to compact your database with CouchDB 3.3 to take advantage of a better algorithm to find byte ranges.

12.5 Design Documents

In CouchDB, design documents provide the main interface for building a CouchDB application. The design document defines the views used to extract information from CouchDB through one or more views. Design documents are created within your CouchDB instance in the same way as you create database documents, but the content and definition of the documents is different. Design Documents are named using an ID defined with the design document URL path, and this URL can then be used to access the database contents.

Views and lists operate together to provide automated (and formatted) output from your database.

12.5.1 /{db}/_design/{ddoc}

HEAD /{db}/_design/{ddoc}

Returns the HTTP Headers containing a minimal amount of information about the specified design document.

See also:

HEAD /{db}/{docid}

GET /{db}/_design/{ddoc}

Returns the contents of the design document specified with the name of the design document and from the specified database from the URL. Unless you request a specific revision, the latest revision of the document will always be returned.

See also:

GET /{db}/{docid}

PUT /{db}/_design/{ddoc}

The PUT method creates a new named design document, or creates a new revision of the existing design document.

The design documents have some agreement upon their fields and structure. Currently it is the following:

- **language** (*string*): Defines *Query Server* to process design document functions
- **options** (*object*): View's default options
- **filters** (*object*): *Filter functions* definition
- **lists** (*object*): *List functions* definition. *Deprecated.*
- **rewrites** (*array* or *string*): Rewrite rules definition. *Deprecated.*
- **shows** (*object*): *Show functions* definition. *Deprecated.*
- **updates** (*object*): *Update functions* definition
- **validate_doc_update** (*string*): *Validate document update* function source
- **views** (*object*): *View functions* definition.
- **autoupdate** (*boolean*): Indicates whether to automatically build indexes defined in this design document. Default is `true`.

Note, that for `filters`, `lists`, `shows` and `updates` fields objects are mapping of function name to string function source code. For `views` mapping is the same except that values are objects with `map` and `reduce` (optional) keys which also contains functions source code.

See also:

PUT /{db}/{docid}

DELETE /{db}/_design/{ddoc}

Deletes the specified document from the database. You must supply the current (latest) revision, either by using the `rev` parameter to specify the revision.

See also:

DELETE /{db}/{docid}

COPY /{db}/_design/{ddoc}

The COPY (which is non-standard HTTP) copies an existing design document to a new or existing one.

Given that view indexes on disk are named after their MD5 hash of the view definition, and that a *COPY* operation won't actually change that definition, the copied views won't have to be reconstructed. Both views will be served from the same index on disk.

See also:

COPY /{db}/{docid}

12.5.2 /{db}/_design/{ddoc}/{attname}

HEAD /{db}/_design/{ddoc}/{attname}

Returns the HTTP headers containing a minimal amount of information about the specified attachment.

See also:

HEAD /{db}/{docid}/{attname}

GET /{db}/_design/{ddoc}/{attname}

Returns the file attachment associated with the design document. The raw data of the associated attachment is returned (just as if you were accessing a static file.

See also:

GET /{db}/{docid}/{attname}

PUT /{db}/_design/{ddoc}/{attname}

Uploads the supplied content as an attachment to the specified design document. The attachment name provided must be a URL encoded string.

See also:

PUT /{db}/{docid}/{attname}

DELETE /{db}/_design/{ddoc}/{attname}

Deletes the attachment of the specified design document.

See also:

DELETE /{db}/{docid}/{attname}

12.5.3 /{db}/_design/{ddoc}/_info

GET /{db}/_design/{ddoc}/_info

Obtains information about the specified design document, including the index, index size and current status of the design document and associated index information.

Parameters

- **db** – Database name
- **ddoc** – Design document name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **name** (*string*) – Design document name
- **view_index** (*object*) – *View Index Information*

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /recipes/_design/recipe/_info HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 263
Content-Type: application/json
Date: Sat, 17 Aug 2013 12:54:17 GMT
Server: CouchDB (Erlang/OTP)

{
    "name": "recipe",
    "view_index": {
        "compact_running": false,
        "language": "python",
        "purge_seq": 0,
        "signature": "a59a1bb13fdf8a8a584bc477919c97ac",
        "sizes": {
          "active": 926691,
          "disk": 1982704,
          "external": 1535701
        },
        "update_seq": 12397,
        "updater_running": false,
        "waiting_clients": 0,
        "waiting_commit": false
    }
}
```

View Index Information

The response from *GET /{db}/_design/{ddoc}/_info* contains `view_index` (*object*) field with the next structure:

- **compact_running** (*boolean*): Indicates whether a compaction routine is currently running on the view
- **sizes.active** (*number*): The size of live data inside the view, in bytes
- **sizes.external** (*number*): The uncompressed size of view contents in bytes
- **sizes.file** (*number*): Size in bytes of the view as stored on disk
- **language** (*string*): Language for the defined views
- **purge_seq** (*number*): The purge sequence that has been processed
- **signature** (*string*): MD5 signature of the views for the design document
- **update_seq** (*number / string*): The update sequence of the corresponding database that has been indexed
- **updater_running** (*boolean*): Indicates if the view is currently being updated
- **waiting_clients** (*number*): Number of clients waiting on views from this design document
- **waiting_commit** (*boolean*): Indicates if there are outstanding commits to the underlying database that need to processed

12.5.4 /{db}/_design/{ddoc}/_view/{view}

GET /{db}/_design/{ddoc}/_view/{view}

Executes the specified view function from the specified design document.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **view** – View function name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **conflicts** (*boolean*) – Include *conflicts* information in response. Ignored if `include_docs` isn't `true`. Default is `false`.
- **descending** (*boolean*) – Return the documents in descending order by key. Default is `false`.
- **endkey** (*json*) – Stop returning records when the specified key is reached.
- **end_key** (*json*) – Alias for `endkey` param
- **endkey_docid** (*string*) – Stop returning records when the specified document ID is reached. Ignored if `endkey` is not set.
- **end_key_doc_id** (*string*) – Alias for `endkey_docid`.
- **group** (*boolean*) – Group the results using the reduce function to a group or single row. Implies `reduce` is `true` and the maximum `group_level`. Default is `false`.
- **group_level** (*number*) – Specify the group level to be used. Implies `group` is `true`.
- **include_docs** (*boolean*) – Include the associated document with each row. Default is `false`.
- **attachments** (*boolean*) – Include the Base64-encoded content of *attachments* in the documents that are included if `include_docs` is `true`. Ignored if `include_docs` isn't `true`. Default is `false`.
- **att_encoding_info** (*boolean*) – Include encoding information in attachment stubs if `include_docs` is `true` and the particular attachment is compressed. Ignored if `include_docs` isn't `true`. Default is `false`.
- **inclusive_end** (*boolean*) – Specifies whether the specified end key should be included in the result. Default is `true`.
- **key** (*json*) – Return only documents that match the specified key.
- **keys** (*json-array*) – Return only documents where the key matches one of the keys specified in the array.
- **limit** (*number*) – Limit the number of the returned documents to the specified number.
- **reduce** (*boolean*) – Use the reduction function. Default is `true` when a reduce function is defined.
- **skip** (*number*) – Skip this number of records before starting to return the results. Default is `0`.

- **sorted** (*boolean*) – Sort returned rows (see *Sorting Returned Rows*). Setting this to `false` offers a performance boost. The `total_rows` and `offset` fields are not available when this is set to `false`. Default is `true`.
- **stable** (*boolean*) – Whether or not the view results should be returned from a stable set of shards. Default is `false`.
- **stale** (*string*) – Allow the results from a stale view to be used. Supported values: `ok` and `update_after`. `ok` is equivalent to `stable=true&update=false`. `update_after` is equivalent to `stable=true&update=lazy`. The default behavior is equivalent to `stable=false&update=true`. Note that this parameter is deprecated. Use `stable` and `update` instead. See *Views Generation* for more details.
- **startkey** (*json*) – Return records starting with the specified key.
- **start_key** (*json*) – Alias for `startkey`.
- **startkey_docid** (*string*) – Return records starting with the specified document ID. Ignored if `startkey` is not set.
- **start_key_doc_id** (*string*) – Alias for `startkey_docid` param
- **update** (*string*) – Whether or not the view in question should be updated prior to responding to the user. Supported values: `true`, `false`, `lazy`. Default is `true`.
- **update_seq** (*boolean*) – Whether to include in the response an `update_seq` value indicating the sequence id of the database the view reflects. Default is `false`.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- ETag – Response signature
- Transfer-Encoding – `chunked`

Response JSON Object

- **offset** (*number*) – Offset where the document list started.
- **rows** (*array*) – Array of view row objects. By default the information returned contains only the document ID and revision.
- **total_rows** (*number*) – Number of documents in the database/view.
- **update_seq** (*object*) – Current update sequence for the database.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 404 Not Found – Specified database, design document or view is missed

Request:

```
GET /recipes/_design/ingredients/_view/by_name HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 21 Aug 2013 09:12:06 GMT
ETag: "2FOLSBSW4O6WB798XU4AQYA9B"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "offset": 0,
    "rows": [
        {
            "id": "SpaghettiWithMeatballs",
            "key": "meatballs",
            "value": 1
        },
        {
            "id": "SpaghettiWithMeatballs",
            "key": "spaghetti",
            "value": 1
        },
        {
            "id": "SpaghettiWithMeatballs",
            "key": "tomato sauce",
            "value": 1
        }
    ],
    "total_rows": 3
}
```

Changed in version 1.6.0: added `attachments` and `att_encoding_info` parameters

Changed in version 2.0.0: added `sorted` parameter

Changed in version 2.1.0: added `stable` and `update` parameters

Changed in version 3.3.1: treat single-element `keys` as `key`

> **Warning:** Using the `attachments` parameter to include attachments in view results is not recommended for large attachment sizes. Also note that the Base64-encoding that is used leads to a 33% overhead (i.e. one third) in transfer size for attachments.

POST /{db}/_design/{ddoc}/_view/{view}

Executes the specified view function from the specified design document. POST view functionality supports identical parameters and behavior as specified in the *GET /{db}/_design/{ddoc}/_view/{view}* API but allows for the query string parameters to be supplied as keys in a JSON object in the body of the *POST* request.

Request:

```
POST /recipes/_design/ingredients/_view/by_name HTTP/1.1
Accept: application/json
Content-Length: 37
Host: localhost:5984

{
    "keys": [
        "meatballs",
```

(continues on next page)

(continued from previous page)

```
        "spaghetti"
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 21 Aug 2013 09:14:13 GMT
ETag: "6R5NM8E872JIJF796VF7WI3FZ"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "offset": 0,
    "rows": [
        {
            "id": "SpaghettiWithMeatballs",
            "key": "meatballs",
            "value": 1
        },
        {
            "id": "SpaghettiWithMeatballs",
            "key": "spaghetti",
            "value": 1
        }
    ],
    "total_rows": 3
}
```

View Options

There are two view indexing options that can be defined in a design document as boolean properties of an `options` object. Unlike the others querying options, these aren't URL parameters because they take effect when the view index is generated, not when it's accessed:

- **local_seq** (*boolean*): Makes documents' local sequence numbers available to map functions (as a `_local_seq` document property)
- **include_design** (*boolean*): Allows map functions to be called on design documents as well as regular documents

Querying Views and Indexes

The definition of a view within a design document also creates an index based on the key information defined within each view. The production and use of the index significantly increases the speed of access and searching or selecting documents from the view.

However, the index is not updated when new documents are added or modified in the database. Instead, the index is generated or updated, either when the view is first accessed, or when the view is accessed after a document has been updated. In each case, the index is updated before the view query is executed against the database.

View indexes are updated incrementally in the following situations:

- A new document has been added to the database.
- A document has been deleted from the database.

- A document in the database has been updated.

View indexes are rebuilt entirely when the view definition changes. To achieve this, a `fingerprint` of the view definition is created when the design document is updated. If the fingerprint changes, then the view indexes are entirely rebuilt. This ensures that changes to the view definitions are reflected in the view indexes.

Note: View index rebuilds occur when one view from the same the view group (i.e. all the views defined within a single a design document) has been determined as needing a rebuild. For example, if you have a design document with different views, and you update the database, all three view indexes within the design document will be updated.

Because the view is updated when it has been queried, it can result in a delay in returned information when the view is accessed, especially if there are a large number of documents in the database and the view index does not exist. There are a number of ways to mitigate, but not completely eliminate, these issues. These include:

- Create the view definition (and associated design documents) on your database before allowing insertion or updates to the documents. If this is allowed while the view is being accessed, the index can be updated incrementally.
- Manually force a view request from the database. You can do this either before users are allowed to use the view, or you can access the view manually after documents are added or updated.
- Use the *changes feed* to monitor for changes to the database and then access the view to force the corresponding view index to be updated.

None of these can completely eliminate the need for the indexes to be rebuilt or updated when the view is accessed, but they may lessen the effects on end-users of the index update affecting the user experience.

Another alternative is to allow users to access a 'stale' version of the view index, rather than forcing the index to be updated and displaying the updated results. Using a stale view may not return the latest information, but will return the results of the view query using an existing version of the index.

For example, to access the existing stale view `by_recipe` in the `recipes` design document:

```
http://localhost:5984/recipes/_design/recipes/_view/by_recipe?stale=ok
```

Accessing a stale view:

- Does not trigger a rebuild of the view indexes, even if there have been changes since the last access.
- Returns the current version of the view index, if a current version exists.
- Returns an empty result set if the given view index does not exist.

As an alternative, you use the `update_after` value to the `stale` parameter. This causes the view to be returned as a stale view, but for the update process to be triggered after the view information has been returned to the client.

In addition to using stale views, you can also make use of the `update_seq` query argument. Using this query argument generates the view information including the update sequence of the database from which the view was generated. The returned value can be compared this to the current update sequence exposed in the database information (returned by *GET /{db}*).

Sorting Returned Rows

Each element within the returned array is sorted using native UTF-8 sorting according to the contents of the key portion of the emitted content. The basic order of output is as follows:

- `null`
- `false`
- `true`
- Numbers
- Text (case sensitive, lowercase first)
- Arrays (according to the values of each element, in order)
- Objects (according to the values of keys, in key order)

Request:

```
GET /db/_design/test/_view/sorting HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 21 Aug 2013 10:09:25 GMT
ETag: "8LA1LZPQ37B6R9U8BK9BGQH27"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "offset": 0,
    "rows": [
        {
            "id": "dummy-doc",
            "key": null,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": false,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": true,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": 0,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": 1,
            "value": null
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "id": "dummy-doc",
            "key": 10,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": 42,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": "10",
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": "hello",
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": "Hello",
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": "\u043f\u0440\u0438\u0432\u0435\u0442",
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": [],
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": [
                1,
                2,
                3
            ],
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": [
                2,
                3
            ],
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": [
```

(continues on next page)

(continued from previous page)

```
                3
            ],
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": {},
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": {
                "foo": "bar"
            },
            "value": null
        }
    ],
    "total_rows": 17
}
```

You can reverse the order of the returned view information by using the `descending` query value set to true:

Request:

```
GET /db/_design/test/_view/sorting?descending=true HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 21 Aug 2013 10:09:25 GMT
ETag: "Z4N468R15JBT98OM0AMNSR8U"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "offset": 0,
    "rows": [
        {
            "id": "dummy-doc",
            "key": {
                "foo": "bar"
            },
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": {},
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": [
                3
```

(continues on next page)

(continued from previous page)

```
            ],
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": [
                2,
                3
            ],
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": [
                1,
                2,
                3
            ],
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": [],
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": "\u043f\u0440\u0438\u0432\u0435\u0442",
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": "Hello",
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": "hello",
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": "10",
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": 42,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": 10,
            "value": null
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "id": "dummy-doc",
            "key": 1,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": 0,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": true,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": false,
            "value": null
        },
        {
            "id": "dummy-doc",
            "key": null,
            "value": null
        }
    ],
    "total_rows": 17
}
```

Sorting order and startkey/endkey

The sorting direction is applied before the filtering applied using the `startkey` and `endkey` query arguments. For example the following query will operate correctly when listing all the matching entries between `carrots` and `egg`:

```
GET http://couchdb:5984/recipes/_design/recipes/_view/by_ingredient?startkey="carrots
↪"&endkey="egg" HTTP/1.1
Accept: application/json
```

If the order of output is reversed with the `descending` query argument, the view request will get a 400 Bad Request response:

```
GET /recipes/_design/recipes/_view/by_ingredient?descending=true&startkey="carrots"&
↪endkey="egg" HTTP/1.1
Accept: application/json
Host: localhost:5984

{
    "error": "query_parse_error",
    "reason": "No rows can match your key range, reverse your start_key and end_key␣
↪or set descending=false",
    "ref": 3986383855
}
```

The result will be an error because the entries in the view are reversed before the key filter is applied, so the `endkey` of "egg" will be seen before the `startkey` of "carrots".

Instead, you should reverse the values supplied to the `startkey` and `endkey` parameters to match the descending sorting applied to the keys. Changing the previous example to:

```
GET /recipes/_design/recipes/_view/by_ingredient?descending=true&startkey="egg"&
↪endkey="carrots" HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Using key, keys, start_key and end_key

`key`: Behaves like setting `start_key=$key&end_key=$key`.

`keys`: there are some differences between single-element `keys` and multi-element `keys`. For single-element `keys`, treat it as a `key`.

```
$ curl -X POST http://adm:pass@127.0.0.1:5984/db/_bulk_docs \
       -H 'Content-Type: application/json' \
       -d '{"docs":[{"_id":"a","key":"a","value":1},{"_id":"b","key":"b","value":2},{
↪"_id":"c","key":"c","value":3}]}'
$ curl -X POST http://adm:pass@127.0.0.1:5984/db \
       -H 'Content-Type: application/json' \
       -d '{"_id":"_design/ddoc","views":{"reduce":{"map":"function(doc) { emit(doc.
↪key, doc.value) }","reduce":"_sum"}}}'

$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?key="a"'
{"rows":[{"key":null,"value":1}]}

$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?keys="[\"a\"]"'
{"rows":[{"key":null,"value":1}]}

$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?keys=\["a","b"\]'
{"error":"query_parse_error","reason":"Multi-key fetches for reduce views must use␣
↪`group=true`"}

$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?keys=\["a","c"\]&
↪group=true'
{"rows":[{"key":"a","value":1},{"key":"c","value":3}]}
```

`keys` is incompatible with `key`, `start_key` and `end_key`, but it's possible to use `key` with `start_key` and `end_key`. Different orders of query parameters may result in different responses. Precedence is the order in which query parameters are specified. Usually, the last argument wins.

```
# start_key=a and end_key=b
$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?key="a"&endkey="b"
↪'
{"rows":[{"key":null,"value":3}]}

# start_key=a and end_key=a
$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?endkey="b"&key="a"
↪'
{"rows":[{"key":null,"value":1}]}

# start_key=a and end_key=a
$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?endkey="b"&keys=\[
↪"a"\]'
{"rows":[{"key":null,"value":1}]}
```

(continues on next page)

(continued from previous page)

```
$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?endkey="b"&keys=\[
↪"a","b"\]'
{"error":"query_parse_error","reason":"Multi-key fetches for reduce views must use␣
↪`group=true`"}

$ curl http://adm:pass@127.0.0.1:5984/db/_design/ddoc/_view/reduce'?endkey="b"&keys=\[
↪"a","b"\]&group=true'
{"error":"query_parse_error","reason":"`keys` is incompatible with `key`, `start_key`␣
↪and `end_key`"}
```

Raw collation

By default CouchDB uses an ICU driver for sorting view results. It's possible use binary collation instead for faster view builds where Unicode collation is not important.

To use raw collation add `"options":{"collation":"raw"}` within the view object of the design document. After that, views will be regenerated and new order applied for the appropriate view.

See also:

Views Collation

Using Limits and Skipping Rows

By default, views return all results. That's ok when the number of results is small, but this may lead to problems when there are billions results, since the client may have to read them all and consume all available memory.

But it's possible to reduce output result rows by specifying `limit` query parameter. For example, retrieving the list of recipes using the `by_title` view and limited to 5 returns only 5 records, while there are total 2667 records in view:

Request:

```
GET /recipes/_design/recipes/_view/by_title?limit=5 HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 21 Aug 2013 09:14:13 GMT
ETag: "9Q6Q2GZKPH8D5F8L7PB6DBSS9"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "offset" : 0,
    "rows" : [
        {
            "id" : "3-tiersalmonspinachandavocadoterrine",
            "key" : "3-tier salmon, spinach and avocado terrine",
            "value" : [
                null,
                "3-tier salmon, spinach and avocado terrine"
            ]
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "id" : "Aberffrawcake",
            "key" : "Aberffraw cake",
            "value" : [
                null,
                "Aberffraw cake"
            ]
        },
        {
            "id" : "Adukiandorangecasserole-microwave",
            "key" : "Aduki and orange casserole - microwave",
            "value" : [
                null,
                "Aduki and orange casserole - microwave"
            ]
        },
        {
            "id" : "Aioli-garlicmayonnaise",
            "key" : "Aioli - garlic mayonnaise",
            "value" : [
                null,
                "Aioli - garlic mayonnaise"
            ]
        },
        {
            "id" : "Alabamapeanutchicken",
            "key" : "Alabama peanut chicken",
            "value" : [
                null,
                "Alabama peanut chicken"
            ]
        }
    ],
    "total_rows" : 2667
}
```

To omit some records you may use `skip` query parameter:

Request:

```
GET /recipes/_design/recipes/_view/by_title?limit=3&skip=2 HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 21 Aug 2013 09:14:13 GMT
ETag: "H3G7YZSNIVRRHO5FXPE16NJHN"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "offset" : 2,
```

(continues on next page)

(continued from previous page)

```
    "rows" : [
        {
            "id" : "Adukiandorangecasserole-microwave",
            "key" : "Aduki and orange casserole - microwave",
            "value" : [
                null,
                "Aduki and orange casserole - microwave"
            ]
        },
        {
            "id" : "Aioli-garlicmayonnaise",
            "key" : "Aioli - garlic mayonnaise",
            "value" : [
                null,
                "Aioli - garlic mayonnaise"
            ]
        },
        {
            "id" : "Alabamapeanutchicken",
            "key" : "Alabama peanut chicken",
            "value" : [
                null,
                "Alabama peanut chicken"
            ]
        }
    ],
    "total_rows" : 2667
}
```

Warning: Using `limit` and `skip` parameters is not recommended for results pagination. Read *pagination recipe* why it's so and how to make it better.

Sending multiple queries to a view

New in version 2.2.

POST /{db}/_design/{ddoc}/_view/{view}/queries

Executes multiple specified view queries against the view function from the specified design document.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **view** – View function name

Request Headers

- Content-Type –
  - *application/json*
- Accept –
  - *application/json*

Request JSON Object

- **queries** – An array of query objects with fields for the parameters of each individual view query to be executed. The field names and their meaning are the same as the query parameters of a regular *view request*.

Response Headers

- Content-Type –
  - *application/json*
- ETag – Response signature
- Transfer-Encoding – chunked

Response JSON Object

- **results** (*array*) – An array of result objects - one for each query. Each result object contains the same fields as the response to a regular *view request*.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 404 Not Found – Specified database, design document or view is missing
- 500 Internal Server Error – View function execution error

Request:

```
POST /recipes/_design/recipes/_view/by_title/queries HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: localhost:5984

{
    "queries": [
        {
            "keys": [
                "meatballs",
                "spaghetti"
            ]
        },
        {
            "limit": 3,
            "skip": 2
        }
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 20 Dec 2016 11:17:07 GMT
ETag: "1H8RGBCK3ABY6ACDM7ZSC30QK"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "results" : [
```

(continues on next page)

(continued from previous page)

```
        {
            "offset": 0,
            "rows": [
                {
                    "id": "SpaghettiWithMeatballs",
                    "key": "meatballs",
                    "value": 1
                },
                {
                    "id": "SpaghettiWithMeatballs",
                    "key": "spaghetti",
                    "value": 1
                },
                {
                    "id": "SpaghettiWithMeatballs",
                    "key": "tomato sauce",
                    "value": 1
                }
            ],
            "total_rows": 3
        },
        {
            "offset" : 2,
            "rows" : [
                {
                    "id" : "Adukiandorangecasserole-microwave",
                    "key" : "Aduki and orange casserole - microwave",
                    "value" : [
                        null,
                        "Aduki and orange casserole - microwave"
                    ]
                },
                {
                    "id" : "Aioli-garlicmayonnaise",
                    "key" : "Aioli - garlic mayonnaise",
                    "value" : [
                        null,
                        "Aioli - garlic mayonnaise"
                    ]
                },
                {
                    "id" : "Alabamapeanutchicken",
                    "key" : "Alabama peanut chicken",
                    "value" : [
                        null,
                        "Alabama peanut chicken"
                    ]
                }
            ],
            "total_rows" : 2667
        }
    ]
}
```

12.5.5 /{db}/_design/{ddoc}/_search/{index}

> **Warning:** Search endpoints require a running search plugin connected to each cluster node. See *Search Plugin Installation* for details.

New in version 3.0.

GET /{db}/_design/{ddoc}/_search/{index}

Executes a search request against the named index in the specified design document.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **index** – Search index name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **bookmark** (*string*) – A bookmark received from a previous search. This parameter enables paging through the results. If there are no more results after the bookmark, you get a response with an empty rows array and the same bookmark, confirming the end of the result list.
- **counts** (*json*) – An array of names of string fields for which counts are requested. The response contains counts for each unique value of this field name among the documents that match the search query. *Faceting* must be enabled for this parameter to function.
- **drilldown** (*json*) – This field can be used several times. Each use defines a pair with a field name and a value. The search matches only documents containing the value that was provided in the named field. It differs from using `"fieldname:value"` in the `q` parameter only in that the values are not analyzed. *Faceting* must be enabled for this parameter to function.
- **group_field** (*string*) – Field by which to group search matches. :query number group_limit: Maximum group count. This field can be used only if `group_field` is specified.
- **group_sort** (*json*) – This field defines the order of the groups in a search that uses `group_field`. The default sort order is relevance.
- **highlight_fields** (*json*) – Specifies which fields to highlight. If specified, the result object contains a `highlights` field with an entry for each specified field.
- **highlight_pre_tag** (*string*) – A string that is inserted before the highlighted word in the highlights output.
- **highlight_post_tag** (*string*) – A string that is inserted after the highlighted word in the highlights output.
- **highlight_number** (*number*) – Number of fragments that are returned in highlights. If the search term occurs less often than the number of fragments that are specified, longer fragments are returned.
- **highlight_size** (*number*) – Number of characters in each fragment for highlights.
- **include_docs** (*boolean*) – Include the full content of the documents in the response.

- **include_fields** (*json*) – A JSON array of field names to include in search results. Any fields that are included must be indexed with the store:true option.
- **limit** (*number*) – Limit the number of the returned documents to the specified number. For a grouped search, this parameter limits the number of documents per group.
- **q** (*string*) – Alias for `query`.
- **query** (*string*) – Required. The Lucene query string.
- **ranges** (*json*) – This field defines ranges for faceted, numeric search fields. The value is a JSON object where the fields names are faceted numeric search fields, and the values of the fields are JSON objects. The field names of the JSON objects are names for ranges. The values are strings that describe the range, for example "[0 TO 10]".
- **sort** (*json*) – Specifies the sort order of the results. In a grouped search (when `group_field` is used), this parameter specifies the sort order within a group. The default sort order is relevance. A JSON string of the form `"fieldname<type>"` or `-fieldname<type>` for descending order, where fieldname is the name of a string or number field, and `type` is either a number, a string, or a JSON array of strings. The `type` part is optional, and defaults to number. Some examples are `"foo"`, `"-foo"`, `"bar<string>"`, `"-foo<number>"` and `["-foo<number>", "bar<string>"]`. String fields that are used for sorting must not be analyzed fields. Fields that are used for sorting must be indexed by the same indexer that is used for the search query.
- **stale** (*string*) – Set to `ok` to allow the use of an out-of-date index.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- ETag – Response signature
- Transfer-Encoding – `chunked`

Response JSON Object

- **rows** (*array*) – Array of view row objects. By default the information returned contains only the document ID and revision.
- **total_rows** (*number*) – Number of documents in the database/view.
- **bookmark** (*string*) – Opaque identifier to enable pagination.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 404 Not Found – Specified database, design document or view is missed

Note: You must enable *faceting* before you can use the `counts`, `drilldown`, and `ranges` parameters.

Note: Faceting and grouping are not supported on partitioned searches, so the following query parameters should not be used on those requests: `counts`, `drilldown`, `ranges`, and `group_field`, `group_limit`, group_sort``.

Note: Do not combine the `bookmark` and `stale` options. These options constrain the choice of shard replicas to use for the response. When used together, the options might cause problems when contact is attempted with replicas that are slow or not available.

See also:

For more information about how search works, see the *Search User Guide*.

12.5.6 /{db}/_design/{ddoc}/_search_info/{index}

Warning: Search endpoints require a running search plugin connected to each cluster node. See *Search Plugin Installation* for details.

New in version 3.0.

GET /{db}/_design/{ddoc}/_search_info/{index}

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **index** – Search index name

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Request body is wrong (malformed or missing one of the mandatory fields)
- 500 Internal Server Error – A server error (or other kind of error) occurred

Request:

```
GET /recipes/_design/cookbook/_search_info/ingredients HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "name": "_design/cookbook/ingredients",
    "search_index": {
        "pending_seq": 7125496,
        "doc_del_count": 129180,
        "doc_count": 1066173,
        "disk_size": 728305827,
        "committed_seq": 7125496
    }
}
```

12.5.7 /{db}/_design/{ddoc}/_nouveau/{index}

Warning: Nouveau is an experimental feature. Future releases might change how the endpoints work and might invalidate existing indexes.

Warning: Nouveau endpoints require a running nouveau server. See *Nouveau Server Installation* for details.

New in version 4.0.

GET /{db}/_design/{ddoc}/_nouveau/{index}

Executes a nouveau request against the named index in the specified design document.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **index** – Nouveau index name

Request Headers

- Accept –
  - *application/json*

Query Parameters

- **bookmark** (*string*) – A bookmark received from a previous search. This parameter enables paging through the results. If there are no more results after the bookmark, you get a response with an empty rows array and the same bookmark, confirming the end of the result list.
- **counts** (*json*) – An array of names of string fields for which counts are requested. The response contains counts for each unique value of this field name among the documents that match the search query.
- **include_docs** (*boolean*) – Include the full content of the documents in the response.
- **locale** (*string*) – The (Java) locale used to parse numbers in range queries. Defaults to the JDK default locale if not specified. Some examples are `de` , `us`, `gb`.
- **limit** (*number*) – Limit the number of the returned documents to the specified number. For a grouped search, this parameter limits the number of documents per group.
- **query** (*string*) – Required. The Lucene query string.
- **ranges** (*json*) – This field defines ranges for numeric search fields. The value is a JSON object where the fields names are numeric search fields, and the values of the fields are arrays of JSON objects. The objects must have a `label`, `min` and `max` value (of type string, number, number respectively), and optional `min_inclusive` and `max_inclusive` properties (defaulting to `true` if not specified). Example: `{"bar":[{"label":"cheap","min":0,"max":100}]}`
- **sort** (*json*) – Specifies the sort order of the results. The default sort order is relevance. A JSON string of the form `"fieldname<type>"` or `"-fieldname<type>"` for descending order, where fieldname is the name of a string or number field, and `type` is either `double` or `string`. You can use a single string to sort by one field or an array of strings to sort by several fields in the same order as the array. Some examples are `"relevance"`, `"bar<string>"`, `"-foo<double>"` and `["-foo<double>", "bar<string>"]`.
- **update** (*boolean*) – Set to `false` to allow the use of an out-of-date index.

Response Headers

- Content-Type –
  - *application/json*
- Transfer-Encoding – chunked

Response JSON Object

- **hits** (*array*) – Array of search hits. By default the information returned contains only the document ID and revision.
- **total_hits** (*number*) – Number of matches for the query.
- **total_hits_relation** (*string*) – EQUAL_TO if total_hits is exact. GREATER_THAN_OR_EQUAL_TO if not.
- **bookmark** (*string*) – Opaque identifier to enable pagination.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 404 Not Found – Specified database, design document or view is missed

Note: Faceting is not supported on partitioned searches, so the following query parameters should not be used on those requests: counts and ranges.

See also:

For more information about how nouveau works, see the *Nouveau User Guide*.

12.5.8 /{db}/_design/{ddoc}/_nouveau_info/{index}

Warning: Nouveau is an experimental feature. Future releases might change how the endpoints work and might invalidate existing indexes.

Warning: Nouveau endpoints require a running nouveau server. See *Nouveau Server Installation* for details.

New in version 4.0.

GET /{db}/_design/{ddoc}/_nouveau_info/{index}

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **index** – Search index name

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Request body is wrong (malformed or missing one of the mandatory fields)
- 500 Internal Server Error – A server error (or other kind of error) occurred

Request:

```
GET /recipes/_design/cookbook/_nouveau_info/ingredients HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "name": "_design/cookbook/ingredients",
    "search_index": {
        "num_docs": 1000,
        "update_seq": 5000,
        "disk_size": 1048576
    }
}
```

12.5.9 /{db}/_design/{ddoc}/_show/{func}

> **Warning:** Show functions are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

GET /{db}/_design/{ddoc}/_show/{func}

POST /{db}/_design/{ddoc}/_show/{func}

Applies *show function* for `null` document.

The request and response parameters are depended upon function implementation.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **func** – Show function name

Response Headers

- ETag – Response signature

Query Parameters

- **format** (*string*) – Format of the returned response. Used by *provides()* function

Status Codes

- 200 OK – Request completed successfully
- 500 Internal Server Error – Query server error

Function:

```
function(doc, req) {
    if (!doc) {
        return {body: "no doc"}
    } else {
        return {body: doc.description}
    }
}
```

Request:

```
GET /recipes/_design/recipe/_show/description HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 6
Content-Type: text/html; charset=utf-8
Date: Wed, 21 Aug 2013 12:34:07 GMT
Etag: "7Z2TO7FPEMZ0F4GH0RJCRIOAU"
Server: CouchDB (Erlang/OTP)
Vary: Accept

no doc
```

12.5.10 /{db}/_design/{ddoc}/_show/{func}/{docid}

Warning: Show functions are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

GET /{db}/_design/{ddoc}/_show/{func}/{docid}

POST /{db}/_design/{ddoc}/_show/{func}/{docid}

Applies *show function* for the specified document.

The request and response parameters are depended upon function implementation.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **func** – Show function name
- **docid** – Document ID

Response Headers

- ETag – Response signature

Query Parameters

- **format** (*string*) – Format of the returned response. Used by *provides()* function

Status Codes

- 200 OK – Request completed successfully
- 500 Internal Server Error – Query server error

Function:

```
function(doc, req) {
    if (!doc) {
        return {body: "no doc"}
    } else {
        return {body: doc.description}
    }
}
```

Request:

```
GET /recipes/_design/recipe/_show/description/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 88
Content-Type: text/html; charset=utf-8
Date: Wed, 21 Aug 2013 12:38:08 GMT
Etag: "8IEBO8103EI98HDZL5Z4I1T0C"
Server: CouchDB (Erlang/OTP)
Vary: Accept

An Italian-American dish that usually consists of spaghetti, tomato sauce and␣
↪meatballs.
```

12.5.11 /{db}/_design/{ddoc}/_list/{func}/{view}

> **Warning:** List functions are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

GET /{db}/_design/{ddoc}/_list/{func}/{view}

POST /{db}/_design/{ddoc}/_list/{func}/{view}

Applies *list function* for the *view function* from the same design document.

The request and response parameters are depended upon function implementation.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **func** – List function name
- **view** – View function name

Response Headers

- ETag – Response signature
- Transfer-Encoding – chunked

Query Parameters

- **format** (*string*) – Format of the returned response. Used by *provides()* function

Status Codes

- 200 OK – Request completed successfully
- 500 Internal Server Error – Query server error

Function:

```
function(head, req) {
    var row = getRow();
    if (!row){
        return 'no ingredients'
```

(continues on next page)

(continued from previous page)

```
    }
    send(row.key);
    while(row=getRow()){
        send(', ' + row.key);
    }
}
```

Request:

```
GET /recipes/_design/recipe/_list/ingredients/by_name HTTP/1.1
Accept: text/plain
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Date: Wed, 21 Aug 2013 12:49:15 GMT
Etag: "D52L2M1TKQYDD1Y8MEYJR8C84"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked
Vary: Accept

meatballs, spaghetti, tomato sauce
```

12.5.12 /{db}/_design/{ddoc}/_list/{func}/{other-ddoc}/{view}

> **Warning:** List functions are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

GET /{db}/_design/{ddoc}/_list/{func}/{other-ddoc}/{view}

POST /{db}/_design/{ddoc}/_list/{func}/{other-ddoc}/{view}

Applies *list function* for the *view function* from the other design document.

The request and response parameters are depended upon function implementation.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **func** – List function name
- **other-ddoc** – Other design document name that holds view function
- **view** – View function name

Response Headers

- ETag – Response signature
- Transfer-Encoding – chunked

Query Parameters

- **format** (*string*) – Format of the returned response. Used by *provides()* function

Status Codes

- 200 OK – Request completed successfully

- 500 Internal Server Error – Query server error

Function:

```
function(head, req) {
    var row = getRow();
    if (!row){
        return 'no ingredients'
    }
    send(row.key);
    while(row=getRow()){
        send(', ' + row.key);
    }
}
```

Request:

```
GET /recipes/_design/ingredient/_list/ingredients/recipe/by_ingredient?key=
↪"spaghetti" HTTP/1.1
Accept: text/plain
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Date: Wed, 21 Aug 2013 12:49:15 GMT
Etag: "5L0975X493R0FB5Z3043POZHD"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked
Vary: Accept

spaghetti
```

12.5.13 /{db}/_design/{ddoc}/_update/{func}

POST /{db}/_design/{ddoc}/_update/{func}

Executes *update function* on server side for `null` document.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **func** – Update function name

Response Headers

- *X-Couch-Id* – Created/updated document's ID
- *X-Couch-Update-NewRev* – Created/updated document's revision

Status Codes

- 200 OK – No document was created or updated
- 201 Created – Document was created or updated
- 500 Internal Server Error – Query server error

Function:

```
function(doc, req) {
    if (!doc){
      return [null, {'code': 400,
                     'json': {'error': 'missed',
                              'reason': 'no document to update'}}]
    } else {
        doc.ingredients.push(req.body);
        return [doc, {'json': {'status': 'ok'}}];
    }
}
```

Request:

```
POST /recipes/_design/recipe/_update/ingredients HTTP/1.1
Accept: application/json
Content-Length: 10
Content-Type: application/json
Host: localhost:5984

"something"
```

Response:

```
HTTP/1.1 404 Object Not Found
Cache-Control: must-revalidate
Content-Length: 52
Content-Type: application/json
Date: Wed, 21 Aug 2013 14:00:58 GMT
Server: CouchDB (Erlang/OTP)

{
    "error": "missed",
    "reason": "no document to update"
}
```

12.5.14 /{db}/_design/{ddoc}/_update/{func}/{docid}

PUT /{db}/_design/{ddoc}/_update/{func}/{docid}

Executes *update function* on server side for the specified document.

Parameters

- **db** – Database name
- **ddoc** – Design document name
- **func** – Update function name
- **docid** – Document ID

Response Headers

- *X-Couch-Id* – Created/updated document's ID
- *X-Couch-Update-NewRev* – Created/updated document's revision

Status Codes

- 200 OK – No document was created or updated
- 201 Created – Document was created or updated

- 500 Internal Server Error – Query server error

Function:

```
function(doc, req) {
    if (!doc){
        return [null, {'code': 400,
                       'json': {'error': 'missed',
                                'reason': 'no document to update'}}]
    } else {
        doc.ingredients.push(req.body);
        return [doc, {'json': {'status': 'ok'}}];
    }
}
```

Request:

```
PUT /recipes/_design/recipe/_update/ingredients/SpaghettiWithMeatballs HTTP/1.1
Accept: application/json
Content-Length: 5
Content-Type: application/json
Host: localhost:5984

"love"
```

Response:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 16
Content-Type: application/json
Date: Wed, 21 Aug 2013 14:11:34 GMT
Server: CouchDB (Erlang/OTP)
X-Couch-Id: SpaghettiWithMeatballs
X-Couch-Update-NewRev: 12-a5e099df5720988dae90c8b664496baf

{
    "status": "ok"
}
```

12.5.15 /{db}/_design/{ddoc}/_rewrite/{path}

Warning: Rewrites are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

ANY /{db}/_design/{ddoc}/_rewrite/{path}

Rewrites the specified path by rules defined in the specified design document. The rewrite rules are defined by the `rewrites` field of the design document. The `rewrites` field can either be a *string* containing the a rewrite function or an *array* of rule definitions.

Using a stringified function for `rewrites`

New in version 2.0: When the `rewrites` field is a stringified function, the query server is used to pre-process and route requests.

The function takes a *Request2 object*.

The return value of the function will cause the server to rewrite the request to a new location or immediately return a response.

To rewrite the request, return an object containing the following properties:

- **path** (*string*): Rewritten path.
- **query** (*array*): Rewritten query. If omitted, the original query keys are used.
- **headers** (*object*): Rewritten headers. If omitted, the original request headers are used.
- **method** (*string*): HTTP method of rewritten request (`"GET"`, `"POST"`, etc). If omitted, the original request method is used.
- **body** (*string*): Body for `"POST"`/`"PUT"` requests. If omitted, the original request body is used.

To immediately respond to the request, return an object containing the following properties:

- **code** (*number*): Returned HTTP status code (`200`, `404`, etc).
- **body** (*string*): Body of the response to user.

Example A. Restricting access.

```
function(req2) {
  var path = req2.path.slice(4),
    isWrite = /^(put|post|delete)$/i.test(req2.method),
    isFinance = req2.userCtx.roles.indexOf("finance") > -1;
  if (path[0] == "finance" && isWrite && !isFinance) {
    // Deny writes to  DB "finance" for users
    // having no "finance" role
    return {
      code: 403,
      body: JSON.stringify({
        error: "forbidden".
        reason: "You are not allowed to modify docs in this DB"
      })
    };
  }
  // Pass through all other requests
  return { path: "../../../" + path.join("/") };
}
```

Example B. Different replies for JSON and HTML requests.

```
function(req2) {
  var path = req2.path.slice(4),
    h = headers,
    wantsJson = (h.Accept || "").indexOf("application/json") > -1,
    reply = {};
  if (!wantsJson) {
    // Here we should prepare reply object
    // for plain HTML pages
  } else {
    // Pass through JSON requests
    reply.path = "../../../"+path.join("/");
  }
```

(continues on next page)

(continued from previous page)

```
  return reply;
}
```

Using an array of rules for `rewrites`

When the `rewrites` field is an array of rule objects, the server will rewrite the request based on the first matching rule in the array.

Each rule in the array is an *object* with the following fields:

- **method** (*string*): HTTP request method to bind the request method to the rule. If omitted, uses `"*"`, which matches all methods.
- **from** (*string*): The pattern used to compare against the URL and define dynamic variables.
- **to** (*string*): The path to rewrite the URL to. It can contain variables depending on binding variables discovered during pattern matching and query args (URL args and from the query member).
- **query** (*object*): Query args passed to the rewritten URL. They may contain dynamic variables.

The `to` and `from` paths may contains string patterns with leading `:` or `*` characters to define dynamic variables in the match.

The first rule in the `rewrites` array that matches the incoming request is used to define the rewrite. To match the incoming request, the rule's `method` must match the request's HTTP method and the rule's `from` must match the request's path using the following pattern matching logic.

- The *from* pattern and URL are first split on `/` to get a list of tokens. For example, if *from* field is `/somepath/:var/*` and the URL is `/somepath/a/b/c`, the tokens are `somepath`, `:var`, and `*` for the *from* pattern and `somepath`, `a`, `b`, and `c` for the URL.
- Each token starting with `:` in the pattern will match the corresponding token in the URL and define a new dynamic variable whose name is the remaining string after the `:` and value is the token from the URL. In this example, the `:var` token will match `b` and set `var` = `a`.
- The star token `*` in the pattern will match any number of tokens in the URL and must be the last token in the pattern. It will define a dynamic variable with the remaining tokens. In this example, the `*` token will match the `b` and `c` tokens and set `*` = `b/c`.
- The remaining tokens must match exactly for the pattern to be considered a match. In this example, `somepath` in the pattern matches `somepath` in the URL and all tokens in the URL have matched, causing this rule to be a match.

Once a rule is found, the request URL is rewritten using the `to` and `query` fields. Dynamic variables are substituted into the `:` and `*` variables in these fields to produce the final URL.

If no rule matches, a 404 Not Found response is returned.

Examples:

Rule	URL	Rewrite to	Tokens
{"from": "/a", "to": "/some"}	/a	/some	
{"from": "/a/*", "to": "/some/*}	/a/b/c	/some/b/c	
{"from": "/a/b", "to": "/some"}	/a/b?k=v	/some?k=v	k=v
{"from": "/a/b", "to": "/some/:var"}	/a/b	/some/b?var=b	var=b
{"from": "/a/:foo/", "to": "/some/:foo/"}	/a/b/c	/some/b/c?foo=b	foo=b
{"from": "/a/:foo", "to": "/some", "query": { "k": ":foo" }}	/a/b	/some/?k=b&foo=b	foo=b
{"from": "/a", "to": "/some/:foo"}	/a?foo=b	/some/?b&foo=b	foo=b

Request method, header, query parameters, request payload and response body are dependent on the endpoint to which the URL will be rewritten.

param db
Database name

param ddoc
Design document name

param path
URL path to rewrite

12.6 Partitioned Databases

Partitioned databases allow for data colocation in a cluster, which provides significant performance improvements for queries constrained to a single partition.

See the guide for *getting started with partitioned databases*

12.6.1 /{db}/_partition/{partition_id}

GET /{db}/_partition/{partition_id}

This endpoint returns information describing the provided partition. It includes document and deleted document counts along with external and active data sizes.

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /db/_partition/sensor-260 HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 119
Content-Type: application/json
Date: Thu, 24 Jan 2019 17:19:59 GMT
Server: CouchDB/2.3.0-a1e11cea9 (Erlang OTP/21)

{
  "db_name": "my_new_db",
  "doc_count": 1,
  "doc_del_count": 0,
  "partition": "sensor-260",
  "sizes": {
    "active": 244,
    "external": 347
  }
}
```

12.6.2 /{db}/_partition/{partition_id}/_all_docs

GET /{db}/_partition/{partition_id}/_all_docs

Parameters

- **db** – Database name
- **partition_id** – Partition name

This endpoint is a convenience endpoint for automatically setting bounds on the provided partition range. Similar results can be had by using the global `/db/_all_docs` endpoint with appropriately configured values for `start_key` and `end_key`.

Refer to the *view endpoint* documentation for a complete description of the available query parameters and the format of the returned data.

Request:

```
GET /db/_partition/sensor-260/_all_docs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 10 Aug 2013 16:22:56 GMT
ETag: "1W2DJUZFZSZD9K78UFA3GZWB4"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
  "offset": 0,
  "rows": [
    {
      "id": "sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
      "key": "sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
      "value": {
        "rev": "1-05ed6f7abf84250e213fcb847387f6f5"
      }
    }
  ],
  "total_rows": 1
}
```

12.6.3 /{db}/_partition/{partition_id}/_design/{ddoc}/_view/{view}

GET /{db}/_partition/{partition_id}/_design/{ddoc}/_view/{view}

Parameters

- **db** – Database name
- **partition_id** – Partition name
- **ddoc** – Design document id
- **view** – View name

This endpoint is responsible for executing a partitioned query. The returned view result will only contain rows with the specified partition name.

Refer to the *view endpoint* documentation for a complete description of the available query parameters and the format of the returned data.

```
GET /db/_partition/sensor-260/_design/sensor-readings/_view/by_sensor HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Wed, 21 Aug 2013 09:12:06 GMT
ETag: "2FOLSBSW4O6WB798XU4AQYA9B"
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
  "offset": 0,
  "rows": [
```

(continues on next page)

(continued from previous page)

```
    {
      "id": "sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
      "key": [
        "sensor-260",
        "0"
      ],
      "value": null
    },
    {
      "id": "sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
      "key": [
        "sensor-260",
        "1"
      ],
      "value": null
    },
    {
      "id": "sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
      "key": [
        "sensor-260",
        "2"
      ],
      "value": null
    },
    {
      "id": "sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
      "key": [
        "sensor-260",
        "3"
      ],
      "value": null
    }
  ],
  "total_rows": 4
}
```

12.6.4 /{db}/_partition/{partition_id}/_find

POST /{db}/_partition/{partition_id}/_find

Parameters

- **db** – Database name
- **id** (*partition*) – Name of the partition to query

This endpoint is responsible for finding a partition query by its ID. The returned view result will only contain rows with the specified partition id.

Refer to the *find endpoint* documentation for a complete description of the available parameters and the format of the returned data.

12.6.5 /{db}/_partition/{partition_id}/_explain

POST /{db}/_partition/{partition_id}/_explain

Parameters

- **db** – Database name

Partition id
Name of the partition to query

This endpoint shows which index is being used by the query.

Refer to the *explain endpoint* documentation for a complete description of the available parameters and the format of the returned data.

12.7 Local (non-replicating) Documents

The local (non-replicating) document interface allows you to create local documents that are not replicated to other databases. These documents can be used to hold configuration or other information that is required specifically on the local CouchDB instance.

Local documents have the following limitations:

- Local documents are not replicated to other databases.
- Local documents are not output by views, or the */{db}/_all_docs* view.

From CouchDB 2.0, Local documents can be listed by using the */{db}/_local_docs* endpoint.

Local documents can be used when you want to store configuration or other information for the current (local) instance of a given database.

A list of the available methods and URL paths are provided below:

Method	Path	Description
GET, POST	/{db}/_local_docs	Returns a list of all the non-replicated documents in the database
POST	/{db}/_local_docs/queries	Returns a list of specified non-replicated documents in the database
GET	/{db}/_local/{docid}	Returns the latest revision of the non-replicated document
PUT	/{db}/_local/{docid}	Inserts a new version of the non-replicated document
DELETE	/{db}/_local/{docid}	Deletes the non-replicated document
COPY	/{db}/_local/{docid}	Copies the non-replicated document

12.7.1 /{db}/_local_docs

GET /{db}/_local_docs

Returns a JSON structure of all of the local documents in a given database. The information is returned as a JSON structure containing meta information about the return structure, including a list of all local documents and basic contents, consisting the ID, revision and key. The key is the from the local document's `_id`.

Parameters

- **db** – Database name

Request Headers

- Accept –
  - *application/json*
  - *text/plain*

Query Parameters

- **conflicts** (*boolean*) – Includes *conflicts* information in response. Ignored if `include_docs` isn't `true`. Default is `false`.
- **descending** (*boolean*) – Return the local documents in descending by key order. Default is `false`.
- **endkey** (*string*) – Stop returning records when the specified key is reached. *Optional*.
- **end_key** (*string*) – Alias for `endkey` param.
- **endkey_docid** (*string*) – Stop returning records when the specified local document ID is reached. *Optional*.
- **end_key_doc_id** (*string*) – Alias for `endkey_docid` param.
- **include_docs** (*boolean*) – Include the full content of the local documents in the return. Default is `false`.
- **inclusive_end** (*boolean*) – Specifies whether the specified end key should be included in the result. Default is `true`.
- **key** (*string*) – Return only local documents that match the specified key. *Optional*.
- **keys** (*string*) – Return only local documents that match the specified keys. *Optional*.
- **limit** (*number*) – Limit the number of the returned local documents to the specified number. *Optional*.
- **skip** (*number*) – Skip this number of records before starting to return the results. Default is `0`.
- **startkey** (*string*) – Return records starting with the specified key. *Optional*.
- **start_key** (*string*) – Alias for `startkey` param.
- **startkey_docid** (*string*) – Return records starting with the specified local document ID. *Optional*.
- **start_key_doc_id** (*string*) – Alias for `startkey_docid` param.
- **update_seq** (*boolean*) – Response includes an `update_seq` value indicating which sequence id of the underlying database the view reflects. Default is `false`.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*

Response JSON Object

- **offset** (*number*) – Offset where the local document list started
- **rows** (*array*) – Array of view row objects. By default the information returned contains only the local document ID and revision.
- **total_rows** (*number*) – Number of local documents in the database. Note that this is not the number of rows returned in the actual query.
- **update_seq** (*number*) – Current update sequence for the database

Status Codes

- 200 OK – Request completed successfully

Request:

```
GET /db/_local_docs HTTP/1.1
Accept: application/json
Host: localhost:5984
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Sat, 23 Dec 2017 16:22:56 GMT
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "offset": null,
    "rows": [
        {
            "id": "_local/localdoc01",
            "key": "_local/localdoc01",
            "value": {
                "rev": "0-1"
            }
        },
        {
            "id": "_local/localdoc02",
            "key": "_local/localdoc02",
            "value": {
                "rev": "0-1"
            }
        },
        {
            "id": "_local/localdoc03",
            "key": "_local/localdoc03",
            "value": {
                "rev": "0-1"
            }
        },
        {
            "id": "_local/localdoc04",
            "key": "_local/localdoc04",
            "value": {
                "rev": "0-1"
            }
        },
        {
            "id": "_local/localdoc05",
            "key": "_local/localdoc05",
            "value": {
                "rev": "0-1"
            }
        }
    ],
    "total_rows": null
}
```

POST /{db}/_local_docs

POST *_local_docs* functionality supports identical parameters and behavior as specified in the *GET /{db}/_local_docs* API but allows for the query string parameters to be supplied as keys in a JSON object in the

body of the *POST* request.

Request:

```
POST /db/_local_docs HTTP/1.1
Accept: application/json
Content-Length: 70
Content-Type: application/json
Host: localhost:5984

{
    "keys" : [
        "_local/localdoc02",
        "_local/localdoc05"
    ]
}
```

The returned JSON is the all documents structure, but with only the selected keys in the output:

```
{
    "total_rows" : null,
    "rows" : [
        {
            "value" : {
                "rev" : "0-1"
            },
            "id" : "_local/localdoc02",
            "key" : "_local/localdoc02"
        },
        {
            "value" : {
                "rev" : "0-1"
            },
            "id" : "_local/localdoc05",
            "key" : "_local/localdoc05"
        }
    ],
    "offset" : null
}
```

12.7.2 /{db}/_local_docs/queries

POST /{db}/_local_docs/queries

Querying with specified `keys` will return local documents only. You can also combine `keys` with other query parameters, such as `limit` and `skip`.

Parameters

- **db** – Database name

Request Headers

- Content-Type –
  - *application/json*
- Accept –
  - *application/json*

Request JSON Object

- **queries** – An array of query objects with fields for the parameters of each individual view query to be executed. The field names and their meaning are the same as the query parameters of a regular *_local_docs request*.

Response Headers

- Content-Type –
  - *application/json*
  - *text/plain; charset=utf-8*
- Transfer-Encoding – chunked

Response JSON Object

- **results** (*array*) – An array of result objects - one for each query. Each result object contains the same fields as the response to a regular *_local_docs request*.

Status Codes

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 404 Not Found – Specified database is missing
- 500 Internal Server Error – Query execution error

Request:

```
POST /db/_local_docs/queries HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: localhost:5984

{
    "queries": [
        {
            "keys": [
                "_local/localdoc05",
                "_local/not-exist",
                "_design/recipe",
                "spaghetti"
            ]
        }
    ]
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Type: application/json
Date: Thu, 20 Jul 2023 21:45:37 GMT
Server: CouchDB (Erlang/OTP)
Transfer-Encoding: chunked

{
    "results": [
        {
            "total_rows": null,
            "offset": null,
```

(continues on next page)

(continued from previous page)

```
            "rows": [
                {
                    "id": "_local/localdoc05",
                    "key": "_local/localdoc05",
                    "value": {
                      "rev": "0-1"
                    }
                },
                {
                    "key": "_local/not-exist",
                    "error": "not_found"
                }
            ]
        },
        {
            "total_rows": null,
            "offset": null,
            "rows": [
                {
                  "id": "_local/localdoc04",
                  "key": "_local/localdoc04",
                  "value": {
                      "rev": "0-1"
                    }
                }
            ]
        }
    ]
}
```

Note: Similar to *_design_docs/queries*, /{db}/_local_docs/queries will only return local documents. The difference is `total_rows` and `offset` are always `null`.

12.7.3 /{db}/_local/{docid}

GET /{db}/_local/{docid}

Gets the specified local document. The semantics are identical to accessing a standard document in the specified database, except that the document is not replicated. See *GET /{db}/{docid}*.

PUT /{db}/_local/{docid}

Stores the specified local document. The semantics are identical to storing a standard document in the specified database, except that the document is not replicated. See *PUT /{db}/{docid}*.

DELETE /{db}/_local/{docid}

Deletes the specified local document. The semantics are identical to deleting a standard document in the specified database, except that the document is not replicated. See *DELETE /{db}/{docid}*.

COPY /{db}/_local/{docid}

Copies the specified local document. The semantics are identical to copying a standard document in the specified database, except that the document is not replicated. See *COPY /{db}/{docid}*.

CHAPTER

THIRTEEN

JSON STRUCTURE REFERENCE

The following appendix provides a quick reference to all the JSON structures that you can supply to CouchDB, or get in return to requests.

13.1 All Database Documents

Field	Description
total_rows	Number of documents in the database/view
offset	Offset where the document list started
update_seq (optional)	Current update sequence for the database
rows [array]	Array of document object

13.2 Bulk Document Response

Field	Description
docs [array]	Bulk Docs Returned Documents
id	Document ID
error	Error type
reason	Error string with extended reason

13.3 Bulk Documents

Field	Description
docs [array]	Bulk Documents Document
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)
_deleted (optional)	Whether the document should be deleted

13.4 Changes information for a database

Field	Description
last_seq	Last update sequence
pending	Count of remaining items in the feed
results [array]	Changes made to a database
seq	Update sequence
id	Document ID
changes [array]	List of changes, field-by-field, for this document

13.5 CouchDB Document

Field	Description
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)

13.6 CouchDB Error Status

Field	Description
id	Document ID
error	Error type
reason	Error string with extended reason

13.7 CouchDB database information object

Field	Description
db_name	The name of the database.
committed_update_seq	The number of committed updates.
doc_count	The number of documents in the database.
doc_del_count	The number of deleted documents.
compact_running	Set to true if the database compaction routine is operating on this database.
disk_format_version	The version of the physical format used for the data when it is stored on hard disk.
disk_size	Size in bytes of the data as stored on disk. View indexes are not included in the calculation.
instance_start_time	Timestamp indicating when the database was opened, expressed in microseconds since the epoch.
purge_seq	The number of purge operations on the database.
update_seq	Current update sequence for the database.

13.8 Design Document

Field	Description
_id	Design Document ID
_rev	Design Document Revision
views	View
viewname	View Definition
map	Map Function for View
reduce (optional)	Reduce Function for View

13.9 Design Document Information

Field	Description
name	Name/ID of Design Document
view_index	View Index
com-pact_running	Indicates whether a compaction routine is currently running on the view
disk_size	Size in bytes of the view as stored on disk
language	Language for the defined views
purge_seq	The purge sequence that has been processed
signature	MD5 signature of the views for the design document
update_seq	The update sequence of the corresponding database that has been indexed
updater_running	Indicates if the view is currently being updated
waiting_clients	Number of clients waiting on views from this design document
waiting_commit	Indicates if there are outstanding commits to the underlying database that need to processed

13.10 Document with Attachments

Field	Description
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)
_attachments (optional)	Document Attachment
filename	Attachment information
content_type	MIME Content type string
data	File attachment content, Base64 encoded

13.11 List of Active Tasks

Field	Description
tasks [array]	Active Tasks
pid	Process ID
status	Task status message
task	Task name
type	Operation Type

13.12 Replication Settings

Field	Description
source	Source database name or URL.
target	Target database name or URL.
cancel (optional)	Cancels the replication.
checkpoint_interval (optional)	Specifies the checkpoint interval in ms.
continuous (optional)	Configure the replication to be continuous.
create_target (optional)	Creates the target database.
doc_ids (optional)	Array of document IDs to be synchronized.
filter (optional)	name of the filter function in the form of `ddoc/myfilter`.
source_proxy (optional)	Address of a proxy server through which replication from the source should occur.
target_proxy (optional)	Address of a proxy server through which replication to the target should occur.
query_params (optional)	Query parameter that are passed to the filter function; the value should be a document containing parameters as members.
selector (optional)	Select the documents included in the replication. This option provides performance benefits compared with using the `filter` option.
since_seq (optional)	Sequence from which the replication should start.
use_checkpoints (optional)	Whether to use replication checkpoints or not.
winning_revs_only (optional)	Replicate only the winning revisions.
use_bulk_get (optional)	Try to use `_bulk_get` to fetch revisions.

13.13 Replication Status

Field	Description
ok	Replication status
session_id	Unique session ID
source_last_seq	Last sequence number read from the source database
history [array]	Replication History
session_id	Session ID for this replication operation
recorded_seq	Last recorded sequence number
docs_read	Number of documents read
docs_written	Number of documents written to target
doc_write_failures	Number of document write failures
start_time	Date/Time replication operation started
start_last_seq	First sequence number in changes stream
end_time	Date/Time replication operation completed
end_last_seq	Last sequence number in changes stream
missing_checked	Number of missing documents checked
missing_found	Number of missing documents found
bulk_get_attempts	Number of attempted _bulk_get fetches
bulk_get_docs	Number of documents read with _bulk_get

13.14 Request object

Field	Description
body	Request body data as *string*. If the request method is *GET* this field contains the value `"undefined"`. If the method is *DELETE* or *HEAD* the value is `""` (empty string).
cookie	Cookies *object*.
form	Form data *object*. Contains the decoded body as key-value pairs if the *Content-Type* header was `application/x-www-form-urlencoded`.
headers	Request headers *object*.
id	Requested document id *string* if it was specified or `null` otherwise.
info	*Database information*
method	Request method as *string* or *array*. String value is a method as one of: *HEAD*, *GET*, *POST*, *PUT*, *DELETE*, *OPTIONS*, and *TRACE*. Otherwise it will be represented as an array of char codes.
path	List of requested path sections.
peer	Request source IP address.
query	URL query parameters *object*. Note that multiple keys are not supported and the last key value suppresses others.
requested_p	List of actual requested path section.
raw_path	Raw requested path *string*.
secObj	*Security Object*.
userCtx	*User Context Object*.
uuid	Generated UUID by a specified algorithm in the config file.

```
{
    "body": "undefined",
    "cookie": {
        "AuthSession": "cm9vdDo1MDZBRjQzRjrfcuikzPRfAn-EA37FmjyfM8G8Lw",
        "m": "3234"
    },
    "form": {},
    "headers": {
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
        "Accept-Charset": "ISO-8859-1,utf-8;q=0.7,*;q=0.3",
        "Accept-Encoding": "gzip,deflate,sdch",
        "Accept-Language": "en-US,en;q=0.8",
        "Connection": "keep-alive",
        "Cookie":
↪"m=3234:t|3247:t|6493:t|6967:t|34e2:|18c3:t|2c69:t|5acb:t|ca3:t|c01:t|5e55:t|77cb:t|2a03:t|1d98:t|
↪ AuthSession=cm9vdDo1MDZBRjQzRjrfcuikzPRfAn-EA37FmjyfM8G8Lw",
        "Host": "127.0.0.1:5984",
        "User-Agent": "Mozilla/5.0 (Windows NT 5.2) AppleWebKit/535.7 (KHTML, like␣
↪Gecko) Chrome/16.0.912.75 Safari/535.7"
    },
    "id": "foo",
    "info": {
        "committed_update_seq": 2701412,
        "compact_running": false,
        "db_name": "mailbox",
        "disk_format_version": 6,
        "doc_count": 2262757,
        "doc_del_count": 560,
        "instance_start_time": "1347601025628957",
        "purge_seq": 0,
        "sizes": {
```

(continues on next page)

(continued from previous page)


```
            "active": 7580843252,
            "disk": 14325313673,
            "external": 7803423459
        },
        "update_seq": 2701412
    },
    "method": "GET",
    "path": [
        "mailbox",
        "_design",
        "request",
        "_show",
        "dump",
        "foo"
    ],
    "peer": "127.0.0.1",
    "query": {},
    "raw_path": "/mailbox/_design/request/_show/dump/foo",
    "requested_path": [
        "mailbox",
        "_design",
        "request",
        "_show",
        "dump",
        "foo"
    ],
    "secObj": {
        "admins": {
            "names": [
                "Bob"
            ],
            "roles": []
        },
        "members": {
            "names": [
                "Mike",
                "Alice"
            ],
            "roles": []
        }
    },
    "userCtx": {
        "db": "mailbox",
        "name": "Mike",
        "roles": [
            "user"
        ]
    },
    "uuid": "3184f9d1ea934e1f81a24c71bde5c168"
}
```

13.15 Request2 object

Field	Description
body	Request body data as *string*. If the request method is *GET* this field contains the value `"undefined"`. If the method is *DELETE* or *HEAD* the value is `""` (empty string).
cookie	Cookies *object*.
headers	Request headers *object*.
method	Request method as *string* or *array*. String value is a method as one of: *HEAD*, *GET*, *POST*, *PUT*, *DELETE*, *OPTIONS*, and *TRACE*. Otherwise it will be represented as an array of char codes.
path	List of requested path sections.
peer	Request source IP address.
query	URL query parameters *object*. Note that multiple keys are not supported and the last key value suppresses others.
requested_p	List of actual requested path section.
raw_path	Raw requested path *string*.
secObj	*Security Object*.
userCtx	*User Context Object*.

13.16 Response object

Field	Description
code	HTTP status code *number*.
json	JSON encodable *object*. Implicitly sets *Content-Type* header as `application/json`.
body	Raw response text *string*. Implicitly sets *Content-Type* header as `text/html; charset=utf-8`.
base64	Base64 encoded *string*. Implicitly sets *Content-Type* header as `application/binary`.
headers	Response headers *object*. *Content-Type* header from this object overrides any implicitly assigned one.
stop	*boolean* signal to stop iteration over view result rows (for list functions only)

Warning: The `body`, `base64` and `json` object keys are overlapping each other where the last one wins. Since most realizations of key-value objects do not preserve the key order or if they are mixed, confusing situations can occur. Try to use only one of them.

Note: Any custom property makes CouchDB raise an internal exception. Furthermore, the *Response object* could be a simple string value which would be implicitly wrapped into a `{"body": ...}` object.

13.17 Returned CouchDB Document with Detailed Revision Info

Field	Description
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)
_revs_info [array]	CouchDB document extended revision info
rev	Full revision string
status	Status of the revision

13.18 Returned CouchDB Document with Revision Info

Field	Description
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)
_revisions	CouchDB document revisions
ids [array]	Array of valid revision IDs, in reverse order (latest first)
start	Prefix number for the latest revision

13.19 Returned Document with Attachments

Field	Description
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)
_attachments (optional)	Document attachment
filename	Attachment
stub	Indicates whether the attachment is a stub
content_type	MIME Content type string
length	Length (bytes) of the attachment data
revpos	Revision where this attachment exists

13.20 Security Object

Field	Description
admins	Roles/Users with admin privileges
roles [array]	List of roles with parent privilege
names [array]	List of users with parent privilege
members	Roles/Users with non-admin privileges
roles [array]	List of roles with parent privilege
names [array]	List of users with parent privilege

```
{
    "admins": {
        "names": [
            "Bob"
        ],
        "roles": []
    },
    "members": {
        "names": [
            "Mike",
            "Alice"
        ],
        "roles": []
    }
}
```

13.21 User Context Object

Field	Description
db	Database name in the context of the provided operation.
name	User name.
roles	List of user roles.

```
{
    "db": "mailbox",
    "name": null,
    "roles": [
        "_admin"
    ]
}
```

13.22 View Head Information

Field	Description
total_rows	Number of documents in the view
offset	Offset where the document list started

```
{
    "total_rows": 42,
    "offset": 3
}
```

CHAPTER

FOURTEEN

QUERY SERVER

The *Query server* is an external process that communicates with CouchDB by JSON protocol through stdio interface and processes all *design functions* calls, such as JavaScript *views*.

The default query server is written in *JavaScript*, running via Mozilla SpiderMonkey. You can use *other languages* by setting a Query server key in the `language` property of a design document or the *Content-Type* header of a *temporary view*. Design documents that do not specify a `language` property are assumed to be of type *javascript*.

14.1 Query Server Protocol

A *Query Server* is an external process that communicates with CouchDB via a simple, custom JSON protocol over stdin/stdout. It is used to processes all design functions calls: *views*, *shows*, *lists*, *filters*, *updates* and *validate_doc_update*.

CouchDB communicates with the Query Server process through stdin/stdout with JSON messages that are terminated by a newline character. Messages that are sent to the Query Server are always *array*-typed and follow the pattern `[<command>, <*arguments>]\n`.

Note: In the documentation examples, we omit the trailing `\n` for greater readability. Also, examples contain formatted JSON values while real data is transferred in compact mode without formatting spaces.

14.1.1 `reset`

Command
`reset`

Arguments
Query server state (optional)

Returns
`true`

This resets the state of the Query Server and makes it forget all previous input. If applicable, this is the point to run garbage collection.

CouchDB sends:

```
["reset"]
```

The Query Server answers:

```
true
```

To set up new Query Server state, the second argument is used with object data.

CouchDB sends:

```
["reset", {"reduce_limit": true, "timeout": 5000}]
```

The Query Server answers:

```
true
```

14.1.2 add_lib

Command
add_lib

Arguments
CommonJS library object by `views/lib` path

Returns
`true`

Adds *CommonJS* library to Query Server state for further usage in *map* functions.

CouchDB sends:

```
[
    "add_lib",
    {
        "utils": "exports.MAGIC = 42;"
    }
]
```

The Query Server answers:

```
true
```

Note: This library shouldn't have any side effects nor track its own state or you'll have a lot of happy debugging time if something goes wrong. Remember that a complete index rebuild is a heavy operation and this is the only way to fix mistakes with shared state.

14.1.3 add_fun

Command
add_fun

Arguments
Map function source code.

Returns
`true`

When creating or updating a view, this is how the Query Server is sent the view function for evaluation. The Query Server should parse, compile, and evaluate the function it receives to make it callable later. If this fails, the Query Server returns an error. CouchDB may store multiple functions before sending any documents.

CouchDB sends:

```
[
    "add_fun",
    "function(doc) { if(doc.score > 50) emit(null, {'player_name': doc.name}); }"
]
```

The Query Server answers:

```
true
```

14.1.4 map_doc

Command
map_doc

Arguments
Document object

Returns
Array of key-value pairs per applied *function*

When the view function is stored in the Query Server, CouchDB starts sending all the documents in the database, one at a time. The Query Server calls the previously stored functions one after another with a document and stores its result. When all functions have been called, the result is returned as a JSON string.

CouchDB sends:

```
[
    "map_doc",
    {
        "_id": "8877AFF9789988EE",
        "_rev": "3-235256484",
        "name": "John Smith",
        "score": 60
    }
]
```

If the function above is the only function stored, the Query Server answers:

```
[
    [
        [null, {"player_name": "John Smith"}]
    ]
]
```

That is, an array with the result for every function for the given document.

If a document is to be excluded from the view, the array should be empty.

CouchDB sends:

```
[
    "map_doc",
    {
        "_id": "9590AEB4585637FE",
        "_rev": "1-674684684",
        "name": "Jane Parker",
        "score": 43
    }
]
```

The Query Server answers:

```
[[]]
```

14.1.5 reduce

Command
`reduce`

Arguments

- Reduce function source
- Array of *map function* results where each item represented in format `[[key, id-of-doc], value]`

Returns
Array with pair values: `true` and another array with reduced result

If the view has a reduce function defined, CouchDB will enter into the reduce phase. The Query Server will receive a list of reduce functions and some map results on which it can apply them.

CouchDB sends:

```
[
    "reduce",
    [
        "function(k, v) { return sum(v); }"
    ],
    [
        [[1, "699b524273605d5d3e9d4fd0ff2cb272"], 10],
        [[2, "c081d0f69c13d2ce2050d684c7ba2843"], 20],
        [[null, "foobar"], 3]
    ]
]
```

The Query Server answers:

```
[
    true,
    [33]
]
```

Note that even though the view server receives the map results in the form `[[key, id-of-doc], value]`, the function may receive them in a different form. For example, the JavaScript Query Server applies functions on the list of keys and the list of values.

14.1.6 rereduce

Command
`rereduce`

Arguments

- Reduce function source
- List of values

When building a view, CouchDB will apply the reduce step directly to the output of the map step and the rereduce step to the output of a previous reduce step.

CouchDB will send a list of reduce functions and a list of values, with no keys or document ids to the rereduce step.

CouchDB sends:

```
[
    "rereduce",
    [
        "function(k, v, r) { return sum(v); }"
    ],
    [
        33,
        55,
        66
    ]
]
```

The Query Server answers:

```
[
    true,
    [154]
]
```

14.1.7 ddoc

Command
`ddoc`

Arguments
Array of objects.

- First phase (ddoc initialization):
  - `"new"`
  - Design document `_id`
  - Design document object
- Second phase (design function execution):
  - Design document `_id`
  - Function path as an array of object keys
  - Array of function arguments

Returns

- First phase (ddoc initialization): `true`
- Second phase (design function execution): custom object depending on executed function

This command acts in two phases: *ddoc* registration and *design function* execution.

In the first phase CouchDB sends a full design document content to the Query Server to let it cache it by `_id` value for further function execution.

To do this, CouchDB sends:

```
[
    "ddoc",
    "new",
    "_design/temp",
    {
        "_id": "_design/temp",
        "_rev": "8-d7379de23a751dc2a19e5638a7bbc5cc",
```

(continues on next page)

(continued from previous page)

```
        "language": "javascript",
        "shows": {
            "request": "function(doc,req){ return {json: req}; }",
            "hello": "function(doc,req){ return {body: 'Hello, ' + (doc || {})._id +
↪'!'}; }"
        }
    }
]
```

The Query Server answers:

```
true
```

After this, the design document will be ready to serve subcommands in the second phase.

Note: Each `ddoc` subcommand is the root design document key, so they are not actually subcommands, but first elements of the JSON path that may be handled and processed.

The pattern for subcommand execution is common:

```
["ddoc", <design_doc_id>, [<subcommand>, <funcname>], [<argument1>, <argument2>, ...]]
```

shows

Warning: Show functions are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

Command
`ddoc`

SubCommand
`shows`

Arguments

- Document object or `null` if document *id* isn't specified in request
- *Request object*

Returns
Array with two elements:

- `"resp"`
- *Response object*

Executes *show function*.

Couchdb sends:

```
[
    "ddoc",
    "_design/temp",
    [
        "shows",
        "doc"
    ],
    [
        null,
```

(continues on next page)

(continued from previous page)

```
        {
            "info": {
                "db_name": "test",
                "doc_count": 8,
                "doc_del_count": 0,
                "update_seq": 105,
                "purge_seq": 0,
                "compact_running": false,
                "sizes": {
                  "active": 1535048,
                  "disk": 15818856,
                  "external": 15515850
                },
                "instance_start_time": "1359952188595857",
                "disk_format_version": 6,
                "committed_update_seq": 105
            },
            "id": null,
            "uuid": "169cb4cc82427cc7322cb4463d0021bb",
            "method": "GET",
            "requested_path": [
                "api",
                "_design",
                "temp",
                "_show",
                "request"
            ],
            "path": [
                "api",
                "_design",
                "temp",
                "_show",
                "request"
            ],
            "raw_path": "/api/_design/temp/_show/request",
            "query": {},
            "headers": {
                "Accept": "*/*",
                "Host": "localhost:5984",
                "User-Agent": "curl/7.26.0"
            },
            "body": "undefined",
            "peer": "127.0.0.1",
            "form": {},
            "cookie": {},
            "userCtx": {
                "db": "api",
                "name": null,
                "roles": [
                    "_admin"
                ]
            },
            "secObj": {}
        }
    ]
]
```

The Query Server sends:

```
[
    "resp",
    {
        "body": "Hello, undefined!"
    }
]
```

lists

Warning: List functions are deprecated in CouchDB 3.0, and will be removed in CouchDB 4.0.

Command
ddoc

SubCommand
lists

Arguments

- *View Head Information*:
- *Request object*

Returns
Array. See below for details.

Executes *list function*.

The communication protocol for *list* functions is a bit complex so let's use an example to illustrate.

Assume we have view a function that emits *id-rev* pairs:

```
function(doc) {
    emit(doc._id, doc._rev);
}
```

And we'd like to emulate `_all_docs` JSON response with list function. Our *first* version of the list functions looks like this:

```
function(head, req){
    start({'headers': {'Content-Type': 'application/json'}});
    var resp = head;
    var rows = [];
    while(row=getRow()){
        rows.push(row);
    }
    resp.rows = rows;
    return toJSON(resp);
}
```

The whole communication session during list function execution could be divided on three parts:

1. Initialization

   The first returned object from the list function is an array with the following structure:

   ```
   ["start", <chunks>, <headers>]
   ```

Where <chunks> is an array of text chunks that will be sent to the client and <headers> is an object with response HTTP headers.

This message is sent from the Query Server to CouchDB on the *start()* call which initializes the HTTP response to the client:

```
[
    "start",
    [],
    {
        "headers": {
            "Content-Type": "application/json"
        }
    }
]
```

After this, the list function may start to process view rows.

2. View Processing

   Since view results can be extremely large, it is not wise to pass all its rows in a single command. Instead, CouchDB can send view rows one by one to the Query Server allowing view processing and output generation to be processed as a stream.

   CouchDB sends a special array that carries view row data:

```
[
    "list_row",
    {
        "id": "0cb42c267fe32d4b56b3500bc503e030",
        "key": "0cb42c267fe32d4b56b3500bc503e030",
        "value": "1-967a00dff5e02add41819138abb3284d"
    }
]
```

   If the Query Server has something to return on this, it returns an array with a "chunks" item in the head and an array of data in the tail. For this example it has nothing to return, so the response will be:

```
[
  "chunks",
  []
]
```

   When there are no more view rows to process, CouchDB sends a *list_end* message to signify there is no more data to send:

```
["list_end"]
```

3. Finalization

   The last stage of the communication process is the returning *list tail*: the last data chunk. After this, processing of the list function will be complete and the client will receive a complete response.

   For our example the last message is:

```
[
    "end",
    [
        "{\"total_rows\":2,\"offset\":0,\"rows\":[{\"id\":\
↪"0cb42c267fe32d4b56b3500bc503e030\",\"key\":\"0cb42c267fe32d4b56b3500bc503e030\
↪",\"value\":\"1-967a00dff5e02add41819138abb3284d\"},{\"id\":\
```

(continues on next page)

(continued from previous page)

```
↪"431926a69504bde41851eb3c18a27b1f\",\"key\":\"431926a69504bde41851eb3c18a27b1f\
↪",\"value\":\"1-967a00dff5e02add41819138abb3284d\"}]}"
    ]
]
```

In this example, we have returned our result in a single message from the Query Server. This is okay for small numbers of rows, but for large data sets, perhaps with millions of documents or millions of view rows, this would not be acceptable.

Let's fix our list function and see the changes in communication:

```
function(head, req){
    start({'headers': {'Content-Type': 'application/json'}});
    send('{');
    send('"total_rows":' + toJSON(head.total_rows) + ',');
    send('"offset":' + toJSON(head.offset) + ',');
    send('"rows":[');
    if (row=getRow()){
        send(toJSON(row));
    }
    while(row=getRow()){
        send(',' + toJSON(row));
    }
    send(']');
    return '}';
}
```

"Wait, what?" - you'd like to ask. Yes, we'd build JSON response manually by string chunks, but let's take a look on logs:

```
[Wed, 24 Jul 2013 05:45:30 GMT] [debug] [<0.19191.1>] OS Process #Port<0.4444> Output␣
↪:: ["start",["{","\"total_rows\":2,","\"offset\":0,","\"rows\":[],{"headers":{
↪"Content-Type":"application/json"}}]
[Wed, 24 Jul 2013 05:45:30 GMT] [info] [<0.18963.1>] 127.0.0.1 - - GET /blog/_design/
↪post/_list/index/all_docs 200
[Wed, 24 Jul 2013 05:45:30 GMT] [debug] [<0.19191.1>] OS Process #Port<0.4444> Input ␣
↪:: ["list_row",{"id":"0cb42c267fe32d4b56b3500bc503e030","key":
↪"0cb42c267fe32d4b56b3500bc503e030","value":"1-967a00dff5e02add41819138abb3284d"}]
[Wed, 24 Jul 2013 05:45:30 GMT] [debug] [<0.19191.1>] OS Process #Port<0.4444> Output␣
↪:: ["chunks",["{\"id\":\"0cb42c267fe32d4b56b3500bc503e030\",\"key\":\
↪"0cb42c267fe32d4b56b3500bc503e030\",\"value\":\"1-967a00dff5e02add41819138abb3284d\
↪"}"]]
[Wed, 24 Jul 2013 05:45:30 GMT] [debug] [<0.19191.1>] OS Process #Port<0.4444> Input ␣
↪:: ["list_row",{"id":"431926a69504bde41851eb3c18a27b1f","key":
↪"431926a69504bde41851eb3c18a27b1f","value":"1-967a00dff5e02add41819138abb3284d"}]
[Wed, 24 Jul 2013 05:45:30 GMT] [debug] [<0.19191.1>] OS Process #Port<0.4444> Output␣
↪:: ["chunks",[",{\"id\":\"431926a69504bde41851eb3c18a27b1f\",\"key\":\
↪"431926a69504bde41851eb3c18a27b1f\",\"value\":\"1-967a00dff5e02add41819138abb3284d\
↪"}"]]
[Wed, 24 Jul 2013 05:45:30 GMT] [debug] [<0.19191.1>] OS Process #Port<0.4444> Input ␣
↪:: ["list_end"]
[Wed, 24 Jul 2013 05:45:30 GMT] [debug] [<0.19191.1>] OS Process #Port<0.4444> Output␣
↪:: ["end",["]","}"]]
```

Note, that now the Query Server sends response by lightweight chunks and if our communication process was extremely slow, the client will see how response data appears on their screen. Chunk by chunk, without waiting for the complete result, like they have for our previous list function.

updates

Command
ddoc

SubCommand
updates

Arguments

- Document object or `null` if document *id* wasn't specified in request
- *Request object*

Returns
Array with there elements:

- `"up"`
- Document object or `null` if nothing should be stored
- *Response object*

Executes *update function*.

CouchDB sends:

```
[
    "ddoc",
    "_design/id",
    [
        "updates",
        "nothing"
    ],
    [
        null,
        {
            "info": {
                "db_name": "test",
                "doc_count": 5,
                "doc_del_count": 0,
                "update_seq": 16,
                "purge_seq": 0,
                "compact_running": false,
                "sizes": {
                  "active": 7979745,
                  "disk": 8056936,
                  "external": 8024930
                },
                "instance_start_time": "1374612186131612",
                "disk_format_version": 6,
                "committed_update_seq": 16
            },
            "id": null,
            "uuid": "7b695cb34a03df0316c15ab529002e69",
            "method": "POST",
            "requested_path": [
                "test",
                "_design",
                "1139",
                "_update",
                "nothing"
```

(continues on next page)

(continued from previous page)

```
            ],
            "path": [
                "test",
                "_design",
                "1139",
                "_update",
                "nothing"
            ],
            "raw_path": "/test/_design/1139/_update/nothing",
            "query": {},
            "headers": {
                "Accept": "*/*",
                "Accept-Encoding": "identity, gzip, deflate, compress",
                "Content-Length": "0",
                "Host": "localhost:5984"
            },
            "body": "",
            "peer": "127.0.0.1",
            "form": {},
            "cookie": {},
            "userCtx": {
                "db": "test",
                "name": null,
                "roles": [
                    "_admin"
                ]
            },
            "secObj": {}
        }
    ]
]
```

The Query Server answers:

```
[
    "up",
    null,
    {"body": "document id wasn't provided"}
]
```

or in case of successful update:

```
[
    "up",
    {
        "_id": "7b695cb34a03df0316c15ab529002e69",
        "hello": "world!"
    },
    {"body": "document was updated"}
]
```

filters

Command
ddoc

SubCommand
filters

Arguments

- Array of document objects
- *Request object*

Returns
Array of two elements:

- true
- Array of booleans in the same order of input documents.

Executes *filter function*.

CouchDB sends:

```
[
    "ddoc",
    "_design/test",
    [
        "filters",
        "random"
    ],
    [
        [
            {
                "_id": "431926a69504bde41851eb3c18a27b1f",
                "_rev": "1-967a00dff5e02add41819138abb3284d",
                "_revisions": {
                    "start": 1,
                    "ids": [
                        "967a00dff5e02add41819138abb3284d"
                    ]
                }
            },
            {
                "_id": "0cb42c267fe32d4b56b3500bc503e030",
                "_rev": "1-967a00dff5e02add41819138abb3284d",
                "_revisions": {
                    "start": 1,
                    "ids": [
                        "967a00dff5e02add41819138abb3284d"
                    ]
                }
            }
        ],
        {
            "info": {
                "db_name": "test",
                "doc_count": 5,
                "doc_del_count": 0,
                "update_seq": 19,
                "purge_seq": 0,
```

(continues on next page)

(continued from previous page)

```
                "compact_running": false,
                "sizes": {
                  "active": 7979745,
                  "disk": 8056936,
                  "external": 8024930
                },
                "instance_start_time": "1374612186131612",
                "disk_format_version": 6,
                "committed_update_seq": 19
            },
            "id": null,
            "uuid": "7b695cb34a03df0316c15ab529023a81",
            "method": "GET",
            "requested_path": [
                "test",
                "_changes?filter=test",
                "random"
            ],
            "path": [
                "test",
                "_changes"
            ],
            "raw_path": "/test/_changes?filter=test/random",
            "query": {
                "filter": "test/random"
            },
            "headers": {
                "Accept": "application/json",
                "Accept-Encoding": "identity, gzip, deflate, compress",
                "Content-Length": "0",
                "Content-Type": "application/json; charset=utf-8",
                "Host": "localhost:5984"
            },
            "body": "",
            "peer": "127.0.0.1",
            "form": {},
            "cookie": {},
            "userCtx": {
                "db": "test",
                "name": null,
                "roles": [
                    "_admin"
                ]
            },
            "secObj": {}
        }
    ]
]
```

The Query Server answers:

```
[
    true,
    [
        true,
        false
```

(continues on next page)

(continued from previous page)

```
    ]
]
```

views

Command
ddoc

SubCommand
views

Arguments
Array of document objects

Returns
Array of two elements:

- true
- Array of booleans in the same order of input documents.

New in version 1.2.

Executes *view function* in place of the filter.

Acts in the same way as *filters* command.

validate_doc_update

Command
ddoc

SubCommand
validate_doc_update

Arguments

- Document object that will be stored
- Document object that will be replaced
- *User Context Object*
- *Security Object*

Returns
1

Executes *validation function*.

CouchDB send:

```
[
    "ddoc",
    "_design/id",
    ["validate_doc_update"],
    [
        {
            "_id": "docid",
            "_rev": "2-e0165f450f6c89dc6b071c075dde3c4d",
            "score": 10
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "_id": "docid",
            "_rev": "1-9f798c6ad72a406afdbf470b9eea8375",
            "score": 4
        },
        {
            "name": "Mike",
            "roles": ["player"]
        },
        {
            "admins": {},
            "members": []
        }
    ]
]
```

The Query Server answers:

```
1
```

Note: While the only valid response for this command is `true`, to prevent the document from being saved, the Query Server needs to raise an error: `forbidden` or `unauthorized`; these errors will be turned into correct `HTTP 403` and `HTTP 401` responses respectively.

rewrites

Command
`ddoc`

SubCommand
`rewrites`

Arguments

- *Request2 object*

Returns
`1`

Executes *rewrite function*.

CouchDB send:

```
[
    "ddoc",
    "_design/id",
    ["rewrites"],
    [
        {
            "method": "POST",
            "requested_path": [
                "test",
                "_design",
                "1139",
                "_update",
                "nothing"
            ],
            "path": [
```

(continues on next page)

(continued from previous page)

```
                "test",
                "_design",
                "1139",
                "_update",
                "nothing"
            ],
            "raw_path": "/test/_design/1139/_update/nothing",
            "query": {},
            "headers": {
                "Accept": "*/*",
                "Accept-Encoding": "identity, gzip, deflate, compress",
                "Content-Length": "0",
                "Host": "localhost:5984"
            },
            "body": "",
            "peer": "127.0.0.1",
            "cookie": {},
            "userCtx": {
                "db": "test",
                "name": null,
                "roles": [
                    "_admin"
                ]
            },
            "secObj": {}
        }
    ]
]
```

The Query Server answers:

```
[
    "ok",
    {
        "path": "some/path",
        "query": {"key1": "value1", "key2": "value2"},
        "method": "METHOD",
        "headers": {"Header1": "value1", "Header2": "value2"},
        "body": ""
    }
]
```

or in case of direct response:

```
[
    "ok",
    {
        "headers": {"Content-Type": "text/plain"},
        "body": "Welcome!",
        "code": 200
    }
]
```

or for immediate redirect:

```
[
    "ok",
```

(continues on next page)

(continued from previous page)

```
    {
        "headers": {"Location": "http://example.com/path/"},
        "code": 302
    }
]
```

14.1.8 Returning errors

When something goes wrong, the Query Server can inform CouchDB by sending a special message in response to the received command.

Error messages prevent further command execution and return an error description to CouchDB. Errors are logically divided into two groups:

- *Common errors*. These errors only break the current Query Server command and return the error info to the CouchDB instance *without* terminating the Query Server process.
- *Fatal errors*. Fatal errors signal a condition that cannot be recovered. For instance, if your a design function is unable to import a third party module, it's better to count such error as fatal and terminate whole process.

error

To raise an error, the Query Server should respond with:

```
["error", "error_name", "reason why"]
```

The `"error_name"` helps to classify problems by their type e.g. if it's `"value_error"` to indicate improper data, `"not_found"` to indicate a missing resource and `"type_error"` to indicate an improper data type.

The `"reason why"` explains in human-readable terms what went wrong, and possibly how to resolve it.

For example, calling *Update Functions* against a non-existent document could produce the error message:

```
["error", "not_found", "Update function requires existent document"]
```

forbidden

The *forbidden* error is widely used by *Validate Document Update Functions* to stop further function processing and prevent storage of the new document revision. Since this is not actually an error, but an assertion against user actions, CouchDB doesn't log it at *"error"* level, but returns *HTTP 403 Forbidden* response with error information object.

To raise this error, the Query Server should respond with:

```
{"forbidden": "reason why"}
```

unauthorized

The *unauthorized* error mostly acts like *forbidden* one, but with the meaning of *please authorize first*. This small difference helps end users to understand what they can do to solve the problem. Similar to *forbidden*, CouchDB doesn't log it at *"error"* level, but returns a *HTTP 401 Unauthorized* response with an error information object.

To raise this error, the Query Server should respond with:

```
{"unauthorized": "reason why"}
```

14.1.9 Logging

At any time, the Query Server may send some information that will be saved in CouchDB's log file. This is done by sending a special *log* object with a single argument, on a separate line:

```
["log", "some message"]
```

CouchDB does not respond, but writes the received message to the log file:

```
[Sun, 13 Feb 2009 23:31:30 GMT] [info] [<0.72.0>] Query Server Log Message: some␣
↪message
```

These messages are only logged at `info level`.

14.2 JavaScript

Note: While every design function has access to all JavaScript objects, the table below describes appropriate usage cases. For example, you may use `emit()` in *Map Functions*, but `getRow()` is not permitted during *Map Functions*.

JS Function	Reasonable to use in design doc functions
`emit()`	*Map Functions*
`getRow()`	*List Functions*
`JSON`	any
`isArray()`	any
`log()`	any
`provides()`	*Show Functions*, *List Functions*
`registerType()`	*Show Functions*, *List Functions*
`require()`	any, except *Reduce and Rereduce Functions*
`send()`	*List Functions*
`start()`	*List Functions*
`sum()`	any
`toJSON()`	any

14.2.1 Design functions context

Each design function executes in a special context of predefined objects, modules and functions:

emit(*key*, *value*)

Emits a *key*-*value* pair for further processing by CouchDB after the map function is done.

Arguments

- **key** – The view key
- **value** – The *key*'s associated value

```
function(doc){
    emit(doc._id, doc._rev);
}
```

getRow()

Extracts the next row from a related view result.

Returns
View result row

Return type
object

```
function(head, req){
    send('[');
    row = getRow();
    if (row){
        send(toJSON(row));
        while(row = getRow()){
            send(',');
            send(toJSON(row));
        }
    }
    return ']';
}
```

JSON

JSON2 object.

isArray(*obj*)

A helper function to check if the provided value is an *Array*.

Arguments

- **obj** – Any JavaScript value

Returns
`true` if *obj* is *Array*-typed, `false` otherwise

Return type
boolean

log(*message*)

Log a message to the CouchDB log (at the *INFO* level).

Arguments

- **message** – Message to be logged

```
function(doc){
    log('Procesing doc ' + doc['_id']);
    emit(doc['_id'], null);
}
```

After the map function has run, the following line can be found in CouchDB logs (e.g. at */var/log/couchdb/couch.log*):

```
[Sat, 03 Nov 2012 17:38:02 GMT] [info] [<0.7543.0>] OS Process #Port<0.3289> Log␣
↪:: Processing doc 8d300b86622d67953d102165dbe99467
```

provides(*key*, *func*)

Registers callable handler for specified MIME key.

Arguments

- **key** – MIME key previously defined by *registerType()*
- **func** – MIME type handler

registerType(*key*, **mimes*)

Registers list of MIME types by associated *key*.

Arguments

- **key** – MIME types
- **mimes** – MIME types enumeration

Predefined mappings (*key-array*):

- **all**: `*/*`
- **text**: `text/plain; charset=utf-8`, `txt`
- **html**: `text/html; charset=utf-8`
- **xhtml**: `application/xhtml+xml`, `xhtml`
- **xml**: `application/xml`, `text/xml`, `application/x-xml`
- **js**: `text/javascript`, `application/javascript`, `application/x-javascript`
- **css**: `text/css`
- **ics**: `text/calendar`
- **csv**: `text/csv`
- **rss**: `application/rss+xml`
- **atom**: `application/atom+xml`
- **yaml**: `application/x-yaml`, `text/yaml`
- **multipart_form**: `multipart/form-data`
- **url_encoded_form**: `application/x-www-form-urlencoded`
- **json**: `application/json`, `text/x-json`

require(*path*)

Loads CommonJS module by a specified *path*. The path should not start with a slash.

Arguments

- **path** – A CommonJS module path started from design document root

Returns

Exported statements

send(*chunk*)

Sends a single string *chunk* in response.

Arguments

- **chunk** – Text chunk

```
function(head, req){
    send('Hello,');
    send(' ');
    send('Couch');
    return ;
}
```

start(*init_resp*)

Initiates chunked response. As an option, a custom *response* object may be sent at this point. For *list*-functions only!

Note: list functions may set the *HTTP response code* and *headers* by calling this function. This function must be called before `send()`, `getRow()` or a *return* statement; otherwise, the query server will implicitly call this function with the empty object (`{}`).

```
function(head, req){
    start({
        "code": 302,
        "headers": {
            "Location": "http://couchdb.apache.org"
        }
    });
    return "Relax!";
}
```

sum(*arr*)

Sum *arr*'s items.

Arguments

- **arr** – Array of numbers

Return type
number

toJSON(*obj*)

Encodes *obj* to JSON string. This is an alias for the `JSON.stringify` method.

Arguments

- **obj** – JSON-encodable object

Returns
JSON string

14.2.2 CommonJS Modules

Support for CommonJS Modules (introduced in CouchDB 0.11.0) allows you to create modular design functions without the need for duplication of functionality.

Here's a CommonJS module that checks user permissions:

```
function user_context(userctx, secobj) {
    var is_admin = function() {
        return userctx.indexOf('_admin') != -1;
    }
    return {'is_admin': is_admin}
}

exports['user'] = user_context
```

Each module has access to additional global variables:

- **module** (*object*): Contains information about the stored module
  - **id** (*string*): The module id; a JSON path in ddoc context
  - **current** (*code*): Compiled module code object
  - **parent** (*object*): Parent frame
  - **exports** (*object*): Export statements
- **exports** (*object*): Shortcut to the `module.exports` object

The CommonJS module can be added to a design document, like so:

```
{
    "views": {
        "lib": {
            "security": "function user_context(userctx, secobj) { ... }"
        }
    },
    "validate_doc_update": "function(newdoc, olddoc, userctx, secobj) {
        user = require('views/lib/security').user_context(userctx, secobj);
        return user.is_admin();
    }"
    "_id": "_design/test"
}
```

Modules paths are relative to the design document's `views` object, but modules can only be loaded from the object referenced via `lib`. The `lib` structure can still be used for view functions as well, by simply storing view functions at e.g. `views.lib.map`, `views.lib.reduce`, etc.

14.3 Erlang

Note: The Erlang query server is disabled by default. Read *configuration guide* about reasons why and how to enable it.

Emit(*Id*, *Value*)

Emits *key-value* pairs to view indexer process.

```
fun({Doc}) ->
    <<K,_/binary>> = proplists:get_value(<<"_rev">>, Doc, null),
    V = proplists:get_value(<<"_id">>, Doc, null),
    Emit(<<K>>, V)
end.
```

FoldRows(*Fun*, *Acc*)

Helper to iterate over all rows in a list function.

Arguments

- **Fun** – Function object.
- **Acc** – The value previously returned by *Fun*.

```
fun(Head, {Req}) ->
    Fun = fun({Row}, Acc) ->
        Id = couch_util:get_value(<<"id">>, Row),
        Send(list_to_binary(io_lib:format("Previous doc id: ~p~n", [Acc]))),
        Send(list_to_binary(io_lib:format("Current  doc id: ~p~n", [Id]))),
        {ok, Id}
    end,
    FoldRows(Fun, nil),
    ""
end.
```

GetRow()

Retrieves the next row from a related view result.

```
%% FoldRows background implementation.
%% https://git-wip-us.apache.org/repos/asf?p=couchdb.git;a=blob;f=src/couchdb/
↪couch_native_process.erl;hb=HEAD#l368
%%
foldrows(GetRow, ProcRow, Acc) ->
    case GetRow() of
        nil ->
            {ok, Acc};
        Row ->
            case (catch ProcRow(Row, Acc)) of
                {ok, Acc2} ->
                    foldrows(GetRow, ProcRow, Acc2);
                {stop, Acc2} ->
                    {ok, Acc2}
            end
end.
```

Log(*Msg*)

Arguments

- **Msg** – Log a message at the *INFO* level.

```
fun({Doc}) ->
    <<K,_/binary>> = proplists:get_value(<<"_rev">>, Doc, null),
    V = proplists:get_value(<<"_id">>, Doc, null),
    Log(lists:flatten(io_lib:format("Hello from ~s doc!", [V]))),
    Emit(<<K>>, V)
end.
```

After the map function has run, the following line can be found in CouchDB logs (e.g. at */var/log/couchdb/couch.log*):

```
[Sun, 04 Nov 2012 11:33:58 GMT] [info] [<0.9144.2>] Hello from␣
↪8d300b86622d67953d102165dbe99467 doc!
```

Send(*Chunk*)

Sends a single string *Chunk* in response.

```
fun(Head, {Req}) ->
    Send("Hello,"),
    Send(" "),
    Send("Couch"),
    "!"
end.
```

The function above produces the following response:

```
Hello, Couch!
```

Start(*Headers*)

Arguments

- **Headers** – Proplist of *response object*.

Initialize *List Functions* response. At this point, response code and headers may be defined. For example, this function redirects to the CouchDB web site:

```
fun(Head, {Req}) ->
    Start({[{<<"code">>, 302},
            {<<"headers">>, {[
                {<<"Location">>, <<"http://couchdb.apache.org">>}]
            }}
        ]}),
    "Relax!"
end.
```

CHAPTER

FIFTEEN

PARTITIONED DATABASES

A partitioned database forms documents into logical partitions by using a partition key. All documents are assigned to a partition, and many documents are typically given the same partition key. The benefit of partitioned databases is that secondary indices can be significantly more efficient when locating matching documents since their entries are contained within their partition. This means a given secondary index read will only scan a single partition range instead of having to read from a copy of every shard.

As a means to introducing partitioned databases, we'll consider a motivating use case to describe the benefits of this feature. For this example, we'll consider a database that stores readings from a large network of soil moisture sensors.

Note: Before reading this document you should be familiar with the *theory* of *sharding* in CouchDB.

Traditionally, a document in this database may have something like the following structure:

```
{
    "_id": "sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
    "_rev":"1-14e8f3262b42498dbd5c672c9d461ff0",
    "sensor_id": "sensor-260",
    "location": [41.6171031, -93.7705674],
    "field_name": "Bob's Corn Field #5",
    "readings": [
        ["2019-01-21T00:00:00", 0.15],
        ["2019-01-21T06:00:00", 0.14],
        ["2019-01-21T12:00:00", 0.16],
        ["2019-01-21T18:00:00", 0.11]
    ]
}
```

Note: While this example uses IoT sensors, the main thing to consider is that there is a logical grouping of documents. Similar use cases might be documents grouped by user or scientific data grouped by experiment.

So we've got a bunch of sensors, all grouped by the field they monitor along with their readouts for a given day (or other appropriate time period).

Along with our documents, we might expect to have two secondary indexes for querying our database that might look something like:

```
function(doc) {
    if(doc._id.indexOf("sensor-reading-") != 0) {
        return;
    }
    for(var r in doc.readings) {
        emit([doc.sensor_id, r[0]], r[1])
```

(continues on next page)

(continued from previous page)

```
    }
}
```

and:

```
function(doc) {
    if(doc._id.indexOf("sensor-reading-") != 0) {
        return;
    }
    emit(doc.field_name, doc.sensor_id)
}
```

With these two indexes defined, we can easily find all readings for a given sensor, or list all sensors in a given field.

Unfortunately, in CouchDB, when we read from either of these indexes, it requires finding a copy of every shard and asking for any documents related to the particular sensor or field. This means that as our database scales up the number of shards, every index request must perform more work, which is unnecessary since we are only interested in a small number of documents. Fortunately for you, dear reader, partitioned databases were created to solve this precise problem.

15.1 What is a partition?

In the previous section, we introduced a hypothetical database that contains sensor readings from an IoT field monitoring service. In this particular use case, it's quite logical to group all documents by their `sensor_id` field. In this case, we would call the `sensor_id` the partition key.

A good partition has two basic properties. First, it should have a high cardinality. That is, a large partitioned database should have many more partitions than documents in any single partition. A database that has a single partition would be an anti-pattern for this feature. Secondly, the amount of data per partition should be "small". The general recommendation is to limit individual partitions to less than ten gigabytes (10 GB) of data. Which, for the example of sensor documents, equates to roughly 60,000 years of data.

Note: The `max_partition_size` under CouchDB dictates the partition limit. The default value for this option is 10GiB but can be changed accordingly. Setting the value for this option to 0 disables the partition limit.

15.2 Why use partitions?

The primary benefit of using partitioned databases is for the performance of partitioned queries. Large databases with lots of documents often have a similar pattern where there are groups of related documents that are queried together.

By using partitions, we can execute queries against these individual groups of documents more efficiently by placing the entire group within a specific shard on disk. Thus, the view engine only has to consult one copy of the given shard range when executing a query instead of executing the query across all `q` shards in the database. This mean that you do not have to wait for all `q` shards to respond, which is both efficient and faster.

15.3 Partitions By Example

To create a partitioned database, we simply need to pass a query string parameter:

```
shell> curl -X PUT 'http://adm:pass@127.0.0.1:5984/my_new_db?partitioned=true'
{"ok":true}
```

To see that our database is partitioned, we can look at the database information:

```
shell> curl http://adm:pass@127.0.0.1:5984/my_new_db
{
  "cluster": {
    "n": 3,
    "q": 8,
    "r": 2,
    "w": 2
  },
  "compact_running": false,
  "db_name": "my_new_db",
  "disk_format_version": 7,
  "doc_count": 0,
  "doc_del_count": 0,
  "instance_start_time": "0",
  "props": {
    "partitioned": true
  },
  "purge_seq": "0-g1AAAAFDeJzLYWBg4M...",
  "sizes": {
    "active": 0,
    "external": 0,
    "file": 66784
  },
  "update_seq": "0-g1AAAAFDeJzLYWBg4M..."
}
```

You'll now see that the `"props"` member contains `"partitioned": true`.

Note: Every document in a partitioned database (except _design and _local documents) must have the format "partition:docid". More specifically, the partition for a given document is everything before the first colon. The document id is everything after the first colon, which may include more colons.

Note: System databases (such as _users) are *not* allowed to be partitioned. This is due to system databases already having their own incompatible requirements on document ids.

Now that we've created a partitioned database, it's time to add some documents. Using our earlier example, we could do this as such:

```
shell> cat doc.json
{
    "_id": "sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
    "sensor_id": "sensor-260",
    "location": [41.6171031, -93.7705674],
    "field_name": "Bob's Corn Field #5",
    "readings": [
        ["2019-01-21T00:00:00", 0.15],
```

(continues on next page)

(continued from previous page)

```
        ["2019-01-21T06:00:00", 0.14],
        ["2019-01-21T12:00:00", 0.16],
        ["2019-01-21T18:00:00", 0.11]
    ]
}
shell> $ curl -X POST -H "Content-Type: application/json" \
            http://adm:pass@127.0.0.1:5984/my_new_db -d @doc.json
{
    "ok": true,
    "id": "sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
    "rev": "1-05ed6f7abf84250e213fcb847387f6f5"
}
```

The only change required to the first example document is that we are now including the partition name in the document id by prepending it to the old id separated by a colon.

Note: The partition name in the document id is not magical. Internally, the database is simply using only the partition for hashing the document to a given shard, instead of the entire document id.

Working with documents in a partitioned database is no different than a non-partitioned database. All APIs are available, and existing client code will all work seamlessly.

Now that we have created a document, we can get some info about the partition containing the document:

```
shell> curl http://adm:pass@127.0.0.1:5984/my_new_db/_partition/sensor-260
{
  "db_name": "my_new_db",
  "doc_count": 1,
  "doc_del_count": 0,
  "partition": "sensor-260",
  "sizes": {
    "active": 244,
    "external": 347
  }
}
```

And we can also list all documents in a partition:

```
shell> curl http://adm:pass@127.0.0.1:5984/my_new_db/_partition/sensor-260/_all_docs
{"total_rows": 1, "offset": 0, "rows":[
    {
        "id":"sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
        "key":"sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf",
        "value": {"rev": "1-05ed6f7abf84250e213fcb847387f6f5"}
    }
]}
```

Note that we can use all of the normal bells and whistles available to `_all_docs` requests. Accessing `_all_docs` through the `/dbname/_partition/name/_all_docs` endpoint is mostly a convenience so that requests are guaranteed to be scoped to a given partition. Users are free to use the normal `/dbname/_all_docs` to read documents from multiple partitions. Both query styles have the same performance.

Next, we'll create a design document containing our index for getting all readings from a given sensor. The map function is similar to our earlier example except we've accounted for the change in the document id.

```
function(doc) {
    if(doc._id.indexOf(":sensor-reading-") < 0) {
```

(continues on next page)

(continued from previous page)

```
        return;
    }
    for(var r in doc.readings) {
        emit([doc.sensor_id, r[0]], r[1])
    }
}
```

After uploading our design document, we can try out a partitioned query:

```
shell> cat ddoc.json
{
    "_id": "_design/sensor-readings",
    "views": {
        "by_sensor": {
            "map": "function(doc) { ... }"
        }
    }
}
shell> $ curl -X POST -H "Content-Type: application/json" http://adm:pass@127.0.0.
↪1:5984/my_new_db -d @ddoc.json
{
    "ok": true,
    "id": "_design/sensor-readings",
    "rev": "1-13859808da293bd72fde3b31be97372a"
}
shell> curl http://adm:pass@127.0.0.1:5984/my_new_db/_partition/sensor-260/_design/
↪sensor-readings/_view/by_sensor
{"total_rows":4,"offset":0,"rows":[
{"id":"sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf","key":["sensor-
↪260","0"],"value":null},
{"id":"sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf","key":["sensor-
↪260","1"],"value":null},
{"id":"sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf","key":["sensor-
↪260","2"],"value":null},
{"id":"sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf","key":["sensor-
↪260","3"],"value":null}
]}
```

Hooray! Our first partitioned query. For experienced users, that may not be the most exciting development, given that the only things that have changed are a slight tweak to the document id, and accessing views with a slightly different path. However, for anyone who likes performance improvements, it's actually a big deal. By knowing that the view results are all located within the provided partition name, our partitioned queries now perform nearly as fast as document lookups!

The last thing we'll look at is how to query data across multiple partitions. For that, we'll implement the example sensors by field query from our initial example. The map function will use the same update to account for the new document id format, but is otherwise identical to the previous version:

```
function(doc) {
    if(doc._id.indexOf(":sensor-reading-") < 0) {
        return;
    }
    emit(doc.field_name, doc.sensor_id)
}
```

Next, we'll create a new design doc with this function. Be sure to notice that the `"options"` member contains `"partitioned": false`.

```
shell> cat ddoc2.json
{
  "_id": "_design/all_sensors",
  "options": {
    "partitioned": false
  },
  "views": {
    "by_field": {
      "map": "function(doc) { ... }"
    }
  }
}
shell> $ curl -X POST -H "Content-Type: application/json" http://adm:pass@127.0.0.
↪1:5984/my_new_db -d @ddoc2.json
{
    "ok": true,
    "id": "_design/all_sensors",
    "rev": "1-4a8188d80fab277fccf57bdd7154dec1"
}
```

Note: Design documents in a partitioned database default to being partitioned. Design documents that contain views for queries across multiple partitions must contain the `"partitioned": false` member in the `"options"` object.

Note: Design documents are either partitioned or global. They cannot contain a mix of partitioned and global indexes.

And to see a request showing us all sensors in a field, we would use a request like:

```
shell> curl -u adm:pass http://adm:pass@127.0.0.1:15984/my_new_db/_design/all_sensors/
↪_view/by_field
{"total_rows":1,"offset":0,"rows":[
{"id":"sensor-260:sensor-reading-ca33c748-2d2c-4ed1-8abf-1bca4d9d03cf","key":"Bob's␣
↪Corn Field #5","value":"sensor-260"}
]}
```

Notice that we're not using the `/dbname/_partition/...` path for global queries. This is because global queries, by definition, do not cover individual partitions. Other than having the `"partitioned": false` parameter in the design document, global design documents and queries are identical in behavior to design documents on non-partitioned databases.

Warning: To be clear, this means that global queries perform identically to queries on non-partitioned databases. Only partitioned queries on a partitioned database benefit from the performance improvements.

CHAPTER

SIXTEEN

RELEASE NOTES

16.1 3.3.x Branch

- *Version 3.3.3*
- *Version 3.3.2*
- *Version 3.3.1*
- *Version 3.3.0*

16.1.1 Version 3.3.3

Features and Enhancements

- #4623: Handle replicator instance start time during upgrades better.
- #4653: Fix the ability to use `;` in config values.
- #4626: Fix purge infos replicating to the wrong shards during shard splitting.
- #4669: Make it possible to override `[replicator] valid_socket_options` with more than two items.
- #4670: Allow setting of some ibrowse replication options such as `{prefer_ipv6, true}`.
- #4679: Fix multipart parser "attachment longer than expected" error. Previously there was a small chance attachments were not replicated.
- #4680: Allow restarting failed shard splitting jobs.
- #4722: Fix badmatch error when purge requests time out.
- #4736: Stop client process and clean up if client disconnects when processing streaming requests.
- #4758: Remove sensitive headers from the mochiweb request in process dictionary. This should prevent sensitive headers leaking into logs when a request process crashes.
- #4784: Extract the correct node name from `ERL_FLAGS` in `remsh`.
- #4794: Fix incorrect raising of `database_does_not_exist` error.
- #4821: Wait for compacted indexes to flip. Previously, a timeout during compact file flips could lead to crashes and a subsequent recompaction.
- #4837: Fix update bug in ets_lru.
- #4847: Require auth for `_replicate` endpoint.
- #4878: Add an option to scrub some sensitive headers from external json requests.

16.1.2 Version 3.3.2

Features and Enhancements

- #4529: In Javascript process manager, use a database tag in addition to a ddoc ID to quickly find processes. This should improve performance.
- #4509, #4405: Make remsh work with quoted cookie values.
- #4473: Avoid re-compiling filter view functions. This could speed up Javascript filter functions.
- #4412: Remove Javascript json2 script and the try/except clause around seal.
- #4513: Allow configurable timeouts for `_view` and `_search`. Search timeouts can be specified as `[fabric] search_timeout` and `[fabric] search_permsg`. View per-message timeout can be configured as `[fabric] view_permsg_timeout`.
- #4438: Proxy auth can now use one of the configured hash algorithms from chttpd_auth/hash_algorithms to decode authentication tokens.
- #4370: Ensure design docs are uploaded individually when replicating with `_bulk_get`. This restores previous behavior before version 3.3.0.
- #4416: Allow `_local` doc writes to the replicator dbs. Previously this issue prevented replicating the replicator db itself, since checkpointing was not working properly.
- #4363: Fix replication `_scheduler/docs` `"total_rows"` value.
- #4380: Be more defensive about SpiderMonkey location. An error should be emitted early if the Spidermonkey library cannot be found.
- #4388: Bump recon to 2.5.3. See the changelog for more details.
- #4476, #4515, #4490, #4350, #4379: Various documentation cleanups and fixes.
- Fix for CVE-2023-26268.

16.1.3 Version 3.3.1

Features and Enhancements

- #4343, #4344, #4345: Fix `undef` when parsing replication doc body with a `user_ctx`.
- #4346: Add `make` target to find `undef` errors.
- #4347: Remove failed `couch_plugins` experiment, fixes more `undef` errors.
- #4348: Fix `undef` error in `weatherreport`.
- #4353: Allow starting of more than one replication job. (D'OH!)

16.1.4 Version 3.3.0

Highlights

- #4308: Replicator was optimized and should be faster. It now uses the *_bulk_get* endpoint on the source, and can statistically skip calling *_revs_diff* on the target. Benchmark tests replicating 1M documents, 10KB each, from UK to US East show a 3x speed improvement.

Features and Enhancements

- #3766, #3970, #3972, #4093, #4102, #4104, #4110, #4111, #4114, #4245, #4246:, #4266: Add `smoosh` queue persistence. This allows resuming `smoosh` operations after a node restart. This is disabled by default and can be enabled with `[smoosh] persist = true`. Optimise `smoosh` operations and increase test coverage to 90%.
- #3798: Add `libicu` version and collation algorithm version to `/_node/{node-name}/_versions`.
- #3837: The Erlang source tree is now auto-formatted with `erlfmt`.
- #3845: Clean up the `couch_ejson_compare` C-module and squash Microsoft compiler warnings.
- #3832: Add `GET` variant to `_dbs_info` endpoint, used to be `POST` only.
- #3864: Improve `erlang_ls` configuration.
- #3853: Remove legacy `ddoc_cache_opener` `gen_server` and speed up event routing.
- #3879: Remove use of `ERL_OPTS` environment variable. All supported Erlang versions now use `ERL_COMPILER_OPTIONS` for the same purpose.
- #3883: Add support for SpiderMonkey 91.
- #3889: Track `libicu` collator versions in the view header.
- #3952: Make the timeout for receiving requests from attachment writers configurable.
- #3927: Include index signature in `_search_info`.
- #3963: Optimtize key tree stemming by using maps instead of sets. This greatly reduced memory usage for heavily conflicted docs in some situations.
- #3974: Create new config options in `[couchdb]` and `[smoosh]` sections to enable finer control of compaction logging levels.
- #3983, #3984, #3985, #3987, #4033: Add various functions to `couch_debug` module.
- #4000: Ensure `Object.prototype.toSource()` is always available.
- #4018: Update `jiffy` to 1.1.1 and `b64url` to 1.0.3.
- #4021: Reduce smoosh compaction log level to `debug`.
- #4041: Allow and evaluate nested json claim roles in JWT token.

- #4060, #4290: Add support for Erlang 25.
- #4064: Enable replicating purge requests between nodes. Also avoid applying interactive purges more than once.
- #4069, #4084: Drop support for Erlang < 23, update `vm.args` settings to match. Review this if you have customized your `vm.args`.
- #4083: Support Elixir 13.
- #4085: Add an option to let `custodian` always use `[cluster] n` value.
- #4095: Implement `winning_revs_only` option for the replicator. It replicates only the winning revisions from the source to the target, effectively discarding conflicts.
- #4135: Separate search IO from file IO.
- #4140, #4162: Upgrade hash algorithm for cookie auth (sha1 -> sha256). This introduces a new config setting `hash_algorithms`. New cookie values are hashed with sha256, sha1 hashes are still accepted. Admins can set this to sha256 only. Sha1 will be disallowed in the next major release. Show supported hash algorithms in `/_node/{node-name}/_versions` endpoint.
- #4179: Don't double-encode changes sequence strings in the replicator.
- #4182: Explicitly maintain a fully connected cluster. Previously, it was possible for the nodes to disconnect, and for that state to persist until the nodes restarted.
- #4198: Redact passwords in log file.
- #4243: Update `mochiweb` to 3.1.1.
- #4254: The `_dbs_info` access control is now configured with the `[couchdb] admin_only_all_dbs` setting. Defaults to true. This was a leftover from the 3.0.0 release.
- #4264: `active` database sizes is now limited to leaf nodes. Previously, it included intermediate tree nodes, which had the effect that deleting (large) documents did not decrease `active` database size. In addition, `smoosh` now picks up databases where large documents are deleted for compaction more eagerly, reclaiming the deleted space quicker.
- #4270: Shard splitting now uses its own `reshard` IO priority. It can be configured to be safely run in the background with production loads, or with maximum IO available, if admins prefer quicker progress.
- #4274: Improve validation of replicator job parameters & move `_replicator` VDU design doc to internal BDU.
- #4280: Add `CFLAGS` and `LDFLAGS` to ICU build parameters.
- #4284: Remove all usage of global to avoid potential deadlocks in replication jobs.
- #4287: Allow = in config key names.
- #4306: Fauxton was updated to version v1.2.9. Changes since v1.2.8 can be found here
- #4317: Write "Relax" welcome message to standard out on Windows.

Performance

- #3860: Add sharding to `couch_index_server`, similar to #3366, avoids processing bottlenecks on servers with a lot of concurrent view indexing going on.
- #3891: Avoid decoding JWT payloads when not necessary.
- #4031: Default `[rexi] use_kill_all` to `true`. This improves intra-cluster-node messaging. Set to false if you run a cluster with nodes that have a version <3.0.0.
- #4052: Optimise `couch_util:reorder_results/2,3`, which speeds up `_bulk_docs` and `_revs_diff`.
- #4055: Avoid using `length/1` guard for `>0` or `==0` tests in `couch_key_tree`.
- #4056: Optimise `couch_key_tree:find_missing/2`. This speeds up `_revs_diff`.

- #4059: Reduce complexity of `possible_ancestors` from quadratic to linear. This speeds up working with heavily conflicted documents significantly.
- #4091: Optimise `couch_util:to_hex/1`, this speeds up all operations that need to encode a revision id into JSON (this is most operations).
- #4106: Set `io_priority` in all IO paths. Introduces `system io_priority`.
- #4144, #4172: Implement `_bulk_get` support for the replicator. Backward compatibility is ensured. This speeds up all replications. Add option to disable new behaviour for legacy setups.
- #4163: Statistically skip `_revs_diff` in the replicator. This improves performance for replications into empty targets.
- #4177: Remove the long deprecated `bigcouch 0.4` change sequence support.
- #4238: Optimise `_bulk_get` endpoint. This speeds up replication of 1M docs by ~2x. Individual `_bulk_get` requests are up to 8x faster.
- #3517: Add experimental fix for reduce performance regression due to expensive repeated AST-transformations on newer SpiderMonkey versions. Set `COUCHDB_QUERY_SERVER_JAVASCRIPT` env var to `COUCHDB_QUERY_SERVER_JAVASCRIPT="/opt/couchdb/bin/couchjs /opt/couchdb/share/server/main-ast-bypass.js"`.
- #4262: `couchjs` executable built against Spidermonkey >= 78 will return the detailed `major.minor.patch` as opposed to just the `major` version as previously.

Bugfixes

- #3817: Fix undefined function call in `weatherreport`.
- #3819: Return `400` instead of `500` response code for known invalid `_bulk_docs` with `new_edits=false` request.
- #3861: Add `SameSite` setting when clearing session cookies.
- #3863: Fix custom TLS distribution for Erlang 20.
- #3870: Always send all cookie attributes.
- #3886: Avoid changes feed rewind after shard move with no subsequent db updates.
- #3888: Make `_stats` endpoint resilient against nodes that go offline.
- #3901: Use db-creation time instead of `0` for `instance_start_time` to help replicator recognise whether a peer database was deleted and recreated.
- #3909: Fix `new_edits:false` and VDU `function_clause`.
- #3934: Fix `replicated_changes` typo for purge doc updates.
- #3940: Ensure the multipart parser always monitors the worker and make sure to wait for attachment uploads before responding.
- #3950: Ignore responses from timed-out or retried `ibrowse` calls.
- #3969: Fix `skip` and `limit` for `_all_dbs` and `_dbs_info`.
- #3979: Correctly respond with a `500` code when document updates time out under heavy load.
- #3992: Show that Search is available if it was available before. Avoid Search availability disappearing just because a Search node was temporarily not available.
- #3993: Return a `400` error when decoding a JWT token fails, rather than crashing and not responding at all.
- #3990: Prevent creation of ddocs with no name through Mango index creation.
- #4003: Improve index building during shard splitting.
- #4016: Fix `function_clause` error for replicated changes with a target VDU.

- #4020: Fix `maybe_handle_error` clauses.
- #4037: Fix ES{256,384,512} support for JWTs.
- #4040: Handle `exit(shutdown)` error in `chttpd`.
- #4043: Fix purge request timeouts (5s -> infinity).
- #4146: The `devcontainer` has been updated.
- #4050: Handle `all_dbs_active` in `fabric_doc_update`.
- #4160: Return a proper `400` error when an invalid object is sent to `_bulk_get`.
- #4070: Prevent `error:function_clause` in `check_security/3` if roles claim is malformed.
- #4075: Fix `couch_debug:opened_files*` functions.
- #4108: Trim `X-Auth-CouchDB-Roles` header after reading.
- #4153: The `require_valid_user` setting is now under `chttpd`.
- #4161: Fix `content-type` handling in `_session`.
- #4176: Fix `eventsource` `_changes` feed.
- #4197: Support large (and impractical as-of-yet) `q` values. Fix shard open timeouts for `q > 64`.
- #4199: Fix spurious unlock in `close_db_if_idle`.
- #4230: Avoid refresh messages piling up in prometheus server.
- #4240: Implement global password hasher process. This fixes a race-condition when setting new admin passwords in quick succession on a multicore server.
- #4261, #4271: Clean up stale view checkpoints, improve purge client cleanup logging
- #4272: Kill all `couch_server_N` if `database_dir` changes.
- #4313: Use `chttpd` config section when patching local `_replicate` endpoints.
- #4321: Downgrade jiffy to allow building on Windows again.
- #4329, #4323: Ignore build windows binaries in git.

Tests

- #3825: Eliminate Elixir compiler warnings.
- #3830: Reduce skipped Elixir integration tests.
- #3890: Handle `not_found` lookups removing ddoc cache key.
- #3892: Use Debian Stable for CI, add Erlang 24 to CI.
- #3898: Remove CI support for Ubuntu 16.04.
- #3903, #3914: Refactor Jenkins to dynamically generate stages. Drop `MINIMUM_ERLANG_VERSION` to 20, drop the packaging `ERLANG_VERSION` to 23, add the `weatherreport-test` as a build step, and add `ARM` and `POWER` back into the matrix.
- #3921:, #3923: Execute various tests in clean `database_dir` to avoid subsequent test flakiness.
- #3968: Ensure key tree rev stemming does't take too much memory.
- #3980: Upgrade Mango test dependency `nose` to `nose` and fix flaky-on-Windows tests.
- #4006: Remove CI support for Debian 9.
- #4061, #4082: Update PPC CI builder.
- #4096: Fix flaky `validate_doc_update` Elixir test.
- #4123: Fix `haproxy.cfg`.

- #4126: Return a `400` response for a single `new_edits=false` doc update without revision.
- #4129: Fix `proxyauth_test` and removed it from skip list.
- #4132: Address race condition in `cpse_incref_decref` test.
- #4151: Refactor replication tests to use clustered endpoints.
- #4178: Add test coverage to prevent junk in `eventsource`.
- #4188: Enable eunit coverage for all applications instead of enabling it per-application.
- #4202: Fix race condition in ddoc cache LRU test.
- #4203, #4205: Reduce test log noise.
- #4268: Improve flaky `_dbs_info` test.
- #4319: Fix offline `configure` and `make release`.
- #4328: Fix `eaddrnotavail` in Elixir tests under Windows.
- #4330: Do not run source checks in main CI build.

Docs

- #4164: The CouchDB documentation has been moved into the main CouchDB repository.
- #4307, #4174: Update Sphinx to version 5.3.0
- #4170: Document the `/_node/{node-name}/_versions` endpoint.

Builds

- #4097: Stop publication of nightly packages. They were not used anywhere.
- #4322: Reuse installed rebar and rebar3 for mix. Compatible with Elixir =< 13 only. Elixir 14 is not supported yet.
- #4326: Move Elixir source checks to a separate build step.

Other

- Added pumpkin spice to selected endpoints. — Thank you for reading the 3.3.0 release notes.

16.2 3.2.x Branch

- *Version 3.2.3*
- *Version 3.2.2*
- *Version 3.2.1*
- *Version 3.2.0*

16.2.1 Version 3.2.3

Features and Enhancements

- #4529: In Javascript process manager, use a database tag in addition to a ddoc ID to quickly find processes. This should improve performance.
- #4509, #4405: Make remsh work with quoted cookie values.
- Fix for CVE-2023-26268.

16.2.2 Version 3.2.2

Bugfixes

- Fix for CVE-2022-24706. This is a security release for a *critical* vulnerability.
- #3963: Optimize compaction and doc updates for conflicted documents on Erlang versions higher than 21.
- #3852: Add support for SpiderMonkey 91esr.

16.2.3 Version 3.2.1

Features and Enhancements

- #3746: `couch_icu_driver` collation driver has been removed. ICU collation functionality is consolidated in the single `couch_ejson_compare` module. View performance might slightly increase as there are less corner cases when the C collation driver fails and falls back to Erlang.
- #3787: Update sequences generated from DB info and `_changes?since=now&limit=0` now contain shard uuids as part of their internal, opaque, representation. As a result, there should be less chance of experiencing changes feed rewinds with these sequences.
- #3798: ICU driver and collator algorithm versions are returned in the `_node/{node-name}/_versions` result.
- #3801: Users with the `_metrics` role can now read `_prometheus` metrics.

Bugfixes

- #3780: Avoid changes feed rewinds after shard moves.
- #3779, #3785: Prevent deleted view file cleanup from crashing when database is deleted while the cleanup process is running.
- #3789: Fix `badarith` 500 errors when `[fabric] request_timeout` is set to `infinity`.
- #3786: Fix off-by-one `limit` error for `_all_dbs`. Also, the auto-injected shard `_dbs` design doc is removed and replaced with an Erlang module.
- #3788: Minimize changes feeds rewinds when a node is down.
- #3807: Enable `custodian` application reporting. Previously, `custodian` was accidentally left disabled as it used a hard-coded shards db name different than `_dbs`.
- #3805: Cluster setup correctly syncs admin passwords and uses the new (since 3.2.0) `[chttpd_auth]` config section instead of the previous `[couch_httpd_auth]` section.
- #3810: Local development `dev/run` script now uses the `[chttpd_auth]` section in `local.ini` instead of `[couch_httpd_auth]`.
- #3773: Fix reduce view collation results for unicode equivalent keys.

16.2.4 Version 3.2.0

Features and Enhancements

- #3364: CouchDB's replicator now implements a Fair Share replication scheduler. Rather than using a round-robin scheduling mechanism, this update allows specifying the relative priority of jobs via different `_replicator` databases. More information is available in the *_replicator DB docs*.

- #3166: Allow custom JWT claims for roles, via the `[jwt_auth] roles_claim_name` config setting.
- #3296, #3312: CouchDB now includes `weatherreport` and its dependency `custodian`, a diagnostic app forked from Basho's `riaknostic` tool. More documentation is available in the *Cluster Troubleshooting* section.
- #2911, #3298, #3425: CouchDB now returns the version of SpiderMonkey to administrators in the `GET /_node/{node-name}/_versions` response.
- #3303: CouchDB now treats a `408` response received by the replicator similar to any `5xx` error (by retrying, as opposed to a permanent error). CouchDB will never return a `408`, but some reverse proxies in front of CouchDB may return this code.
- #3322: `_session` now accepts gzip encoding.
- #3254: The new `$keyMapMatch` operator allows Mango to query on the keys of a map. It is similar to the `$elemMatch` operator, but instead of operating on the elements of array, it operates on the keys of a map.
- #3336: Developers now have access to a `.devcontainer` configuration for the 3.x version of CouchDB, right in the source code repository.
- #3347: The default maximum attachment size has been reduced from `infinity` to 1 GiB.
- #3361: Compaction process suspension now appears in the `active_tasks` output, allowing administrators to verify that the `strict_window` value is being respected.
- #3378: The `[admins]` section and the `[replicator] password` are now redacted from all logs. In addition, #3380 removes user credentials, user documents and design documents from logfiles as much as possible. Further, #3489 no longer logs all of the messages received by a terminated internal Erlang process.
- #3421, #3500: CouchDB now supports SpiderMonkey 78 and 86.
- #3422: CouchDB now supports Erlang/OTP 23 and `error_logger` reports for Erlang/OTP >= 21.
- #3566: CouchDB now also supports Erlang/OTP 24.
- #3571: CouchDB *no longer supports Erlang/OTP 19*.

- #3643: Contribute a custom Erlang network protocol to CouchDB, users can specify nodes to use TCP or TLS.

- #3472, #3473, #3609: Migrate some config options from `[httpd]` to `[chttpd]`, migrate some from `[couch_httpd_auth]` to `[chttpd_auth]`, and comment all out in the `default.ini`.
  - Config options moved from `[httpd]` to `[chttpd]`: `allow_jsonp`, `changes_timeout`, `config_whitelist`, `enable_cors`, `secure_rewrites`, `x_forwarded_host`, `x_forwarded_proto`, `x_forwarded_ssl`, `enable_xframe_options`, `max_http_request_size`.
  - Config options moved from `[couch_httpd_auth]` to `[chttpd_auth]`: `authentication_redirect`, `timeout`, `auth_cache_size`, `allow_persistent_cookies`, `iterations`, `min_iterations`, `max_iterations`, `password_scheme`, `proxy_use_secret`, `public_fields`, `secret`, `users_db_public`, `x_auth_roles`, `x_auth_token`, `x_auth_username`, `cookie_domain`, `same_site`
- #3586: We added a new way of specifying basic auth credentials which can include various characters previously not allowed to be included in the url info part of endpoint urls.
- #3483: We added a way of specifying requirements for new user passwords using a list of regular expressions.
- #3506, #3416, #3377: CouchDB now provides a Prometheus compatible endpoint at `GET /_node/{node-name}/_prometheus`. A configuration option allows for scraping via a different port (17986) that does not require authentication, if desired. More information is available at the *Prometheus API endpoint* summary.
- #3697, COUCHDB-883 (JIRA): As an opt-in policy, CouchDB can now stop encoding the plus sign + in non-query parts of URLs, in compliance with the original CouchDB standards. The opt-in is via the `[chttpd] decode_plus_to_space = true` setting. *In CouchDB 4.x, this is going to be an opt-out policy.*
- #3724: CouchDB now has new CSP settings for attachments and show/list functions. This deprecates the old `[csp] enable` and `[csp] header_value` settings, replacing them with the new `[csp] utils_enable` and `[csp] utils_header_value` settings respectively. In addition, new settings for `attachments_enable`, `attachments_header_value`, `showlist_enable` and `showlist_header_value` now are available. Documentation is in the `default.ini` file.
- #3734, #3733: Users with databases that have low `q` and `n` values would often receive the `No DB shards could be opened` error when the cluster is overloaded, due to a hard-coded 100ms timeout. CouchDB now calculates a more reasonable timeout, based on the number of shards and the overall maximum fabric request timeout limit, using a geometric series.

Performance

- #3337: Developer nodes now start faster when using the `dev/run` script.
- #3366: The monolithic `couch_server` process has been sharded for performance. Previously, as a single `gen_server`, the process would have a finite throughput that, in busy clusters, is easily breached – causing a sizeable backlog in the message queue, ultimately leading to failure and errors. No more! The aggregate message queue info is still available in the `_system` output. (#3370)
- #3208: CouchDB now uses the latest ibrowse 4.4.2 client for the replicator.
- #3600, #3047, #3019: The default `slack` channel for smoosh auto-compaction has been increased to a more reasonable value, reducing load on systems that would have normally been idle in CouchDB 2.x (where no auto-compaction daemon exists).
- #3711: Changes feeds may no longer rewind after shard moves, assuming the node and range specified by the changes feed nonce can still match an existing node's shard.

Bugfixes

- Complete retirement of the JavaScript test suite - replaced by Elixir. Hooray!
- #3165: Allow configurability of JWT claims that require a value. Also fixes #3232. Further, #3392 no longer validates claims provided that CouchDB does not require.
- #3160, #3161: The `run_queue` statistic now returns valid information even when using Erlang BEAM dirty CPU and IO queues.
- #3162: Makefiles updated to include local configs & clean configs when running `make devclean`.
- #3195: The `max_document_size` parameter now has a clearer explanation in `default.ini`.
- #3207, #2536: Improve the `INSTALL.Unix.md` file.
- #3212: Base and extra headers are properly combined when making replicator requests that contain duplicate headers.
- #3201: When using a POST with request body to pass parameters to a view-like request, the boolean parameters are accepting only JSON strings, but not booleans. Now, CouchDB accepts `true` and `false` for the `stable` parameter, in addition to `"true"` and `"false"`. comment in
- #1988: Attachment operations `PUT /db/doc` and `POST /db` now perform consistent attachment name validation.
- #3249: Documents with lots of conflicts no longer blow up couchjs if the user calls `_changes` with a JS filter and with `style=all_docs`.
- #3144: Respawning compaction jobs to catch up with intervening changes are now handled correctly by the smoosh monitor.
- #3252: CouchDB now exports the `couch_util:json_decode/2` function to support maps instead of the default data structure.
- #3255, #2558: View files that have incorrect `db_headers` now reset the index forcing a rebuild.
- #3271: Attachments that are stored uncompressed but later replicated to nodes that compress the attachment no longer fail an internal md5 check that would break eventual consistency between nodes.
- #3277: `req_body` requests that have `req_body` set already now properly return the field without parsing.
- #3279: Some default headers were missing from some responses in replication, including `X-CouchDB-Body-Time` and `X-Couch-Request-ID`.
- #3329, #2962: CouchDB no longer returns broken couchjs processes to the internal viewserver process pool.
- #3340, #1943: `PUT`s of `multipart/related` attachments now support a `Transfer-Encoding` value of `chunked`. Hooray!

- #2858, #3359: The cluster setup wizard no longer fails when a request to `/` is not made before a request to `finish_cluster`.
- #3368: Changing the `max_dbs_open` configuration setting correctly ensures that each new `couch_server_X` property receives `1/num_servers()` of it.
- #3373: Requests to `{db}/_changes` with a custom filter no longer result in a fabric request timeout if the request body is not available to additional cluster nodes, resulting in a more descriptive exit message and proper JSON object validation in the payload.
- #3409: The internal `chttpd_external:json_req_obj/2` function now reads the cached `peer` before falling back to a socket read operation.
- #3335, #3617, #3708: The `COUCHDB_FAUXTON_DOCROOT` environment variable is now introduced to allow its explicit overriding at startup.
- #3471: http clients should no longer receive stacktraces unexpectedly.
- #3491: libicu tests no longer fail on older OS releases such as CentOS 6 and 7.
- #3541: Usernames and passwords can now contain `@` and not break the CouchDB replicator.
- #3545: The `dreyfus_index_manager` process now supports offheap message queues.
- #3551: The replication worker pool now properly cleans up worker processes as they are done via the `worker_trap_exits = false` setting.
- #3633, #3631: All code paths for creating databases now fully respect db creation options, including partitioning options.
- #3424, #3362: When using `latest=true` and an old revision with conflicting children as rev is specified, CouchDB no longer returns an `"error": "case_clause"` response.
- #3673: Non-existent attachments now return a `404` when the attachment is missing.
- #3698: The `dev/run` development script now allows clusters where `n > 5`.
- #3700: The `maybe_close` message is now sent to the correct internal process.
- #3183: The smoosh operator guide now recommends to use the `rpc:multicall` function.
- #3712: Including a payload within a `DELETE` operation no longer hangs the next request made to the same mochiweb acceptor.
- #3715: For clusters with databases where `n > [cluster] n`, attachments chunks are longer dropped on quorum writes.
- #3507: If a file is truncated underneath CouchDB, CouchDB will now log the filename if it finds this situation with a `file_truncate_error`.
- #3739: Shards with large purge sequences no longer fail to split in a shard splitting job.
- #3754: Always return views meta info when `limit=0` and `sorted=true`.
- #3757: Properly sort `descending=true` view results with a `keys` list.
- #3763: Stabilize view row sorting order when they are merged by the coordinator.

Other

- Donuts for everyone! Er, not really - thank you for reading the 3.2 release notes.

16.3 3.1.x Branch

- *Version 3.1.2*
- *Version 3.1.1*
- *Version 3.1.0*

16.3.1 Version 3.1.2

This is a security release for a *low severity* vulnerability. Details of the issue will be published one week after this release. See the CVE database for details at a later time.

16.3.2 Version 3.1.1

Features and Enhancements

- #3102, #1600, #2877, #2041: When a client disconnects unexpectedly, CouchDB will no longer log a "`normal : unknown`" error. Bring forth the rainbows.

- #3109: Drilldown parameters for text index searches may now be specified as a list of lists, to avoid having to define this redundantly in a single query. (Some languages don't have this facility.)
- #3132: The new `[chttpd] buffer_response` option can be enabled to delay the start of a response until the end has been calculated. This increases memory usage, but simplifies client error handling as it eliminates the possibility that a response may be deliberately terminated midway through, due to a timeout. This config value may be changed at runtime, without impacting any in-flight responses.

Performance

Bugfixes

- #2935: The replicator now correctly picks jobs to restart during rescheduling, where previously with high load it may have failed to try to restart crashed jobs.
- #2981: When handling extremely large documents (50MB), CouchDB can no longer time out on a `gen_server:call` if bypassing the IOQ.
- #2941: CouchDB will no longer fail to compact databases if it finds files from a 2.x compaction process (prior to an upgrade) on disk.
- #2955 CouchDB now sends the correct CSP header to ensure Fauxton operates correctly with newer browsers.
- #3061, #3080: The *couch_index* server won't crash and log errors if a design document is deleted while that index is building, or when a ddoc is added immediately after database creation.
- #3078: CouchDB now checks for and complains correctly about invalid parameters on database creation.
- #3090: CouchDB now correctly encodes URLs correctly when encoding the `atts_since` query string.
- #2953: Some parameters not allowed for text-index queries on partitioned database are now properly validated and rejected.
- #3118: Text-based search indexes may now be cleaned up correctly, even if the design document is now invalid.
- #3121: `fips` is now only reported in the welcome message if FIPS mode was enabled at boot (such as in `vm.args`).
- #3128: Using COPY to copy a document will no longer return a JSON result with two `ok` fields.
- #3138: Malformed URLs in replication requests or documents will no longer throw an error.

Other

- JS tests skip faster now.
- More JS tests ported into elixir: `reader_acl`, `reduce_builtin`, `reduce_false`, `rev_stemming`, `update_documents`, `view_collation_raw`, `view_compaction`, all the `view_multi_key` tests, `view_sandboxing`, `view_update_seq`.

16.3.3 Version 3.1.0

Features and Enhancements

- #2648: Authentication via *JSON Web Token (JWT)*. Full documentation is at the friendly link.
- #2770: CouchDB now supports linking against SpiderMonkey 68, the current Mozilla SpiderMonkey ESR release. This provides direct support for packaging on the latest operating system variants, including Ubuntu 20.04 "Focal Fossa."
- **A new Fauxton release is included, with updated dependencies, and a new optional**
  CouchDB news page.

Performance

- #2754: Optimized compactor performance, resulting in a 40% speed improvement when document revisions approach the `revs_limit`. The fixes also include additional metrics on size tracking during the sort and copy phases, accessible via the *`GET /_active_tasks`* endpoint.
- A big bowl of candy! OK, no, not really. If you got this far…thank you for reading.

16.4 3.0.x Branch

- *Upgrade Notes*
- *Version 3.0.1*
- *Version 3.0.0*

16.4.1 Upgrade Notes

- #2228: The default maximum document size has been reduced to 8MB. This means that databases with larger documents will not be able to replicate into CouchDB 3.0 correctly without modification. This change has been made in preparation for anticipated hard upper limits on document size imposed by CouchDB 4.0. For 3.x, the max document size setting can be relaxed via the `[couchdb] max_document_size` config setting.
- #2228: The default database sharding factor `q` has been reduced to 2 by default. This, combined with automated database resharding (see below), is a better starting place for new CouchDB databases. As in CouchDB 2.x, specify `?q=#` to change the value upon database creation if desired. The default can be changed via the config `[cluster] q` setting.
- #1523, #2092, #2336, #2475: The "node-local" HTTP interface, by default exposed on port 5986, has been removed. All functionality previously available at that port is now available on the main, clustered interface (by default, port 5984). Examples:

```
GET /_node/{nodename}/_stats
GET /_node/{nodename}/_system
GET /_node/{nodename}/_all_dbs
GET /_node/{nodename}/_uuids
GET /_node/{nodename}/_config
GET /_node/{nodename}/_config/couchdb/uuid
POST /_node/{nodename}_config/_reload
GET /_node/{nodename}/_nodes/_changes?include_docs=true
PUT /_node/{nodename}/_dbs/{dbname}
POST /_node/{nodename}/_restart
GET /_node/{nodename}/{db-shard}
GET /_node/{nodename}/{db-shard}/{doc}
GET /_node/{nodename}/{db-shard}/{ddoc}/_info
```

  …and so on. Documentation has been updated to reflect this change.

  > **Warning:** The _node endpoint is for adminstrative purposes it is NOT intended as an alternative to the regular endpoints ("GET /dbname", "PUT /dbname/docid" and so on)

- #2389: CouchDB 3.0 now requires a server admin user to be defined at startup, or will print an error message and exit. If you do not have one, be sure to *create an admin user*. (The Admin Party is now over.)
- #2576: CouchDB 3.0 now requires admin-level access for the `/_all_dbs` endpoint.

Fig. 1: CC-BY-NC 2.0: hehaden @ Flickr

- #2339: All databases are now created by default as admin-only. That is, the default new database `_security` object is now:

```
{
  "members" : { "roles" : [ "_admin" ] },
   "admins" : { "roles" : [ "_admin" ] }
}
```

  This can be changed after database creation.

- Due to code changes in #2324, it is not possible to upgrade transparently from CouchDB 1.x to 3.x. In addition, the `couchup` utility has been removed from CouchDB 3.0 by #2399. If you are upgrading from CouchDB 1.x, you must first upgrade to CouchDB 2.3.1 to convert your database and indexes, using `couchup` if desired. You can then upgrade to CouchDB 3.0. Or, you can start a new CouchDB 3.0 installation and replicate directly from 1.x to 3.0.
- #1833, #2358, #1871, #1857: CouchDB 3.0 supports running only under the following Erlang/OTP versions:
  - 19.x - "soft" support only. No longer tested, but should work.
  - 20.x - must be newer than 20.3.8.11 (20.0, 20.1, 20.2 versions all invalid)
  - 21.x - for 21.2, must be newer than 21.2.3
  - 22.x - for 22.0, must be newer than 22.0.5
- #1804: By default, views are limited to return a maximum of 2**28 (268435456) results. This limit can be configured separately for views and partitioned views via the `query_limit` and `partition_query_limit` values in the ini file `[query_server_config]` section.
- After upgrading all nodes in a cluster to 3.0, add `[rexi] use_kill_all = true` to `local.ini` to save some intra-cluster network bandwidth.

Deprecated feature removal

The following features, deprecated in CouchDB 2.x, have been removed or replaced in CouchDB 3.0:

- #2089, #2128, #2251: Local endpoints for replication targets, which never functioned as expected in CouchDB 2.x, have been completely removed. When replicating databases, always specify a full URL for the source and target. In addition, the node local `_replicator` database is no longer automatically created.
- #2163: The `disk_size` and `data_size` fields have been retired from the database info object returned by `GET /{db}/`. These were deprecated in CouchDB 2.x and replaced by the `sizes` object, which contains the improved `file`, `active` and `external` size metrics. Fauxton has been updated to match.
- #2173: The ability to submit multiple queries against a view using the POST to `/{db}/_design/{ddoc}/_view/{view}` with the `?queries=` option has been replaced by the new *queries* endpoint. The same is true of the *_all_docs, _design_docs, and _local_docs* endpoints. Specify a `keys` object when POST-ing to these endpoints.
- #2248: CouchDB externals (`_external/`) have been removed entirely.

- #2208: CouchDB no longer supports the `delayed_commits` option in the configuration file. All writes are now full commits. The `/_ensure_full_commit` API endpoint has been retained (as a no-op) for backwards compatibility with old CouchDB replicators.
- #2395: The security object in the `_users` database cannot be edited by default. A setting exists in the configuration file to revert this behaviour. The ability to override the disable setting is expected to be removed in CouchDB 4.0.

Deprecated feature warnings

The following features are deprecated in CouchDB 3.0 and will be removed in CouchDB 4.0:

- Show functions (`/{db}/{ddoc}/_show`)
- List functions (`/{db}/{ddoc}/_list`)
- Update functions (`/{db}/{ddoc}/_update`)
- Virtual hosts and ini-file rewrites
- Rewrite functions (`/{db}/{ddoc}/_rewrite`)

16.4.2 Version 3.0.1

Features and Enhancements

- Fauxton was updated to version *v1.2.3*.

Bugfixes

- #2441: A memory leak when encoding large binary content was patched. This should resolve a long-standing gradual memory increase bug in CouchDB.
- #2613: Simultaneous attempts to create the same new database should no longer result in a 500 Internal Server Error error.
- #2678: Defaults for the `smoosh` compaction daemon are now consistent with the shipped `default.ini` file.
- #2680: The Windows CouchDB startup batch file will no longer fail to start CouchDB if incompatible versions of OpenSSL are on the `PATH`.
- #2741: A small performance improvement in the `couch_server` process was made.
- #2745: The `require_valid_user` exception logic was corrected.
- #2643: The `users_db_security_editable` setting is now in the correct section of the `default.ini` file.
- #2654: Filtered changes feeds that need to rewind partially should no longer rewind all the way to the beginning of the feed.
- #2655: When deleting a session cookie, CouchDB should now respect the operator-specified cookie domain, if set.
- #2690: Nodes that re-enter a cluster after a database was created (while the node was offline or in maintenance mode) should more correctly handle creating local replicas of that database.
- #2805: Mango operators more correctly handle being passed empty arrays.
- #2716, #2738: The `remsh` utility will now try and guess the node name and Erlang cookie of the local installation. It will also respect the `COUCHDB_ARGS_FILE` environment variable.
- #2797: The cluster setup workflow now uses the correct logging module.
- #2818: Mango now uses a safer method of bookmark creation that prevents unexpectedly creating new Erlang atoms.

- #2756: SpiderMonkey 60+ will no longer corrupt UTF-8 strings when various JS functions are applied to them.
- Multiple test case improvements, including more ports of JS tests to Elixir.

16.4.3 Version 3.0.0

Features and Enhancements

- #1789: *User-defined partitioned databases*.

  These special databases support user-driven placement of documents into the same shard range. *JavaScript views* and *Mango indexes* have specific optimizations for partitioned databases as well.

  Two tweakable configuration parameters exist:

  - #1842: Partition size limits. By default, each partition is limited to 10 GiB.
  - #1684: Partitioned database support can be disabled via feature flag in `default.ini`.
- #1972, #2012: *Automated shard splitting*. Databases can now be re-sharded *while online* to increase the `q` factor to a larger number. This can be configured to require specific node and range parameters upon execution.
- #1910: *Automatic background indexing*, internally known as `ken`. This subsystem ensures secondary indexes (such as JavaScript, Mango, and text search) are kept up to date, without requiring an external query to trigger building them. Many configuration parameters are available.
- #1904: Completely rewritten *automatic compaction daemon*, internally known as `smoosh`. This subsystem automatically triggers background compaction jobs for both databases and views, based on *configurable thresholds*.
- #1889, #2408: New IO Queue subsystem implementation. This is *highly configurable and well-documented*.
- #2436, #2455: CouchDB now regression tests against, and officially supports, running on the `arm64v8` (`aarch64`) and `ppc64le` (`ppc64el`) machine architectures. Convenience binaries are generated on these architectures for Debian 10.x ("buster") packages, and for the Docker containers.
- #1875, #2437, #2423: CouchDB now supports linking against SpiderMonkey 60 or SpiderMonkey 1.8.5. SpiderMonkey 60 provides enhanced support for ES5, ES6, and ES2016+. Full compatibility information is available at the ECMAScript compatibility table. Click on "Show obsolete platforms", then look for "FF 60 ESR" in the list of engine types.

  However, it was discovered that on some ARM 64-bit distributions, SM 60 segfaults frequently, including the SM 60 packages on CentOS 8 and Debian 10.

  As a result, CouchDB's convenience binaries **only link against SM 60 on the ``x86_64`` and ``ppc64le`` architectures**. This includes the Docker image for these architectures.

  At present, CouchDB ships with SM 60 linked in on the following binary distributions:

  - Debian buster (10.x)
  - CentOS / RedHat 8.x
  - macOS (10.10+)
  - Windows (7+)
  - Docker (3.0.0)
  - FreeBSD (CURRENT)

  We expect to add SM 60 support to Ubuntu with Focal Fossa (20.04 LTS) when it ships in April 2020.

  It is unlikely we will backport SM 60 packages to older versions of Debian, CentOS, RedHat, or Ubuntu.
- The Windows installer has many improvements, including:

  - Prompts for an admin user/password as CouchDB 3.0 requires * Will not overwrite existing credentials if in place
  - No longer remove user-modified config files, closing #1989 * Also will not overwrite them on install.
  - Checkbox to disable installation of the Windows service
  - *Silent install support*.
  - Friendly link to these online release notes in the exit dialog
  - Higher resolution icon for HiDPI (500x500)

Warning: Windows 8, 8.1, and 10 require the .NET Framework v3.5 to be installed.

- #2037: Dreyfus, the CouchDB side of the Lucene-powered search solution, is now shipped with CouchDB. When one or more Clouseau Java nodes are joined to the cluster, text-based indexes can be enabled in CouchDB. It is recommended to have as many Clouseau nodes as you have CouchDB nodes. Search is advertised in the feature list present at `GET /` if configured correctly (#2206). *Configuration* and *installation documentation is available*.
- #2411: The `/_up` endpoint no longer requires authentication, even when `require_valid_user` is `true`.
- #2392: A new `_metrics` role can be given to a user. This allows that user access only to the `/_node/{node}/_stats` and `/_node/{node}/_system` endpoints.
- #1912: A new alternative `systemd-journald logging` backend has been added, and can be enabled through the ini file. The new backend does not include CouchDB's microsecond-accurate timestamps, and uses the `sd-daemon(3)` logging levels.
- #2296, #1977: If the configuration file setting `[couchdb] single_node` is set to `true`, CouchDB will automatically create the system databases on startup if they are not present.
- #2338, #2343: POST request to CouchDB views and the `/{db}/_all_docs`, `/{db}/_local_docs` and `/{db}/_design_docs` endpoints now support the same functionality as GET. Parameters are passed in the body as a JSON object, rather than in the URL when using POST.
- #2292: The `_scheduler/docs` and `_scheduler/info` endpoints now return detailed replication stats for running and pending jobs.
- #2282, #2272, #2290: CouchDB now supports specifying separate proxies for both the `source` and `target` in a replication via `source_proxy` and `target_proxy` keys. The *API documentation* has been updated.
- #2240: Headers are now returned from the `/{db}/_changes` feed immediately, even when there are no changes available. This avoids client blocking.
- #2005, #2006: The name of any node can now be retrieved through the *new API endpoint* `GET /_node/{node-name}`.
- #1766: Timeouts for requests, `all_docs`, attachments, views, and partitioned view requests can all be specified separately in the ini file under the `[fabric]` section. See `default.ini` for more detail.
- #1963: Metrics are now kept on the number of partition and global view queries, along with the number of timeouts that occur.
- #2452, #2221: A new configuration field `[couch_httpd_auth] same_site` has been added to set the value of the CouchDB auth cookie's `SameSite` attribute. It may be necessary to set this to `strict` for compatibility with future versions of Google Chrome. If CouchDB CORS support is enabled, set this to `None`.

Performance

- #2277: The `couch_server` process has been highly optimized, supporting significantly more load than before.
- #2360: It is now possible to make the rexi interface's unacked message limit configurable. A new, more optimized default (5, lowered from 10) has been set. This results in a ~50% improvement on view queries on large clusters with `q 8`.
- #2280: Connection sharing for replication now functions correctly when replicating through a forward proxy. Closes #2271.
- #2195, #2207: Metrics aggregation now supports CouchDB systems that sleep or hibernate, ensuring that on wakeup does not trigger thousands of unnecessary function calls.
- #1795: Avoid calling `fabric:update_docs` with empty doc lists.
- #2497: The setup wizard no longer automatically creates the `_global_changes` database, as the majority of users do not need this functionality. This reduces overall CouchDB load.

Bugfixes

- #1752, #2398, #1803: The cluster setup wizard now ensures a consistent UUID and http secret across all nodes in a cluster. CouchDB admin passwords are also synced when the cluster setup wizard is used. This prevents being logged out when using Fauxton as a server admin user through a load balancer.
- #2388: A compatibility change has been made to support replication with future databases containing per-document access control fields.
- #2379: Any replicator error messages will provide an object in the response, or null, but never a string.
- #2244, #2310: CouchDB will no longer send more data than is requested when retrieving partial attachment data blocks.
- #2138: Manual operator updates to a database's shard map will not corrupt additional database properties, such as partitioning values.
- #1877: The `_purge` and `_purged_infos_limit` endpoints are now correctly restricted to server admin only.
- #1794: The minimum purge sequence value for a database is now gathered without a clustered `_all_docs` lookup.
- #2351: A timeout case clause in `fabric_db_info` has been normalised to match other case clauses.
- #1897: The `/{db}/_bulk_docs` endpoint now correctly catches invalid (*i.e.*, non-hexadecimal) `_rev_` values and responds with a 400 Bad Request error.
- #2321: CouchDB no longer requires Basic auth credentials to reach the `/_session` endpoint for login, even when `require_valid_user` is enabled.
- #2295: CouchDB no longer marks a job as failed permanently if the internal doc processor crashes.
- #2178: View compaction files are now removed on view cleanup.
- #2179: The error message logged when CouchDB does not have a `_users` database is now less scary.
- #2153: CouchDB no longer may return a `badmatch` error when querying `all_docs` with a passed `keys` array.
- #2137: If search is not available, return a 400 Bad Request instead of a 500 Internal Server Error status code.
- #2077: Any failed `fsync(2)` calls are now correctly raised to avoid data corruption arising from retry attempts.
- #2027: Handle epoch mismatch when duplicate UUIDs are created through invalid operator intervention.

- #2019: If a database is deleted and re-created while internal cluster replication is still active, CouchDB will no longer retry to delete it continuously.
- #2003, #2438: CouchDB will no longer automatically reset an index file if any attempt to read its header fails (such as when the couch_file process terminates unexpectedly). CouchDB now also handles the case when a view file lacks a proper header.
- #1983: Improve database "external" size calcuation to be more precise.
- #1971: Correctly compare ETags using weak comparison methods to support *W/* prefix added by some load balancer configurations.
- #1901: Invalid revision specified for a document update will no longer result in a `badarg` crash.
- #1845: The `end_time` field in `/_replicate` now correctly converts time to UTC.
- #1824: `rexi` stream workers are now cleaned up when the coordinator process is killed, such as when the ddoc cache is refreshed.
- #1770: Invalid database `_security` objects no longer return a `function_clause` error and stack trace.
- #2412: Mango execution stats now correctly count documents read which weren't followed by a match within a given shard.
- #2393, #2143: It is now possible to override the query server environment variables `COUCHDB_QUERY_SERVER_JAVASCRIPT` and `COUCHDB_QUERY_SERVER_COFFEESCRIPT` without overwriting the `couchdb/couchdb.cmd` startup scripts.
- #2426, #2415: The replicator now better handles the situation where design document writes to the target fail when replicating with non-admin credentials.
- #2444, #2413: Replicator error messages are now significantly improved, reducing `function_clause` responses.
- #2454: The replication auth session plugin now ignores other cookies it may receive without logging an error.
- #2458: Partitioned queries and dreyfus search functions no longer fail if there is a single failed node or rexi worker error.
- #1783: Mango text indexes no longer error when given an empty selector or operators with empty arrays.
- #2466: Mango text indexes no longer error if the indexed document revision no longer exists in the primary index.
- #2486: The `$lt`, `$lte`, `$gt`, and `$gte` Mango operators are correctly quoted internally when used in conjunction with a text index search.
- #2493: The `couch_auth_cache` no longer has a runaway condition in which it creates millions of monitors on the `_users` database.

Other

The 3.0.0 release also includes the following minor improvements:

- #2472: CouchDB now logs the correct, clustered URI at startup (by default: port `5984`.)
- #2034, #2416: The path to the Fauxton installation can now be specified via the `COUCHDB_FAUXTON_DOCROOT` environment variable.
- #2447: Replication stats are both persisted when jobs are re-created, as well as properly handled when bulk document batches are split.
- #2410, #2390, #1913: Many metrics were added for Mango use, including counts of unindexed queries, invalid index queries, docs examined that do and don't meet cluster quorum, query time, etc.
- #2152, #2504: CouchDB can now be started via a symlink to the binary on UNIX-based platforms.
- #1844: A new internal API has been added to write custom Erlang request-level metrics reporting plugins.

- #2293, #1095: The `-args_file`, `-config` and `-couch_ini` parameters may now be overridden via the `COUCHDB_INI_FILES` environment variable on UNIX-based systems.
- #2352: The `remsh` utility now searches for the Erlang cookie in `ERL_FLAGS` as well as `vm.args`.
- #2324: All traces of the (never fully functional) view-based `_changes` feed have been expunged from the code base.
- #2337: The md5 shim (introduced to support FIPS-compliance) is now used consistently throughout the code base.
- #2270: Negative and non-integer `heartbeat` values now return 400 Bad Request.
- #2268: When rescheduling jobs, CouchDB now stops sufficient running jobs to make room for the pending jobs.
- #2186: CouchDB plugin writers have a new field in which endpoint credentials may be stashed for later use.
- #2183: `dev/run` now supports an `--extra-args` flag to modify the Erlang runtime environment during development.
- #2105: `dev/run` no longer fails on unexpected remote end connection close during cluster setup.
- #2118: Improve `couch_epi` process replacement mechanism using map childspecs functionality in modern Erlang.
- #2111: When more than `MaxJobs` replication jobs are defined, CouchDB now correctly handles job rotation when some jobs crash.
- #2020: Fix full ring assertion in fabric stream shard replacements
- #1925: Support list for docid when using `couch_db:purge_docs/3`.
- #1642: `io_priority` is now set properly on view update and compaction processes.
- #1865: Purge now supports >100 document IDs in a single request.
- #1861: The `vm.args` file has improved commentary.
- #1808: Pass document update type for additional checks in `before_doc_update`.
- #1835: Module lists are no longer hardcoded in `.app` files.
- #1798, #1933: Multiple compilation warnings were eliminated.
- #1826: The `couch_replicator_manager` shim has been fully removed.
- #1820: After restarting CouchDB, JS and Elixir tests now wait up to 30s for it to be ready before timing out.
- #1800: `make elixir` supports specifying individual tests to run with `tests=`.
- #1805: `dev/run` supports `--with-haproxy` again.
- #1774: `dev/run` now supports more than 3 nodes.
- #1779: Refactor Elixir test suite initialization.
- #1769: The Elixir test suite uses Credo for static analysis.
- #1776: All Python code is now formatted using Python black.
- #1786: `dev/run`: do not create needless `dev/data/` directory.
- #2482: A redundant `get_ring_opts` call has been removed from `dreyfus_fabric_search`.
- #2506: CouchDB's release candidates no longer propagate the RC tags into each Erlang application's version string.
- #2511: recon, the Erlang diagnostic toolkit, has been added to CouchDB's build process and ships in the release + convenience binaries.
- Fauxton updated to v1.2.3, which includes:
  - Support multiple server-generated warnings when running queries

  - Partitioned database support
  - Search index support
  - Remove references to deprecated dbinfo fields
  - Improve accessibility for screen readers
  - Numerous CSS fixes
- Improved test cases:
  - Many, many test race conditions and bugs have been removed (PR list too long to include here!)
  - More test cases were ported to Elixir, including:
    * Cluster with and without quorum tests (#1812)
    * `delayed_commits` (#1796)
    * `multiple_rows` (#1958)
    * `invalid_docids` (#1968)
    * `replication` (#2090)
    * All `attachment_*` tests (#1999)
    * `copy_doc` (#2000)
    * `attachments` (#1953)
    * `erlang_views` (#2237)
    * `auth_cache`, `cookie_auth`, `lorem*`, `multiple_rows`, `users_db`, `utf8` (#2394)
    * `etags_head` (#2464, #2469)
  - #2431: `chttpd_purge_tests` have been improved in light of CI failures.
  - #2432: Address flaky test failure on `t_invalid_view/1`.
  - #2363: Elixir tests now run against a single node cluster, in line with the original design of the JavaScript test suite. This is a permanent change.
  - #1893: Add "w:3" for lots of doc tests.
  - #1939, #1931: Multiple fixes to improve support in constrained CI environments.
  - #2346: Big-endian support for the `couch_compress` tests.
  - #2314: Do not auto-index when testing `update=false` in Mango.
  - #2141: Fix `couch_views` encoding test.
  - #2123: Timeout added for `fold_docs-with_different_keys` test.
  - #2114: EUnit tests now correctly inherit necessary environment variables.
  - #2122: *:meck.unload()* is now called automatically after every test.
  - #2098: Fix `cpse_test_purge_replication` eunit test.
  - #2085, #2086: Fix a flaky `mem3_sync_event_listener` test.
  - #2084: Increase timeouts on two slow btree tests.
  - #1960, #1961: Fix for `chttpd_socket_buffer_size_test`.
  - #1922: Tests added for shard splitting functionality.
  - #1869: New test added for doc reads with etag `If-None-Match` header.
  - #1831: Re-introduced *cpse_test_purge_seqs* test.
  - #1790: Reorganise `couch_flag_config_tests` into a proper suite.

  - #1785: Use `devclean` on elixir target for consistency of Makefile.
  - #2476: For testing, `Triq` has been replaced with `PropEr` as an optional dependency.
- External dependency updates:
  - #1870: Mochiweb has been updated to 2.19.0.
  - #1938: Folsom has been updated to 0.8.3.
  - #2001: ibrowse has been updated to 4.0.1-1.
  - #2400: jiffy has been updated to 1.0.1.
- A llama! OK, no, not really. If you got this far…thank you for reading.

16.5 2.3.x Branch

- *Upgrade Notes*
- *Version 2.3.1*
- *Version 2.3.0*

16.5.1 Upgrade Notes

- #1602: To improve security, there have been major changes in the configuration of query servers, SSL support, and HTTP global handlers:

  1. Query servers

  Query servers are NO LONGER DEFINED in the .ini files, and can no longer be altered at runtime.

  The JavaScript and CoffeeScript query servers continue to be enabled by default. Setup differences have been moved from default.ini to the `couchdb` and `couchdb.cmd` start scripts respectively.

  Additional query servers can now be configured using environment variables:

  ```
  export COUCHDB_QUERY_SERVER_PYTHON="/path/to/python/query/server.py with␣
  ↪args"
  couchdb
  ```

  where the last segment in the environment variable (`_PYTHON`) matches the usual lowercase(!) query language in the design doc `language` field (here, `python`.)

  Multiple query servers can be configured by using more environment variables.

  You can also override the default servers if you need to set command- line options (such as `couchjs` stack size):

  ```
  export COUCHDB_QUERY_SERVER_JAVASCRIPT="/path/to/couchjs /path/to/main.
  ↪js -S <STACKSIZE>"
  couchdb
  ```

  2. Native Query Servers

  The mango query server continues to be enabled by default. The Erlang query server continues to be disabled by default. This change adds a `[native_query_servers] enable_erlang_query_server = BOOL` setting (defaults to `false`) to enable the Erlang query server.

If the legacy configuration for enabling the query server is detected, that is counted as a `true` setting as well, so existing configurations continue to work just fine.

3. SSL Support

Enabling SSL support in the ini file is now easier:

```
[ssl]
enable = true
```

If the legacy httpsd configuration is found in your ini file, this will still enable SSL support, so existing configurations do not need to be changed.

4. HTTP global handlers

These are no longer defined in the default.ini file, but have been moved to the couch.app context. If you need to customize your handlers, you can modify the app context using a couchdb.config file as usual.

- #1602: Also to improve security, the deprecated `os_daemons` and `couch_httpd_proxy` functionality has been completely removed ahead of the planned CouchDB 3.0 release. We recommend the use of OS-level daemons such as runit, sysvinit, systemd, upstart, etc. to launch and maintain OS daemons instead, and the use of a reverse proxy server in front of CouchDB (such as haproxy) to proxy access to other services or domains alongside CouchDB.
- #1543: The node-local (default port 5986) `/_restart` endpoint has been replaced by the clustered (default port 5984) endpoint `/_node/{node-name}/_restart` and `/_node/_local/_restart` endpoints. The node-local endpoint has been removed.
- #1764: All python scripts shipped with CouchDB, including `couchup` and the `dev/run` development cluster script, now specify and require Python 3.x.
- #1396: CouchDB is now compatible with Erlang 21.x.
- #1680: The embedded version of `rebar` used to build CouchDB has been updated to the last version of `rebar2` available. This assists in building on non-x86 platforms.
- #1857: Refuse building with known bad versions of Erlang.

16.5.2 Version 2.3.1

Features

- #1811: Add new `/{db}/_sync_shards` endpoint (admin-only).
- #1870: Update to mochiweb 2.19.0. See also #1875.
- #1857: Refuse building with known bad versions of Erlang.
- #1880: Compaction: Add snooze_period_ms for finer tuning.

Bugfixes

- #1795: Filter out empty missing_revs results in `mem3_rep`.
- #1384: Fix `function_clause` error on invalid DB `_security` objects.
- #1841: Fix `end_time` field in `/_replicate` response.
- #1860: Fix read repair in a mixed cluster environment.
- #1862: Fix `fabric_open_doc_revs`.
- #1865: Support purge requests with more than 100 doc ids.
- #1867: Fix timeout in `chttpd_purge_tests`.

- #1766: Add default fabric request timeouts.
- #1810: Requests return 400 Bad Request when URL length exceeds 1460 characters. See #1870 for details.
- #1799: Restrict `_purge` to server admin.
- #1874: This fixes inability to set keys with regex symbols in them.
- #1901: Fix badarg crash on invalid rev for individual doc update.
- #1897: Fix `from_json_obj_validate` crash when provided rev isn't a valid hex.
- #1803: Use the same salt for admin passwords on cluster setup.
- #1053: Fix python2 compatibility for `couchup`.
- #1905: Fix python3 compatibility for `couchup`.

16.5.3 Version 2.3.0

Features

- (Multiple) Clustered purge is now available. This feature restores the CouchDB 1.x ability to completely remove any record of a document from a database. Conditions apply; to use the feature safely, and for full details, read the complete *Clustered Purge* documentation.
- #1658: A new config setting is available, allowing an administrator to configure an initial list of nodes that should be contacted when a node boots up. Nodes in the `seedlist` that are successfully reached will be added to that node's `_nodes` database automatically, triggering a distributed Erlang connection and replication of the internal system databases to the new node. This can be used instead of manual config or the cluster setup wizard to bootstrap a cluster. The progress of the initial seeding of new nodes is exposed at the `GET /_up` endpoint.
- Replication supports ipv6-only peers after updating ibrowse dependency.
- #1708: The UUID of the server/cluster is once again exposed in the `GET /` response. This was a regression from CouchDB 1.x.
- #1722: Stats counts between job runs of the replicator are no longer reset on job restart.
- #1195, #1742: CouchDB's `_bulk_get` implementation now supports the `multipart/mixed` and `multipart/related` content types if requested, extending compatibility with third-party replication clients.

Performance

- #1409: CouchDB no longer forces the TCP receive buffer to a fixed size of 256KB, allowing the operating system to dynamically adjust the buffer size. This can lead to significantly improved network performance when transferring large attachments.
- #1423: Mango selector matching now occurs at the shard level, reducing the network traffic within a cluster for a mango query.
- #1423: Long running operations at the node level could exceed the inter-node timeout, leading to a fabric timeout error in the logfile and a cancellation of the task. Nodes can now ping to stop that from happening.
- #1560: An optimization to how external data sizes of attachments were recorded was made.
- #1586: When cleaning up outdated secondary index files, the search is limited to the index directory of a specific database.
- #1593: The `couch_server` ETS table now has the `read_concurrency` option set, improving access to the global list of open database handles.
- #1593: Messages to update the least-recently used (LRU) cache are not sent when the `[couchdb] update_lru_on_read` setting is disabled.

- #1625: All nodes in a cluster now run their own `rexi` server.

Bugfixes

- #1484: `_stats` now correctly handles the case where a map function emits an array of integers. This bug was introduced in 2.2.0.
- #1544: Certain list functions could return a `render_error` error intermittently.
- #1550: Replicator `_session` support was incompatible with CouchDB installations using the `require_valid_user = true` setting.
- #1571: Under very heavy load, it was possible that `rexi_server` could die in such a way that it's never restarted, leaving a cluster without the ability to issue RPC calls - effectively rendering the cluster useless.
- #1574: The built-in `_sum` reduce function has been improved to check if the objects being summed are not overflowing the view storage. Previously, there was no protection for `_sum`-introduced overflows.
- #1582: Database creation parameters now have improved validation, giving a more readable error on invalid input.
- #1588: A missing security check has been restored for the noop `/db/_ensure_full_commit` call to restore database validation checks.
- #1591: CouchDB now creates missing shard files when accessing a database if necessary. This handles the situation when, on database creation, no nodes were capable of creating any of the shard files required for that database.
- #1568: CouchDB now logs a warning if a changes feed is rewound to 0. This can help diagnose problems in busy or malfunctioning clusters.
- #1596: It is no longer possible that a busy `couch_server`, under a specific ordering and timing of events, will incorrectly track `open_async` messages in its mailbox.
- #1601, #1654: CouchDB now logs better when an error causes it to read past the EOF of a database shard. The check for whether CouchDB is trying to read too many bytes has been correctly separated out from the error indicating it has attempted to read past the EOF.
- #1613: Local nodes are now filtered out during read repair operations.
- #1636: A memory leak when replicating over HTTPS and a problem occurs has been squashed.
- #1635: `/_replicate` jobs are no longer restarted if parameters haven't changed.
- #1612: JavaScript rewrite functions now send the body of the request to the rewritten endpoint.
- #1631: The replicator no longer crashes if the user has placed an invalid VDU function into one of the `_replicator` databases.
- #1644, #1647: It is no longer possible to create illegally-named databases within the reserved system space (_ prefix.)
- #1650: `_bulk_get` is once again operational for system databases such as `_users`.
- #1652: Access to `/_active_tasks` is once again restricted to server admins only.
- #1662: The `couch_log` application no longer crashes when new, additional information is supplied by a crashing application, or when any of its own children are restarted.
- #1666: Mango could return an error that would crash the `couch_query_servers` application. This is no longer the case.
- #1655: Configuration of `ets_lru` in `chttpd` now performs proper error checking of the specified config value.
- #1667: The `snappy` dependency has been updated to fix a memory allocation error.

- #1683: Attempting to create a local document with an invalid revision no longer throws a `badarg` exception. Also, when setting `new_edits` to `false` and performing a bulk write operation, local documents are no longer written into the wrong btree. Finally, it is no longer possible to create a document with an empty ID during a bulk operation with `new_edits` set to `false`.
- #1721: The `couchup` convenience script for upgrading from CouchDB 1.x now also copies a database's `_security` object on migration.
- #1672: When checking the status of a view compaction immediately after starting it, the `total_changes` and `changes_done` fields are now immediately populated with valid values.
- #1717: If the `.ini` config file is read only, an attempt to update the config through the HTTP API will now result in a proper `eacces` error response.
- #1603: CouchDB now returns the correct `total_rows` result when querying `/{db}/_design_docs`.
- #1629: Internal load validation functions no longer incorrectly hold open a deleted database or its host process.
- #1746: Server admins defined in the ini file accessing via HTTP API no longer result in the auth cache logging the access as a miss in the statistics.
- #1607: The replicator no longer fails to re-authenticate to open a remote database when its session cookie times out due to a VDU function forbidding writes or a non-standard cookie expiration duration.
- #1579: The compaction daemon no longer incorrectly only compacts a single view shard for databases with a `q` value greater than 1.
- #1737: CouchDB 2.x now performs as well as 1.x when using a `_doc_ids` or `_design_docs` filter on a changes feed.

Mango

Other

The 2.3.0 release also includes the following minor improvements:

- Improved test cases:
  - The Elixir test suite has been merged. These test cases are intended to replace the aging, unmaintainable JavaScript test suite, and help reduce our dependency on Mozilla Spidermonkey 1.8.5. The test suite does not yet cover all of the tests that the JS test suite does. Once it achieves full coverage, the JS test suite will be removed.
  - Many racy test cases improved for reliable CI runs.
  - The Makefile targets for `list-eunit-*` now work correctly on macOS.
  - #1732, #1733, #1736: All of the test suites run and pass on the Windows platform once again.
- #1597: Off-heap messages, a new feature in Erlang 19+, can now be disabled per module if desired.
- #1682: A new `[feature_flags]` config section exists for the purpose of enabling or disabling experimental features by CouchDB developers.
- A narwhal! OK, no, not really. If you got this far... thank you for reading.

16.6 2.2.x Branch

- *Upgrade Notes*
- *Version 2.2.0*

16.6.1 Upgrade Notes

- The minimum supported version of Erlang is now 17, not R16B03. Support for Erlang 21 is still ongoing and will be provided in a future release.
- The CouchDB replication client can now use the `/_session` endpoint when authenticating against remote CouchDB instances, improving performance since re-authorization does not have to be performed with every request. Because of this performance improvement, it is recommended to increase the PBKDF2 work factor beyond the default `10` to a modern default such as `10000`. This is done via the local ini file setting `[couch_httpd_auth] iterations = 10000`.

  Do **not** do this if an older version of CouchDB is replicating TO this instance or cluster regularly, since CouchDB < 2.2.0 must perform authentication on every request and replication performance will suffer.

  A future version will make this increased number of iterations a default.
- #820, #1032: Multiple queries can now be made at the `POST /{db}/_all_docs/queries`, `POST /{db}/_design_docs/queries` and `POST /{db}/_local_docs/queries` endpoints. Also, a new endpoint `POST /{db}/_design/{ddoc}/_view/{view}/queries` has been introduced to replace the `?queries` parameter formerly provided for making multiple queries to a view. The old `?queries` parameter *is now deprecated and will be removed in a future release of CouchDB.*
- The maximum http request limit, which had been lowered in 2.1.0, has been re-raised to a 4GB limit for now. (#1446). Ongoing discussion about the path forward for future releases is available in #1200 and #1253.
- #1118: The least recently used (LRU) cache of databases is now only updated on database write, not read. This has lead to significant performance enhancements on very busy clusters. To restore the previous behaviour, your local ini file can contain the block `[couchdb] update_lru_on_read = true`.
- #1153: The CouchDB replicator can now make use of the `/_session` endpoint rather than relying entirely on HTTP basic authentication headers. This can greatly improve replication performance. We encourage you to upgrade any nodes or clusters that regularly act as replication clients to use this new feature, which is enabled by default (#1462).
- #1283: The `[couchdb] enable_database_recovery` feature, which only soft-deletes databases in response to a `DELETE /{db}` call, is now documented in `default.ini`.
- #1330: CouchDB externals and OS daemons are now officially deprecated and no longer documented. Support for these features will be completely removed in a future release of CouchDB (probably 3.0.0).
- #1436: CouchDB proxy authentication now uses a proper `chttpd_auth` module, simplifying configuration in local ini files. While this is not a backward- compatible breaking change, it is best to update your local ini files to reference the new `{chttpd_auth, proxy_authentication_handler}` handler rather than the `couch_httpd_auth` version, as `couch_httpd` is in the process of being deprecated completely.
- #1476, #1477: The obsolete *update_notification* feature, which was replaced by */{db}/_changes* feeds c. CouchDB 1.2, has been completely removed. This feature never worked in 2.0 for databases, only for shards, making it effectively useless.

16.6.2 Version 2.2.0

Features

- Much improved documentation. Highlights include:
  - A complete rewrite of the *sharding* documentation.
  - Developer installation notes (`INSTALL.*.rst`)
  - Much of the content of the original CouchDB Wiki has been imported into the official docs. (The old CouchDB Wiki is in the process of being deprecated.)
- Much improved Fauxton functionality. Highlights include:
  - Search support in the code editor
  - Support for relative Fauxton URLs (*i.e.*, not always at `/_utils`)
  - Replication setup enhancements for various authentication mechanisms
  - Fixes for IE10, IE11, and Edge (we hope…)
  - Resolving conflicts of design documents is now allowed
- #496, COUCHDB-3287: New pluggable storage engine framework has landed in CouchDB. This internal refactor makes it possible for CouchDB to use different backends for storing the base database file itself. The refactor included a full migration of the existing "legacy" storage engine into the new framework.
- #603: When creating a new database on a cluster without quorum, CouchDB will now return a `202 Accepted` code if possible, indicating that at least one node has written the database record to disk, and that other nodes will be updated as they return to an online state. This replaces the former `500` internal error.
- #1136, #1139: When deleting a database in a cluster without quorum, CouchDB will no longer throw a `500` error status, but a `202` as long as at least one node records the deletion, or a `200` when all nodes respond. This fix parallels the one made for #603.
- #745: CouchDB no longer fails to complete replicating databases with large attachments. The fix for this issue included several related changes:
  - The maximum http request limit, which had been lowered in 2.1.0, has been re-raised to a 4GB limit for now. (#1446). Ongoing discussion about the path forward for future releases is available in #1200 and #1253.
  - An update to the replicator http client that improves active socket accounting, without which CouchDB can cease to be responsive over the main http interface (#1117)
  - The replicator's http client no longer performs unconditional retries on failure (#1177)
  - A path by which CouchDB could lose track of their RPC workers during multipart attachment processing was removed. (#1178)
  - When CouchDB transmits a `413 Payload Too Large` response on attachment upload, it now correctly flushes the receive socket before closing the connection to avoid a TCP reset, and to give the client a better chance of parsing the 413 response. In tandem, the replicator http client correctly closes its own socket after processing any 413 response. (#1234)
  - A `fabric` process to receive unchunked attachments can no longer orphan processes that leave unprocessed binaries in memory until all available memory is exhausted. (#1264).
  - When using CouchDB's native SSL responder (port 6984 by default), sessions are now timed out by default after 300s. This is to work around RAM explosion in the BEAM VM when using the Erlang-native SSL libraries. (#1321
- #822: A new end point */_dbs_info* has been added to return information about a list of specified databases. This endpoint can take the place of multiple queries to `/{db}`.

- #875, #1030: `couch_peruser` installations can now specify a default `q` value for each peruser-created database that is different from the cluster's `q` value. Set this in your local ini file, under `[couch_peruser] q`.
- #876, #1068: The `couch_peruser` database prefix is now configurable through your local ini file, under `[couch_peruser] database_prefix`.
- #887: Replicator documents can now include parameters for target database creation, such as `"create_target_params": {"q": "1"}`. This can assist in database resharding or placement.
- #977: When using `COPY` to copy a document, CouchDB no longer fails if the new ID includes Unicode characters.
- #1095: Recognize the environment variables `ARGS_FILE`, `SYSCONFIG_FILE`, `COUCHDB_ARGS_FILE` and `COUCHDB_SYSCONFIG_FILE` to override where CouchDB looks for the `vm.args` and `sys.config` files at startup.
- #1101, #1425: Mango can now be used to find conflicted documents in a database by adding `conflicts: true` to a mango selector.
- #1126: When queried back after saving, replication documents no longer contain sensitive credential information (such as basic authenticataion headers).
- #1203:
  - The compaction daemon now has a snooze period, during which it waits to start the next compaction after finishing the previous one. This value is useful in setups with many databases (e.g. with `couch_peruser`) or many design docs, which can cause a CPU spike every `check_interval` seconds. The setting can be adjusted in your local ini file via `[compaction_daemon] snooze_period`. The current default is a 3 second pause.
  - The `check_interval` has been raised from 300 seconds to 3600 seconds.
  - A `notice`-level log about closing view indexes has been demoted to the `debug` level. In a sceario with many design docs, this would createsignficant load on the logging subsystem every `[compaction_daemon] check_interval` for no discernible benefit.
- #1309, #1435: CouchDB now reports the git sha at the time of build in the top-level `GET /` version string, in a new `git_sha` key. This can be used to help ensure an unmodified version of CouchDB has been built and is running on any given machine.
- COUCHDB-2971, #1346: CouchDB now includes a new builtin reduce function `_approx_count_distinct`, that uses a HyperLogLog algorithm to estimate the number of distinct keys in the view index. The precision is currently fixed to 2^11 observables, and therefore uses approximately 1.5KB of memory.
- #1377: CouchDB finalization of view reduces now occurs at the coordinator node. This simplified the built-in `_stats` function.
- #1392: When running CouchDB under Erlang 19.0 or newer, messages can now be stored off the process heap. This is extremely useful for Erlang processes that can have huge number of messages in their mailbox, and is now enabled for `couch_server`, `couch_log_server`, `ddoc_cache`, `mem3_shards`, and `rexi_server` whenever possible.
- #1424: The CouchDB native SSL/TLS server `httpsd` now accepts socket-level configuration options through the `[httpsd] server_options` ini file setting.
- #1440: CouchDB can now be configured to prevent non-admins from accessing the `GET /_all_dbs` method by specifying `[chttpd] admin_only_all_dbs = true` in your local ini file(s). The `true` setting will become default in future versions.
- #1171, #1445: CouchDB can now be configured to use the internal Erlang MD5 hash function when not available in the external environment (e.g. FIPS enabled CentOS) at compile time with the `configure` flag `--enable-md5`. Because this implementation is slower, it is not recommended in the general case.

Performance

- #958: The revision stemming algorithm was optimized down from *O(N^2)* to *O(N)* via a depth-first search approach, and then further improved by calling the stemming operation only when necessary. This new algorithm can be disabled by setting the option `[couchdb] stem_interactive_updates = false` if necessary.
- #1246: CouchDB now checks for request authorization only once per each database request, improving the performance of any request that requires authorization.

Bugfixes

- #832, #1064: Tracking of Couch logging stats has been added back into the per-node `/_node/<node-name>/_stats` endpoint.
- #953, #973: Return `404 Not Found` on `GET /_scheduler`, not `405 Method Not Allowed`.
- #955: The `/{db}/_bulk_docs` endpoint now correctly responds with a `400 Bad Request` error if the `new_edits` parameter is not a boolean.
- #969: CouchDB now returns `offset` and `update_seq` values when `keys` are provided to the `GET` or `POST /{db}/_all_docs?update_seq=true` endpoints. This was affecting PouchDB compatibility.
- #984, #1434: CouchDB views now retain their `update_seq` after compaction, preventing potentially expensive client-side view rewinds after compaction.
- #1012: Address a theoretical race condition the replication scheduler could encounter when trying to determine if the cluster is "stable" enough to resume handling replication-introduced document updates.
- #1051: Return a user-friendly error message when attempting to create a CouchDB user with an invalid password field (non-string).
- #1059: DB-specific compaction configurations were not working correctly. The syntax now also supports shard-level custom compaction configuration if desired (which it probably isn't.)
- #1097: Compaction daemon will not crash out when trying to check specific file system mounts that are not "real" file systems (like `/run` on Linux).
- #1198: Fauxton is no longer available on the node-local port (5986, by default). The node-local port is only to be used for specific administrative tasks; removing the Fauxton interface prevents mistaking the node-local port as the correct CouchDB port (5984, by default).
- #1165: `validate_doc_update` view functions can once again be implemented directly in Erlang (after enabling the optional Erlang view server).
- #1223: The `couch_config` application now correctly handles non-persistent integer and boolean-valued configuration changes.
- #1242: `couch_os_daemons` may now reside in directories with spaces.
- #1258: CouchDB will now successfully login users, even if password encryption is very slow.
- #1276: The replication scheduler status for a repeatedly erroring job now correctly reflects the *crashing* state in more scenarios.
- #1375: If CouchDB fails authorization but passes authentication, it no longer drops the `user_ctx` out of the request.
- #1390: The active size of views (as returned in a database info response) no longer is incorrectly calculated in such a way that it could occasionally be larger than the actual on-disk file size.
- #1401: CouchDB Erlang views no longer crash in the `couch_native` process with an unexpected `function_clause` error.
- #1419: When deleting a file, CouchDB now properly ignores the configuration flag `enable_database_recovery` when set when compacting databases, rather than always retaining the old, renamed, uncompacted database file.

- #1439: The CouchDB setup wizard now correctly validates bind_addresses. It also no longer logs credentials by moving logging of internal wizard setup steps to the `debug` level from the `notice` level.

Mango

- #816, #962, #1038: If a user specifies a value for `use_index` that is not valid for the selector (does not meet coverage requirements or proper sort fields), attempt to fall back to a valid index or full DB scan rather than returning a `400`. If we fall back, populate a `warning` field in the response. Mango also tries to use indexes where `$or` may select a field only when certain values are present.
- #849: When `{"seq_indexed": true}` is specified, a badmatch error was returned. This is now fixed.
- #927, #1310: Error messages when attempting to sort incorrectly are now actually useful.
- #951: When using `GET /{db}/_index`, only use a partial filter selector for an index if it is set to something other than the default.
- #961: Do not prefix `_design/` to a Mango index name whose user-specified name already starts with `_design/`.
- #988, #989: When specifying a `use_index` value with an invalid index, correctly return a `400 Bad Request` showing that the requested index is invalid for the request specified.
- #998: The fix for *CVE 2017-12635* presented a breaking change to Mango's `/{db}/_find`, which would evaluate all instances of all JSON fields in a selector. Mango is now tested to ensure it only considers the last instance of a field, silently ignoring those that appear before it.
- #1014: Correctly deduce list of indexed fields in a selector when nested `$and` operators are specified.
- #1023: Fix an unexpected `500` error if `startkey` and `endkey` in a Mango selector were reversed.
- #1067: Prevent an `invalid_cast` crash when the `couch_proc_manager` soft limit for processes is reached and mango idle processes are stopped.
- #1336: The built-in fields `_id` and `rev` will always be covered by any index, and Mango now correctly ignores their presence in any index that explicitly includes them for selector matching purposes.
- #1376: Mango now appropriately selects some indexes as usable for queries, even if not all columns for an index are added to the query's sort field list.
- Multiple fixes related to using Mango as a front-end for full text indexing (a feature not shipped with couch, but for which support is in place as a compile-time addon).

Other

The 2.2.0 release also includes the following minor improvements:

- Developers can, at build time, enable curl libraries & disable Fauxton and documentation builds by specifying the new `--dev` option to the `configure` script.
- The `mochiweb` dependency was bumped to version 2.17.0, in part to address the difficult #745 issue.
- Improved compatibility with newer versions of Erlang (20.x)
- Improved release process for CouchDB maintainers and PMC members.
- Multiple test suite improvements, focused on increased coverage, speed, and reliability.
- Improvements to the Travis CI and Jenkins CI setups, focused on improved long-term project maintenance and automatability.
- Related improvements to the CouchDB deb/rpm packaging and Docker repositories to make deployment even easier.
- #1007: Move `etc/default.ini` entries back into `[replicator]` section (incorrectly moved to `[couch_peruser]` section)

- #1245: Increased debug-level logging for shard open errors is now available.
- #1296: CouchDB by default now always invokes the SMP-enabled BEAM VM, even on single-processor machines. A future release of Erlang will remove the non-SMP BEAM VM entirely.
- A pony! OK, no, not really. If you got this far…thank you for reading.

16.7 2.1.x Branch

- *Upgrade Notes*
- *Version 2.1.2*
- *Version 2.1.1*
- *Version 2.1.0*
- *Fixed Issues*

16.7.1 Upgrade Notes

- When upgrading from 2.x to 2.1.1, if you have not customized your node name in `vm.args`, be sure to retain your original `vm.args` file. The default node name has changed from `couchdb@localhost` to `couchdb@127.0.0.1`, which can prevent CouchDB from accessing existing databases on the system. You may also change the name option back to the old value by setting `-name couchdb@localhost` in `etc/vm.args` by hand. The default has changed to meet new guidelines and to provide additional functionality in the future.

  If you receive errors in the logfile, such as `internal_server_error : No DB shards could be opened.` or in Fauxton, such as `This database failed to load.` you need to make this change.
- The deprecated (and broken) OAuth 1.0 implementation has been removed.
- If user code reads or manipulates replicator document states, consider using the `[replicator] update_docs = true` compatibility parameter. In that case the replicator will continue updating documents with transient replication states. However, that will incur a performance cost. Consider instead using the `_scheduler/docs` HTTP endpoint.
- The `stale` parameter for views and `_find` has been deprecated in favour of two new parameters: `stable` and `update`. The old `stale=ok` behaviour is equivalent to `stable=true&update=false`, and the old `stale=update_after` behaviour is equivalent to `stable=true&update=lazy`. The deprecated `stale` parameter will be removed in CouchDB 3.0.
- The new `:httpd/max_http_request_size` configuration parameter was added. This has the same behavior as the old *`couchdb/max_document_size`* configuration parameter, which had been unfortunately misnamed, and has now been updated to behave as the name would suggest. Both are documented in the shipped `default.ini` file.

  Note that the default for this new parameter is 64MB instead of 4GB. If you get errors when trying to PUT or POST and see HTTP 413 return codes in couchdb logs, this could be the culprit. This can affect couchup in-place upgrades as well.
- #914: Certain critical config sections are blacklisted from being modified through the HTTP API. These sections can still be modified through the standard `local.ini` or `local.d/*.ini` files.
- #916: `couchjs` now disables `eval()` and the `Function()` constructor by default. To restore the original behaviour, add the `--eval` flag to the definition of the javascript query server in your `local.ini` file.

16.7.2 Version 2.1.2

Security

- *CVE 2018-8007*

16.7.3 Version 2.1.1

Security

- *CVE 2017-12635*
- *CVE 2017-12636*

General

- #617: CouchDB now supports compilation and running under Erlang/OTP 20.x.
- #756: The `couch_peruser` functionality is now *really* fixed. Really.
- #827: The cookie domain for AuthSession cookies, used in a proxy authentication configuration, can now be customized via the ini file.
- #858: It is now possible to modify shard maps for system databases.
- #732: Due to an Erlang bug (ERL-343), invalid paths can be returned if volumes are mounted containing whitespace in their name. This problem surfaced primarily on macOS (Time Machine volumes). CouchDB now works around this bug in unpatched versions of Erlang by skipping the free space check performed by the compaction daemon. Erlang itself will correctly perform free space checks in version 21.0.
- #824: The current node's local interface can now be accessed at `/_node/_local/{endpoint}` as well as at `/_node/<nodename>@<hostname>/{endpoint}`.
- The Dockerfile in the source repository has been retired. For a current Dockerfile, see the *couchdb-docker repository*.
- Fauxton now uses a version of React with a BSD license.

Performance

- #835: CouchDB now no longer decompresses documents just to determine their uncompressed size. In tests, this has lead to improvements between 10-40% in both CPU and wall-clock time for database compaction.
- The design document cache (`ddoc_cache`) has been rewritten to improve performance.

Mango

- #808: Mango now supports *partial indexes*. Partial indexes allow documents to be filtered at indexing time, potentially offering significant performance improvements for query selectors that don't map cleanly to a range query on an index.
- #740: Mango queries can now be paginated. Each query response includes a bookmark. The bookmark can be provided on a subsequent query to continue from a specific key.
- #768: Mango `_find` accepts an `execution_stats` parameter. If present, a new object is included in the response which contains information about the query executed. The object contains the count of total keys examined (0 for json indexes), total documents examined (when `include_docs=true` is used), and the total quorum documents examined (when fabric doc lookups are used).

- #816 and #866: Mango now requires that all of the fields in a candidate index must exist in a query's selector. Previously, this check was incorrect, and indexes that might only contain a subset of valid documents might be selected by the query planner if no explicit index was specified at query time. Further, if a sort field is specified at query time, that field needs to exist (but could be null) in the results returned.

Other

The 2.1.1 release also includes the following minor improvements:

- #635: Stop couch_index processes on ddoc update
- #721: Save migrated replicator checkpoint documents immediately
- #688: Reuse http-based replication checkpoints when upgrading to https
- #729: Recommend the use only of `-name` and not `-sname` in *vm.args* for compatibility.
- #738: Allow replicator application to always update replicator docs.
- #605: Add `Prefer: return=minimal` header options from RFC7240 to reduce the number of headers in the response.
- #744: Allow a 503 response to be returned to clients (with metric support)
- #746: Log additional information on crashes from rexi
- #752: Allow Mango $in queries without requiring the index to use an array
- (multiple) Additional debugging utilities have been added.
- (multiple) Hot code upgrades from 2.0 -> 2.1.1 are now possible.
- (multiple) Improvements to the test suite have been made.
- #765: Mango `_explain` now includes view parameters as requested by the user.
- #653: *_show* and *_list* should now work for admin-only databases such as `_users`.
- #807: Mango index selection should occur only once.
- #804: Unhandled Mango errors are now logged.
- #659: Improve accuracy of the `max_document_size` check.
- #817: Invalid Base64 in inline attachments is now caught.
- #825: Replication IDs no longer need to be URL encoded when using the `_scheduler/jobs/<job_id>` endpoint.
- #838: Do not buffer rexi messages to disconnected nodes.
- #830: The stats collection interval is now configurable in an ini file, not in the application context. The default value is 10, and the setting is reloaded every 600 seconds.
- #812: The `/{db}` endpoint now includes a `cluster` block with the database's `q`, `n`, and default `w` and `r` values. This supplements the existing `/{db}/_shards` and `/{db}/_shards/{id}` detailed information on sharding and quorum.
- #810: The replicator scheduler crashed counter gauge more reliably detects replication crashes by reducing the default number of retries from 10 to 5 (reducing the duration from 4 mins to 8 secs).
- COUCHDB-3288: Tolerate mixed clusters for the upcoming pluggable storage engine work.
- #839: Mango python tests now support Python 3 as well as 2.
- #845: A convenience `remsh` script has been added to support live debugging of running systems.
- #846: Replicator logging is now less verbose and more informative when replication terminates unexpectedly.

- #797: Reduce overflow errors are now returned to the client, allowing views with a single bad reduce to build while not exhausting the server's RAM usage.
- #881: Mango now allows match on documents where the indexed value is an object if a range query is issued. Previously, query results might change in the presence of an index, and operators/selectors which explicitly depend on a full index scan (such as `$exists`) would not return a complete result set.
- #883: Erlang time module compatibility has been improved for releases of Erlang newer than 18.0.
- #933: 410 is now returned when attempting to make a temporary view request.
- #934: The replicator now has a configurable delay before retrying to retrieve a document after receiving a `missing_doc` error.
- #936: jiffy now deduplicates JSON keys.

16.7.4 Version 2.1.0

- The Mango `_find` endpoint supports a new combination operator, `$allMatch`, which matches and returns all documents that contain an array field with all its elements matching all the specified query criteria.
- New scheduling replicator. The core of the new replicator is a scheduler which allows running a large number of replication jobs by switching between them, stopping some and starting others periodically. Jobs which fail are backed off exponentially. There is also an improved inspection and querying API: `_scheduler/jobs` and `_scheduler/docs`:
  - `_scheduler/jobs` : This endpoint shows active replication jobs. These are jobs managed by the scheduler. Some of them might be running, some might be waiting to run, or backed off (penalized) because they crashed too many times. Semantically this is somewhat equivalent to `_active_tasks` but focuses only on replications. Jobs which have completed or which were never created because of malformed replication documents will not be shown here as they are not managed by the scheduler. `_replicate` replications, started form _replicate endpoint not from a document in a `_replicator` db, will also show up here.
  - `_scheduler/docs` : This endpoint is an improvement on having to go back and read replication documents to query their state. It represents the state of all the replications started from documents in _replicator db. Unlike `_scheduler/jobs` it will also show jobs which have failed or have completed.

  By default, scheduling replicator will not update documents with transient states like `triggered` or `error` anymore, instead `_scheduler/docs` API should be used to query replication document states.

Other scheduling replicator improvements

- Network resource usage and performance was improved by implementing a shared connection pool. This should help in cases of a large number of connections to the same sources or target. Previously connection pools were shared only within a single replication job.
- Improved request rate limit handling. Replicator requests will auto-discover rate limit capacity on targets and sources based on a proven Additive Increase / Multiplicative Decrease feedback control algorithm.
- Improved performance by having exponential backoff for all replication jobs failures. Previously there were some scenarios were failure led to continuous repeated retries, consuming CPU and disk resources in the process.
- Improved recovery from long but temporary network failure. Currently if replications jobs fail to start 10 times in a row, they will not be retried anymore. This is sometimes desirable, but in some cases, for example, after a sustained DNS failure which eventually recovers, replications reach their retry limit, stop retrying and never recover. Previously it required user intervention to continue. Scheduling replicator will never give up retrying a valid scheduled replication job and so it should recover automatically.
- Better handling of filtered replications. Failing user filter code fetches from the source will not block replicator manager and stall other replications. Failing filter fetches will also be backed off exponentially. Another

improvement is when filter code changes on the source, a running replication will detect that and restart itself with a new replication ID automatically.

The 2.1.0 release also includes the following minor improvements:

- COUCHDB-1946: Hibernate couch_stream after each write (up to 70% reduction in memory usage during replication of DBs with large attachments)
- COUCHDB-2964: Investigate switching replicator manager change feeds to using "normal" instead of "long-poll"
- COUCHDB-2988: (mango) Allow query selector as changes and replication filter
- COUCHDB-2992: Add additional support for document size
- COUCHDB-3046: Improve reduce function overflow protection
- COUCHDB-3061: Use vectored reads to search for buried headers in .couch files. "On a modern linux system with SSD, we see improvements up to 15x."
- COUCHDB-3063: "stale=ok" option replaced with new "stable" and "update" options.
- COUCHDB-3180: Add features list in the welcome message
- COUCHDB-3203: Make auth handlers configurable (in ini files)
- COUCHDB-3234: Track open shard timeouts with a counter instead of logging
- COUCHDB-3242: Make get view group info timeout in couch_indexer configurable
- COUCHDB-3249: Add config to disable index all fields (text indexes)
- COUCHDB-3251: Remove hot loop usage of filename:rootname/1
- COUCHDB-3284: 8Kb read-ahead in couch_file causes extra IO and binary memory usage
- COUCHDB-3298: Optimize writing btree nodes
- COUCHDB-3302: (Improve) Attachment replication over low bandwidth network connections
- COUCHDB-3307: Limit calls to maybe_add_sys_db_callbacks to once per db open
- COUCHDB-3318: bypass couch_httpd_vhost if there are none
- COUCHDB-3323: Idle dbs cause excessive overhead
- COUCHDB-3324: Introduce couch_replicator_scheduler
- COUCHDB-3337: End-point _local_docs doesn't conform to query params of _all_docs
- COUCHDB-3358: (mango) Use efficient set storage for field names
- COUCHDB-3425: Make _doc_ids _changes filter fast-path limit configurable
- #457: TeX/LaTeX/texinfo removed from default docs build chain
- #469: (mango) Choose index based on fields match
- #483: couchup database migration tool
- #582: Add X-Frame-Options support to help protect against clickjacking
- #593: Allow bind address of 127.0.0.1 in `_cluster_setup` for single nodes
- #624: Enable compaction daemon by default
- #626: Allow enable node decom using string "true"
- (mango) Configurable default limit, defaults to 25.
- (mango) _design documents ignored when querying _all_docs
- (mango) add $allMatch selector
- Add local.d/default.d directories by default and document

- Improved INSTALL.* text files

16.7.5 Fixed Issues

The 2.1.0 release includes fixes for the following issues:

- COUCHDB-1447: X-Couch-Update-NewRev header is missed if custom headers are specified in response of _update handler (missed in 2.0 merge)
- COUCHDB-2731: Authentication DB was not considered a system DB
- COUCHDB-3010: (Superseded fix for replication exponential backoff)
- COUCHDB-3090: Error when handling empty "Access-Control-Request-Headers" header
- COUCHDB-3100: Fix documentation on require_valid_user
- COUCHDB-3109: 500 when include_docs=true for linked documents
- COUCHDB-3113: fabric:open_revs can return {ok, []}
- COUCHDB-3149: Exception written to the log if db deleted while there is a change feed running
- COUCHDB-3150: Update all shards with stale=update_after
- COUCHDB-3158: Fix a crash when connection closes for _update
- COUCHDB-3162: Default ssl settings cause a crash
- COUCHDB-3164: Request fails when using _changes?feed=eventsource&heartbeat=30000
- COUCHDB-3168: Replicator doesn't handle well writing documents to a target db which has a small max_document_size
- COUCHDB-3173: Views return corrupt data for text fields containing non-BMP characters
- COUCHDB-3174: max_document_size setting can by bypassed by issuing multipart/related requests
- COUCHDB-3178: Fabric does not send message when filtering lots of documents
- COUCHDB-3181: function_clause error when adding attachment to doc in _users db
- COUCHDB-3184: couch_mrview_compactor:recompact/1 does not handle errors in spawned process
- COUCHDB-3193: fabric:open_revs returns multiple results when one of the shards has stem_interactive_updates=false
- COUCHDB-3199: Replicator VDU function doesn't account for an already malformed document in replicator db
- COUCHDB-3202: (mango) do not allow empty field names
- COUCHDB-3220: Handle timeout in _revs_diff
- COUCHDB-3222: (Fix) HTTP code 500 instead of 400 for invalid key during document creation
- COUCHDB-3231: Allow fixing users' documents (type and roles)
- COUCHDB-3232: user context not passed down in fabric_view_all_docs
- COUCHDB-3238: os_process_limit documentation wrong
- COUCHDB-3241: race condition in couch_server if delete msg for a db is received before open_result msg
- COUCHDB-3245: Make couchjs -S option take effect again
- COUCHDB-3252: Include main-coffee.js in release artifact (broken CoffeeScript view server)
- COUCHDB-3255: Conflicts introduced by recreating docs with attachments
- COUCHDB-3259: Don't trap exits in couch_file
- COUCHDB-3264: POST to _all_docs does not respect conflicts=true

- COUCHDB-3269: view response can 'hang' with filter and limit specified
- COUCHDB-3271: Replications crash with 'kaboom' exit
- COUCHDB-3274: eof in couch_file can be incorrect after error
- COUCHDB-3277: Replication manager crashes when it finds _replicator db shards which are not part of a mem3 db
- COUCHDB-3286: Validation function throwing unexpected json crashes with function_clause
- COUCHDB-3289: handle error clause when calling fabric:open_revs
- COUCHDB-3291: Excessively long document IDs prevent replicator from making progress
- COUCHDB-3293: Allow limiting length of document ID (for CouchDB proper)
- COUCHDB-3305: (mango) don't crash with invalid input to built in reducer function
- COUCHDB-3362: DELETE attachment on non-existing document creates the document, rather than returning 404
- COUCHDB-3364: Don't crash compactor when compacting process fails.
- COUCHDB-3367: Require server admin user for db/_compact and db_view_cleanup endpoints
- COUCHDB-3376: Fix mem3_shards under load
- COUCHDB-3378: Fix mango full text detection
- COUCHDB-3379: Fix couch_auth_cache reinitialization logic
- COUCHDB-3400: Notify couch_index_processes on all shards when ddoc updated
- COUCHDB-3402: race condition in mem3 startup
- #511: (mango) Return false for empty list
- #595: Return 409 to PUT attachment with non-existent rev
- #623: Ensure replicator _active_tasks entry reports recent pending changes value
- #627: Pass UserCtx to fabric's all_docs from mango query
- #631: fix couchdb_os_proc_pool eunit timeouts
- #644: Make couch_event_sup:stop/1 synchronous
- #645: Pass db open options to fabric_view_map for _view and _list queries on _users DB
- #648: Fix couch_replicator_changes_reader:process_change
- #649: Avoid a race when restarting an index updater
- #667: Prevent a terrible race condition
- #677: Make replication filter fetch error for _replicate return a 404
- Fix CORS `max_age` configuration parameter via Access-Control-Max-Age
- Chunk missing revisions before attempting to save on target (improves replication for very conflicted, very deep revision tree documents)
- Allow w parameter for attachments
- Return "Bad Request" when count in `/_uuids` exceeds max
- Fix crashes when replicator db is deleted
- Skip internal replication if changes already replicated
- Fix encoding issues on `_update/../doc_id` and PUT attachments

16.8 2.0.x Branch

- *Version 2.0.0*
- *Upgrade Notes*
- *Known Issues*
- *Breaking Changes*

16.8.1 Version 2.0.0

- Native clustering is now supported. Rather than use CouchDB replication between multiple, distinct CouchDB servers, configure a cluster of CouchDB nodes. These nodes will use an optimized Erlang-driven 'internal replication' to ensure data durability and accessibility. Combine a clustered CouchDB with a load balancer (such as `haproxy`) to scale CouchDB out horizontally. More details of the clustering feature are available in the *Cluster Management*.
- *Futon* replaced by brand-new, completely re-engineered *Fauxton* interface. URL remains the same.
- The new Mango Query Server provides a simple JSON-based way to perform CouchDB queries without JavaScript or MapReduce. Mango Queries have a similar indexing speed advantage over JavaScript Queries than the Erlang Queries have (2x-10x faster indexing depending on doc size and system configuration). We recommend all new apps start using Mango as a default. Further details are available in the *_find, _index and _explain API*.
- Mango *selectors* can be used in _changes feeds instead of JavaScript MapReduce filters. Mango has been tested to be up to an order of magnitude (10x) faster than JavaScript in this application.
- *Rewrite rules* for URLs can be performed using JavaScript functions.
- *Multiple queries* can be made of a view with a single HTTP request.
- Views can be queried with sorting turned off (`sorted=false`) for a performance boost.
- The global changes feed has been enhanced. It is now resumable and persistent.
- New endpoints added (documentation forthcoming):
  - */_membership* shows all nodes in a cluster
  - `/_bulk_get` speeds up the replication protocol over low-latency connections
  - `/_node/` api to access individual nodes' configuration and compaction features
  - `/_cluster_setup` api to set up a cluster from scratch.
  - `/_up` api to signal health of a node to a load-balancer
  - `/db/_local_docs` and `/db/_design_docs` (similar to `/db/_all_docs`)
- The `/_log` endpoint was removed.
- "Backend" interface on port 5986 used for specific cluster admin tasks. Of interest are the `_nodes` and `_dbs` databases visible only through this interface.
- Support added for Erlang/OTP 17.x, 18.x and 19
- New streamlined build system written for Unix-like systems and Microsoft Windows
- *Configuration* has moved from `/_config` to `/_node/{node-name}/_config`
- `instance_start_time` now always reports `"0"`.

16.8.2 Upgrade Notes

- The update sequences returned by the */{db}/_changes* feed are no longer integers. They can be any JSON value. Applications should treat them as opaque values and return them to CouchDB as-is.
- Temporary views are no longer supported.
- It is possible to have multiple replicator databases. `replicator/db` config option has been removed. Instead `_replicator` and any database names ending with the `/_replicator` suffix will be recognized as replicator databases by the system.
- Note that the semantics of some API calls have changed due to the introduction of the clustering feature. Specifically, make note of the difference between receiving a `201` and a `202` when storing a document.
- `all_or_nothing` is no longer supported by the *bulk_docs* API
- After updating a design document containing a `show`, an immediate GET to that same `show` function may still return results from the previous definition. This is due to design document caching, which may take a few seconds to fully evict, or longer (up to ~30s) for a clustered installation.

16.8.3 Known Issues

All known issues filed against the 2.0 release are contained within the official *CouchDB JIRA instance* or *CouchDB GitHub Issues*.

The following are some highlights of known issues for which fixes did not land in time for the 2.0.0 release:

- COUCHDB-2980: The replicator (whether invoked via `_replicate` or a document stored in the `_replicator` database) understands two kinds of source and target:

  1. A URL (e.g., `https://foo:bar@foo.com/db1`), called a "remote" source or target
  2. A database name (e.g., `db1`), called a "local" source or target.

  Whenever the latter type is used, this refers to a local unclustered database, not a clustered one.

  In a future release we hope to support "local" source or target specs to clustered databases. For now, we recommend always using the URL format for both source and target specifications.
- COUCHDB-3034: CouchDB will occasionally return 500 errors when multiple clients attempt to PUT or DELETE the same database concurrently.
- COUCHDB-3119: Adding nodes to a cluster fails if the Erlang node name is not `couchdb` (of the form `couchdb@hostname`.)
- COUCHDB-3050: Occasionally the `dev/run` script used for development purposes to start a local 3-node cluster will fail to start one or more nodes.
- COUCHDB-2817: The compaction daemon will only compact views for shards that contain the design document.
- COUCHDB-2804: The fast_view optimization is not enabled on the clustered interface.
- #656: The OAuth 1.0 support is broken and deprecated. It will be removed in a future version of CouchDB.

16.8.4 Breaking Changes

The following changes in 2.0 represent a significant deviation from CouchDB 1.x and may alter behaviour of systems designed to work with older versions of CouchDB:

- #620: `POST /dbname` no longer returns an ETag response header, in compliance with RFC 7231, Section 7.2.

16.9 1.7.x Branch

- *Version 1.7.2*
- *Version 1.7.1*
- *Version 1.7.0*

16.9.1 Version 1.7.2

Security

- *CVE 2018-8007*

16.9.2 Version 1.7.1

Bug Fix

- #974: Fix access to /db/_all_docs for database members.

16.9.3 Version 1.7.0

Security

- *CVE 2017-12635*
- *CVE 2017-12636*

API Changes

- COUCHDB-1356: Return username on *`POST /_session`*.
- COUCHDB-1876: Fix duplicated Content-Type for show/update functions.
- COUCHDB-2310: Implement *`POST /{db}/_bulk_get`*.
- COUCHDB-2375: 400 Bad Request returned when invalid revision specified.
- COUCHDB-2845: 400 Bad Request returned when *revs* is not a list.

Build

- COUCHDB-1964: Replace etap test suite with EUnit.
- COUCHDB-2225: Enforce that shared libraries can be built by the system.
- COUCHDB-2761: Support glibc >= 2.20.
- COUCHDB-2747: Support Erlang 18.
- #5b9742c: Support Erlang 19.
- #1545bf4: Remove broken benchmarks.

Database Core

- COUCHDB-2534: Improve checks for db admin/member.
- COUCHDB-2735: Duplicate document _ids created under high edit load.

Documentation

- #c3c9588: Improve documentation of *cacert_file* ssl option.
- #3266f23: Clarify the purpose of tombstones.
- #75887d9: Improve CouchDB Replication Protocol definition.
- #3b1dc0f: Remove mention of *group_level=exact*.
- #2a11daa: Remove mention of "Test Suite" in Futon.
- #01c60f1: Clarify type of key, startkey and endkey params.

Futon

- COUCHDB-241: Support document copying.
- COUCHDB-1011: Run replication filtered by document ids from Futon.
- COUCHDB-1275: Unescape database names in Futon recently used list.
- #f18f82a: Update jquery.ui to 1.10.4 with fixes of potential XSS issues.

HTTP Server

- COUCHDB-2430: Disable Nagle's algorithm by default.
- COUCHDB-2583: Don't drop connection by the endpoints which doesn't require any payload.
- COUCHDB-2673: Properly escape Location: HTTP header.
- COUCHDB-2677: Wrong Expires header weekday.
- COUCHDB-2783: Bind both to IPv4 and IPv6.
- #f30f3dd: Support for user configurable SSL ciphers.

Query Server

- COUCHDB-1447: Custom response headers from design functions get merged with default ones.
- #7779c11: Upgrade Coffeescript to version 1.10.

jquery.couch.js

- #f9095e7: Fix document copying.

16.10 1.6.x Branch

- *Upgrade Notes*
- *Version 1.6.0*

16.10.1 Upgrade Notes

The *Proxy Authentication* handler was renamed to `proxy_authentication_handler` to follow the `*_authentication_handler` form of all other handlers. The old `proxy_authentification_handler` name is marked as deprecated and will be removed in future releases. It's strongly recommended to update `httpd/authentication_handlers` option with new value in case if you had used such handler.

16.10.2 Version 1.6.0

- COUCHDB-2200: support Erlang/OTP 17.0 #35e16032
- Fauxton: many improvements in our experimental new user interface, including switching the code editor from CodeMirror to Ace as well as better support for various browsers.
- Add the `max_count` option (*UUIDs Configuration*) to allow rate-limiting the amount of UUIDs that can be requested from the */_uuids* handler in a single request (*CVE 2014-2668*).
- COUCHDB-1986: increase socket buffer size to improve replication speed for large documents and attachments, and fix tests on BSD-like systems. #9a0e561b
- COUCHDB-1953: improve performance of multipart/related requests. #ce3e89dc
- COUCHDB-2221: verify that authentication-related configuration settings are well-formed. #dbe769c6
- COUCHDB-1922: fix CORS exposed headers. #4f619833
- Rename `proxy_authentification_handler` to `proxy_authentication_handler`. #c66ac4a8
- COUCHDB-1795: ensure the startup script clears the pid file on termination. #818ef4f9
- COUCHDB-1962: replication can now be performed without having write access to the source database (#1d5fe2aa), the replication checkpoint interval is now configurable (#0693f98e).
- COUCHDB-2025: add support for SOCKS5 proxies for replication. #fcd76c9
- COUCHDB-1930: redirect to the correct page after submitting a new document with a different ID than the one suggested by Futon. #4906b591
- COUCHDB-1923: add support for *attachments* and *att_encoding_info* options (formerly only available on the documents API) to the view API. #ca41964b
- COUCHDB-1647: for failed replications originating from a document in the *_replicator* database, store the failure reason in the document. #08cac68b

- A number of improvements for the documentation.

16.11 1.5.x Branch

- *Version 1.5.1*
- *Version 1.5.0*

Warning: *Version 1.5.1* contains important security fixes. Previous *1.5.x* releases are not recommended for regular usage.

16.11.1 Version 1.5.1

- Add the `max_count` option (*UUIDs Configuration*) to allow rate-limiting the amount of UUIDs that can be requested from the */_uuids* handler in a single request (*CVE 2014-2668*).

16.11.2 Version 1.5.0

- COUCHDB-1781: The official documentation has been overhauled. A lot of content from other sources have been merged, and the index page has been rebuilt to make the docs much more accessible. #54813a7
- A new administration UI, codenamed Fauxton, has been included as an experimental preview. It can be accessed at `/_utils/fauxton/`. There are too many improvements here to list them all. We are looking for feedback from the community on this preview release.
- COUCHDB-1888: Fixed an issue where admin users would be restricted by the `public_fields` feature.
- Fixed an issue with the JavaScript CLI test runner. #be76882, #54813a7
- COUCHDB-1867: An experimental plugin feature has been added. See `src/couch_plugin/README.md` for details. We invite the community to test and report any findings.
- COUCHDB-1894: An experimental Node.js-based query server runtime has been added. See *Experimental Features* for details. We invite the community to test and report any findings.
- COUCHDB-1901: Better retry mechanism for transferring attachments during replication. #4ca2cec

16.12 1.4.x Branch

- *Upgrade Notes*
- *Version 1.4.0*

Warning: *1.4.x Branch* is affected by the issue described in *CVE-2014-2668: DoS (CPU and memory consumption) via the count parameter to /_uuids*. Upgrading to a more recent release is strongly recommended.

16.12.1 Upgrade Notes

We now support Erlang/OTP R16B and R16B01; the minimum required version is R14B.

User document role values must now be strings. Other types of values will be refused when saving the user document.

16.12.2 Version 1.4.0

- COUCHDB-1139: it's possible to apply *list* functions to `_all_docs` view. #54fd258e
- COUCHDB-1632: Ignore epilogues in `multipart/related` MIME attachments. #2b4ab67a
- COUCHDB-1634: Reduce PBKDF2 work factor. #f726bc4d
- COUCHDB-1684: Support for server-wide changes feed reporting on creation, updates and deletion of databases. #917d8988
- COUCHDB-1772: Prevent invalid JSON output when using *all_or_nothing of bulk API*. #dfd39d57
- Add a `configurable whitelist` of user document properties. #8d7ab8b1
- COUCHDB-1852: Support Last-Event-ID header in EventSource changes feeds. #dfd2199a
- Allow storing pre-hashed admin passwords via *config API*. #c98ba561
- Automatic loading of CouchDB plugins. #3fab6bb5
- Much improved documentation, including an *expanded description* of *validate_doc_update* functions (commit:*ef9ac469*) and a description of how CouchDB handles JSON *number values* (#bbd93f77).
- Split up *replicator_db* tests into multiple independent tests.

16.13 1.3.x Branch

- *Upgrade Notes*
- *Version 1.3.1*
- *Version 1.3.0*

Warning: *1.3.x Branch* is affected by the issue described in *CVE-2014-2668: DoS (CPU and memory consumption) via the count parameter to /_uuids*. Upgrading to a more recent release is strongly recommended.

16.13.1 Upgrade Notes

You can upgrade your existing CouchDB 1.0.x installation to 1.3.0 without any specific steps or migration. When you run CouchDB, the existing data and index files will be opened and used as normal.

The first time you run a compaction routine on your database within 1.3.0, the data structure and indexes will be updated to the new version of the CouchDB database format that can only be read by CouchDB 1.3.0 and later. This step is not reversible. Once the data files have been updated and migrated to the new version the data files will no longer work with a CouchDB 1.0.x release.

Warning: If you want to retain support for opening the data files in CouchDB 1.0.x you must back up your data files before performing the upgrade and compaction process.

16.13.2 Version 1.3.1

Replicator

- COUCHDB-1788: Tolerate missing source and target fields in _replicator docs. #869f42e2

Log System

- COUCHDB-1794: Fix bug in WARN level logging from 1.3.0.
- Don't log about missing .compact files. #06f1a8dc

View Server

- COUCHDB-1792: Fix the -S option to couchjs to increase memory limits. #cfaa66cd

Miscellaneous

- COUCHDB-1784: Improvements to test suite and VPATH build system. #01afaa4f
- Improve documentation: better structure, improve language, less duplication.

16.13.3 Version 1.3.0

Database core

- COUCHDB-1512: Validate bind address before assignment. #09ead8a0
- Restore `max_document_size` protection. #bf1eb135

Documentation

- COUCHDB-1523: Import CouchBase documentation and convert them into Sphinx docs

Futon

- COUCHDB-509: Added view request duration to Futon. #2d2c7d1e
- COUCHDB-627: Support all timezones. #b1a049bb
- COUCHDB-1383: Futon view editor won't allow you to save original view after saving a revision. #ce48342
- COUCHDB-1470: Futon raises pop-up on attempt to navigate to missed/deleted document. #5da40eef
- COUCHDB-1473, COUCHDB-1472: Disable buttons for actions that the user doesn't have permissions to. #7156254d

HTTP Interface

- COUCHDB-431: Introduce experimental *CORS support*. #b90e4021
- COUCHDB-764, COUCHDB-514, COUCHDB-430: Fix sending HTTP headers from `_list` function, #2a74f88375
- COUCHDB-887: Fix `bytes` and `offset` parameters semantic for *_log* resource (explanation) #ad700014
- COUCHDB-986: Added Server-Sent Events protocol to db changes API. See http://www.w3.org/TR/eventsource/ for details. #093d2aa6
- COUCHDB-1026: Database names are encoded with respect of special characters in the rewriter now. #272d6415
- COUCHDB-1097: Allow *OPTIONS* request to shows and lists functions. #9f53704a
- COUCHDB-1210: Files starting with underscore can be attached and updated now. #05858792
- COUCHDB-1277: Better query parameter support and code clarity: #7e3c69ba
  - Responses to documents created/modified via form data *POST* to /db/doc or copied with *COPY* should now include *Location* header.
  - Form data POST to /db/doc now includes an *ETag* response header.
  - `?batch=ok` is now supported for *COPY* and *POST* /db/doc updates.
  - `?new_edits=false` is now supported for more operations.
- COUCHDB-1285: Allow configuration of vendor and modules version in CouchDB welcome message. #3c24a94d
- COUCHDB-1321: Variables in rewrite rules breaks OAuth authentication. #c307ba95
- COUCHDB-1337: Use MD5 for attachment ETag header value. #6d912c9f
- COUCHDB-1381: Add jquery.couch support for Windows 8 Metro apps. #dfc5d37c
- COUCHDB-1441: Limit recursion depth in the URL rewriter. Defaults to a maximum of 100 invocations but is configurable. #d076976c
- COUCHDB-1442: No longer rewrites the *X-CouchDB-Requested-Path* during recursive calls to the rewriter. #56744f2f
- COUCHDB-1501: *Changes feed* now can take special parameter `since=now` to emit changes since current point of time. #3bbb2612
- COUCHDB-1502: Allow users to delete own _users doc. #f0d6f19bc8
- COUCHDB-1511: CouchDB checks *roles* field for *_users* database documents with more care. #41205000
- COUCHDB-1537: Include user name in show/list *ETags*. #ac320479
- Send a 202 response for *_restart*. #b213e16f
- Make password hashing synchronous when using the /_config/admins API. #08071a80
- Add support to serve single file with CouchDB, #2774531ff2
- Allow any 2xx code to indicate success, #0d50103cfd
- Fix *_session* for IE7.
- Restore 400 error for empty PUT, #2057b895
- Return `X-Couch-Id` header if doc is created, #98515bf0b9
- Support auth cookies with `:` characters, #d9566c831d

Log System

- COUCHDB-1380: Minor fixes for logrotate support.
- Improve file I/O error logging and handling, #4b6475da
- Module Level Logging, #b58f069167
- Log 5xx responses at error level, #e896b0b7
- Log problems opening database at ERROR level except for auto-created system dbs, #41667642f7

Replicator

- COUCHDB-1248: *HTTP 500* error now doesn't occurs when replicating with `?doc_ids=null`. #bea76dbf
- COUCHDB-1259: Stabilize replication id, #c6252d6d7f
- COUCHDB-1323: Replicator now acts as standalone application. #f913ca6e
- COUCHDB-1363: Fix rarely occurred, but still race condition in changes feed if a quick burst of changes happens while replication is starting the replication can go stale. #573a7bb9
- COUCHDB-1557: Upgrade some code to use BIFs bring good improvements for replication.

Security

- COUCHDB-1060: Passwords are now hashed using the PBKDF2 algorithm with a configurable work factor. #7d418134

Source Repository

- The source repository was migrated from SVN to Git.

Storage System

- Fixed unnecessary conflict when deleting and creating a document in the same batch.

Test Suite

- COUCHDB-1321: Moved the JS test suite to the CLI.
- COUCHDB-1338: Start CouchDB with `port=0`. While CouchDB might be already running on the default port 5984, port number 0 let the TCP stack figure out a free port to run. #127cbe3
- COUCHDB-1339: Use shell trap to catch dying beam processes during test runs. #2921c78
- COUCHDB-1389: Improved tracebacks printed by the JS CLI tests.
- COUCHDB-1563: Ensures urlPrefix is set in all ajax requests. #07a6af222
- Fix race condition for test running on faster hardware.
- Improved the reliability of a number of tests.

URL Rewriter & Vhosts

- COUCHDB-1026: Database name is encoded during rewriting (allowing embedded /'s, etc). #272d6415

UUID Algorithms

- COUCHDB-1373: Added the utc_id algorithm #5ab712a2

Query and View Server

- COUCHDB-111: Improve the errors reported by the JavaScript view server to provide a more friendly error report when something goes wrong. #0c619ed
- COUCHDB-410: More graceful error handling for JavaScript validate_doc_update functions.
- COUCHDB-1372: *_stats* built-in reduce function no longer produces error for empty view result.
- COUCHDB-1444: Fix missed_named_view error that occurs on existed design documents and views. #b59ac98b
- COUCHDB-1445: CouchDB tries no more to delete view file if it couldn't open it, even if the error is *emfile*.
- COUCHDB-1483: Update handlers requires valid doc ids. #72ea7e38
- COUCHDB-1491: Clean up view tables. #c37204b7
- Deprecate E4X support, #cdfdda2314

Windows

- COUCHDB-1482: Use correct linker flag to build *snappy_nif.dll* on Windows. #a6eaf9f1
- Allows building cleanly on Windows without cURL, #fb670f5712

16.14 1.2.x Branch

- *Upgrade Notes*
- *Version 1.2.2*
- *Version 1.2.1*
- *Version 1.2.0*

16.14.1 Upgrade Notes

Warning: This version drops support for the database format that was introduced in version 0.9.0. Compact your older databases (that have not been compacted for a long time) before upgrading, or they will become inaccessible.

Warning: *Version 1.2.1* contains important security fixes. Previous *1.2.x* releases are not recommended for regular usage.

Security changes

The interface to the `_users` and `_replicator` databases have been changed so that non-administrator users can see less information:

- In the `_users` database:
  - User documents can now only be read by the respective users, as well as administrators. Other users cannot read these documents.
  - Views can only be defined and queried by administrator users.
  - The `_changes` feed can only be queried by administrator users.
- In the `_replicator` database:
  - Documents now have a forced `owner` field that corresponds to the authenticated user that created them.
  - Non-owner users will not see confidential information like passwords or OAuth tokens in replication documents; they can still see the other contents of those documents. Administrators can see everything.
  - Views can only be defined and queried by administrators.

Database Compression

The new optional (but enabled by default) compression of disk files requires an upgrade of the on-disk format (5 -> 6) which occurs on creation for new databases and views, and on compaction for existing files. This format is not supported in previous releases, so rollback would require replication to the previous CouchDB release or restoring from backup.

Compression can be disabled by setting `compression = none` in your `local.ini [couchdb]` section, but the on-disk format will still be upgraded.

16.14.2 Version 1.2.2

Build System

- Fixed issue in *couchdb* script where stopped status returns before process exits.

HTTP Interface

- Reset rewrite counter on new request, avoiding unnecessary request failures due to bogus rewrite limit reports.

16.14.3 Version 1.2.1

Build System

- Fix couchdb start script.
- Win: fix linker invocations.

Futon

- Disable buttons that aren't available for the logged-in user.

HTTP Interface

- No longer rewrites the `X-CouchDB-Requested-Path` during recursive calls to the rewriter.
- Limit recursion depth in the URL rewriter. Defaults to a maximum of 100 invocations but is configurable.

Security

- Fixed *CVE-2012-5641: Information disclosure via unescaped backslashes in URLs on Windows*
- Fixed *CVE-2012-5649: JSONP arbitrary code execution with Adobe Flash*
- Fixed *CVE-2012-5650: DOM based Cross-Site Scripting via Futon UI*

Replication

- Fix potential timeouts.

View Server

- Change use of signals to avoid broken view groups.

16.14.4 Version 1.2.0

Authentication

- Fix use of OAuth with VHosts and URL rewriting.
- OAuth secrets can now be stored in the users system database as an alternative to key value pairs in the .ini configuration. By default this is disabled (secrets are stored in the .ini) but can be enabled via the .ini configuration key *use_users_db* in the *couch_httpd_oauth* section.
- Documents in the _users database are no longer publicly readable.
- Confidential information in the _replication database is no longer publicly readable.
- Password hashes are now calculated by CouchDB. Clients are no longer required to do this manually.
- Cookies used for authentication can be made persistent by enabling the .ini configuration key *allow_persistent_cookies* in the *couch_httpd_auth* section.

Build System

- cURL is no longer required to build CouchDB as it is only used by the command line JS test runner. If cURL is available when building CouchJS you can enable the HTTP bindings by passing -H on the command line.
- Temporarily made *make check* pass with R15B. A more thorough fix is in the works (COUCHDB-1424).
- Fixed –with-js-include and –with-js-lib options.
- Added –with-js-lib-name option.

Futon

- The *Status* screen (active tasks) now displays two new task status fields: *Started on* and *Updated on*.
- Futon remembers view code every time it is saved, allowing to save an edit that amounts to a revert.

HTTP Interface

- Added a native JSON parser.
- The _active_tasks API now offers more granular fields. Each task type is now able to expose different properties.
- Added built-in changes feed filter *_view*.
- Fixes to the *_changes* feed heartbeat option which caused heartbeats to be missed when used with a filter. This caused timeouts of continuous pull replications with a filter.
- Properly restart the SSL socket on configuration changes.

OAuth

- Updated bundled *erlang_oauth* library to the latest version.

Replicator

- A new replicator implementation. It offers more performance and configuration options.
- Passing non-string values to query_params is now a 400 bad request. This is to reduce the surprise that all parameters are converted to strings internally.
- Added optional field *since_seq* to replication objects/documents. It allows to bootstrap a replication from a specific source sequence number.
- Simpler replication cancellation. In addition to the current method, replications can now be canceled by specifying the replication ID instead of the original replication object/document.

Storage System

- Added optional database and view index file compression (using Google's snappy or zlib's deflate). This feature is enabled by default, but it can be disabled by adapting local.ini accordingly. The on-disk format is upgraded on compaction and new DB/view creation to support this.
- Several performance improvements, most notably regarding database writes and view indexing.
- Computation of the size of the latest MVCC snapshot data and all its supporting metadata, both for database and view index files. This information is exposed as the *data_size* attribute in the database and view group information URIs.
- The size of the buffers used for database and view compaction is now configurable.
- Added support for automatic database and view compaction. This feature is disabled by default, but it can be enabled via the .ini configuration.
- Performance improvements for the built-in changes feed filters *_doc_ids* and *_design*.

View Server

- Add CoffeeScript (http://coffeescript.org/) as a first class view server language.
- Fixed old index file descriptor leaks after a view cleanup.
- The requested_path property keeps the pre-rewrite path even when no VHost configuration is matched.
- Fixed incorrect reduce query results when using pagination parameters.
- Made icu_driver work with Erlang R15B and later.

16.15 1.1.x Branch

- *Upgrade Notes*
- *Version 1.1.2*
- *Version 1.1.1*
- *Version 1.1.0*

16.15.1 Upgrade Notes

Warning: *Version 1.1.2* contains important security fixes. Previous *1.1.x* releases are not recommended for regular usage.

16.15.2 Version 1.1.2

Build System

- Don't *ln* the *couchjs* install target on Windows
- Remove ICU version dependency on Windows.
- Improve SpiderMonkey version detection.

HTTP Interface

- ETag of attachment changes only when the attachment changes, not the document.
- Fix retrieval of headers larger than 4k.
- Allow OPTIONS HTTP method for list requests.
- Don't attempt to encode invalid json.

Log System

- Improvements to log messages for file-related errors.

Replicator

- Fix pull replication of documents with many revisions.
- Fix replication from an HTTP source to an HTTP target.

Security

- Fixed *CVE-2012-5641: Information disclosure via unescaped backslashes in URLs on Windows*
- Fixed *CVE-2012-5649: JSONP arbitrary code execution with Adobe Flash*
- Fixed *CVE-2012-5650: DOM based Cross-Site Scripting via Futon UI*

View Server

- Avoid invalidating view indexes when running out of file descriptors.

16.15.3 Version 1.1.1

- Support SpiderMonkey 1.8.5
- Add configurable maximum to the number of bytes returned by _log.
- Allow CommonJS modules to be an empty string.
- Bump minimum Erlang version to R13B02.
- Do not run deleted validate_doc_update functions.
- ETags for views include current sequence if include_docs=true.
- Fix bug where duplicates can appear in _changes feed.
- Fix bug where update handlers break after conflict resolution.
- Fix bug with _replicator where include "filter" could crash couch.
- Fix crashes when compacting large views.
- Fix file descriptor leak in _log
- Fix missing revisions in _changes?style=all_docs.
- Improve handling of compaction at max_dbs_open limit.
- JSONP responses now send "text/javascript" for Content-Type.
- Link to ICU 4.2 on Windows.
- Permit forward slashes in path to update functions.
- Reap couchjs processes that hit reduce_overflow error.
- Status code can be specified in update handlers.
- Support provides() in show functions.
- _view_cleanup when ddoc has no views now removes all index files.
- max_replication_retry_count now supports "infinity".
- Fix replication crash when source database has a document with empty ID.

- Fix deadlock when assigning couchjs processes to serve requests.
- Fixes to the document multipart PUT API.
- Fixes regarding file descriptor leaks for databases with views.

16.15.4 Version 1.1.0

Note: All CHANGES for 1.0.2 and 1.0.3 also apply to 1.1.0.

Externals

- Added OS Process module to manage daemons outside of CouchDB.
- Added HTTP Proxy handler for more scalable externals.

Futon

- Added a "change password"-feature to Futon.

HTTP Interface

- Native SSL support.
- Added support for HTTP range requests for attachments.
- Added built-in filters for *_changes*: *_doc_ids* and *_design*.
- Added configuration option for TCP_NODELAY aka "Nagle".
- Allow POSTing arguments to *_changes*.
- Allow *keys* parameter for GET requests to views.
- Allow wildcards in vhosts definitions.
- More granular ETag support for views.
- More flexible URL rewriter.
- Added support for recognizing "Q values" and media parameters in HTTP Accept headers.
- Validate doc ids that come from a PUT to a URL.

Replicator

- Added *_replicator* database to manage replications.
- Fixed issues when an endpoint is a remote database accessible via SSL.
- Added support for continuous by-doc-IDs replication.
- Fix issue where revision info was omitted when replicating attachments.
- Integrity of attachment replication is now verified by MD5.

Storage System

- Multiple micro-optimizations when reading data.

URL Rewriter & Vhosts

- Fix for variable substitution

View Server

- Added CommonJS support to map functions.
- Added *stale=update_after* query option that triggers a view update after returning a *stale=ok* response.
- Warn about empty result caused by *startkey* and *endkey* limiting.
- Built-in reduce function *_sum* now accepts lists of integers as input.
- Added view query aliases start_key, end_key, start_key_doc_id and end_key_doc_id.

16.16 1.0.x Branch

- *Upgrade Notes*
- *Version 1.0.4*
- *Version 1.0.3*
- *Version 1.0.2*
- *Version 1.0.1*
- *Version 1.0.0*

16.16.1 Upgrade Notes

Note, to replicate with a 1.0 CouchDB instance you must first upgrade in-place your current CouchDB to 1.0 or 0.11.1 – backporting so that 0.10.x can replicate to 1.0 wouldn't be that hard. All that is required is patching the replicator to use the `application/json` content type.

- `_log` and `_temp_views` are now admin-only resources.
- `_bulk_docs` now requires a valid *Content-Type* header of `application/json`.
- *JSONP* is disabled by default. An .ini option was added to selectively enable it.
- The `key`, `startkey` and `endkey` properties of the request object passed to *list* and *show* functions now contain JSON objects representing the URL encoded string values in the query string. Previously, these properties contained strings which needed to be converted to JSON before using.

Warning: *Version 1.0.4* contains important security fixes. Previous *1.0.x* releases are not recommended for regular usage.

16.16.2 Version 1.0.4

HTTP Interface

- Fix missing revisions in `_changes?style=all_docs`.
- Fix validation of attachment names.

Log System

- Fix file descriptor leak in `_log`.

Replicator

- Fix a race condition where replications can go stale

Security

- Fixed *CVE-2012-5641: Information disclosure via unescaped backslashes in URLs on Windows*
- Fixed *CVE-2012-5649: JSONP arbitrary code execution with Adobe Flash*
- Fixed *CVE-2012-5650: DOM based Cross-Site Scripting via Futon UI*

View System

- Avoid invalidating view indexes when running out of file descriptors.

16.16.3 Version 1.0.3

General

- Fixed compatibility issues with Erlang R14B02.

Etap Test Suite

- Etap tests no longer require use of port 5984. They now use a randomly selected port so they won't clash with a running CouchDB.

Futon

- Made compatible with jQuery 1.5.x.

HTTP Interface

- Fix bug that allows invalid UTF-8 after valid escapes.
- The query parameter *include_docs* now honors the parameter *conflicts*. This applies to queries against map views, _all_docs and _changes.
- Added support for inclusive_end with reduce views.

Replicator

- Enabled replication over IPv6.
- Fixed for crashes in continuous and filtered changes feeds.
- Fixed error when restarting replications in OTP R14B02.
- Upgrade ibrowse to version 2.2.0.
- Fixed bug when using a filter and a limit of 1.

Security

- Fixed OAuth signature computation in OTP R14B02.
- Handle passwords with : in them.

Storage System

- More performant queries against _changes and _all_docs when using the *include_docs* parameter.

Windows

- Windows builds now require ICU >= 4.4.0 and Erlang >= R14B03. See COUCHDB-1152, and COUCHDB-963 + OTP-9139 for more information.

16.16.4 Version 1.0.2

Futon

- Make test suite work with Safari and Chrome.
- Fixed animated progress spinner.
- Fix raw view document link due to overzealous URI encoding.
- Spell javascript correctly in loadScript(uri).

HTTP Interface

- Allow reduce=false parameter in map-only views.
- Fix parsing of Accept headers.
- Fix for multipart GET APIs when an attachment was created during a local-local replication. See COUCHDB-1022 for details.

Log System

- Reduce lengthy stack traces.
- Allow logging of native <xml> types.

Replicator

- Updated ibrowse library to 2.1.2 fixing numerous replication issues.
- Make sure that the replicator respects HTTP settings defined in the config.
- Fix error when the ibrowse connection closes unexpectedly.
- Fix authenticated replication (with HTTP basic auth) of design documents with attachments.
- Various fixes to make replication more resilient for edge-cases.

Storage System

- Fix leaking file handles after compacting databases and views.
- Fix databases forgetting their validation function after compaction.
- Fix occasional timeout errors after successfully compacting large databases.
- Fix occasional error when writing to a database that has just been compacted.
- Fix occasional timeout errors on systems with slow or heavily loaded IO.
- Fix for OOME when compactions include documents with many conflicts.
- Fix for missing attachment compression when MIME types included parameters.
- Preserve purge metadata during compaction to avoid spurious view rebuilds.
- Fix spurious conflicts introduced when uploading an attachment after a doc has been in a conflict. See COUCHDB-902 for details.
- Fix for frequently edited documents in multi-master deployments being duplicated in _changes and _all_docs. See COUCHDB-968 for details on how to repair.
- Significantly higher read and write throughput against database and view index files.

View Server

- Don't trigger view updates when requesting *_design/doc/_info*.
- Fix for circular references in CommonJS requires.
- Made isArray() function available to functions executed in the query server.
- Documents are now sealed before being passed to map functions.
- Force view compaction failure when duplicated document data exists. When this error is seen in the logs users should rebuild their views from scratch to fix the issue. See COUCHDB-999 for details.

16.16.5 Version 1.0.1

Authentication

- **Enable basic-auth popup when required to access the server, to prevent**
  people from getting locked out.

Build and System Integration

- Included additional source files for distribution.

Futon

- User interface element for querying stale (cached) views.

HTTP Interface

- Expose *committed_update_seq* for monitoring purposes.
- Show fields saved along with _deleted=true. Allows for auditing of deletes.
- More robust Accept-header detection.

Replicator

- Added support for replication via an HTTP/HTTPS proxy.
- Fix pull replication of attachments from 0.11 to 1.0.x.
- Make the _changes feed work with non-integer seqnums.

Storage System

- Fix data corruption bug COUCHDB-844. Please see http://couchdb.apache.org/notice/1.0.1.html for details.

16.16.6 Version 1.0.0

Security

- Added authentication caching, to avoid repeated opening and closing of the users database for each request requiring authentication.

Storage System

- Small optimization for reordering result lists.
- More efficient header commits.
- Use O_APPEND to save lseeks.
- Faster implementation of pread_iolist(). Further improves performance on concurrent reads.

View Server

- Faster default view collation.
- Added option to include update_seq in view responses.

16.17 0.11.x Branch

- *Upgrade Notes*
- *Version 0.11.2*
- *Version 0.11.1*
- *Version 0.11.0*

16.17.1 Upgrade Notes

Warning: *Version 0.11.2* contains important security fixes. Previous *0.11.x* releases are not recommended for regular usage.

Changes Between 0.11.0 and 0.11.1

- `_log` and `_temp_views` are now admin-only resources.
- `_bulk_docs` now requires a valid *Content-Type* header of `application/json`.
- *JSONP* is disabled by default. An .ini option was added to selectively enable it.
- The `key`, `startkey` and `endkey` properties of the request object passed to *list* and *show* functions now contain JSON objects representing the URL encoded string values in the query string. Previously, these properties contained strings which needed to be converted to JSON before using.

Changes Between 0.10.x and 0.11.0

show, list, update and validation functions

The `req` argument to show, list, update and validation functions now contains the member method with the specified HTTP method of the current request. Previously, this member was called `verb`. `method` is following **RFC 2616** (HTTP 1.1) closer.

_admins -> _security

The */db/_admins* handler has been removed and replaced with a */{db}/_security* object. Any existing *_admins* will be dropped and need to be added to the security object again. The reason for this is that the old system made no distinction between names and roles, while the new one does, so there is no way to automatically upgrade the old admins list.

The security object has 2 special fields, `admins` and `readers`, which contain lists of names and roles which are admins or readers on that database. Anything else may be stored in other fields on the security object. The entire object is made available to validation functions.

json2.js

JSON handling in the query server has been upgraded to use json2.js. This allows us to use faster native JSON serialization when it is available.

In previous versions, attempts to serialize undefined would throw an exception, causing the doc that emitted undefined to be dropped from the view index. The new behavior is to serialize undefined as null. Applications depending on the old behavior will need to explicitly check for undefined.

Another change is that E4X's XML objects will not automatically be stringified. XML users will need to call `my_xml_object.toXMLString()` to return a string value. #8d3b7ab3

WWW-Authenticate

The default configuration has been changed to avoid causing basic-auth popups which result from sending the WWW-Authenticate header. To enable basic-auth popups, uncomment the config option `httpd/WWW-Authenticate` line in *local.ini*.

Query server line protocol

The query server line protocol has changed for all functions except *map*, *reduce*, and *rereduce*. This allows us to cache the entire design document in the query server process, which results in faster performance for common operations. It also gives more flexibility to query server implementers and shouldn't require major changes in the future when adding new query server features.

UTF8 JSON

JSON request bodies are validated for proper UTF-8 before saving, instead of waiting to fail on subsequent read requests.

_changes line format

Continuous changes are now newline delimited, instead of having each line followed by a comma.

16.17.2 Version 0.11.2

Authentication

- User documents can now be deleted by admins or the user.

Futon

- Add some Futon files that were missing from the Makefile.

HTTP Interface

- Better error messages on invalid URL requests.

Replicator

- Fix bug when pushing design docs by non-admins, which was hanging the replicator for no good reason.
- Fix bug when pulling design documents from a source that requires basic-auth.

Security

- Avoid potential DOS attack by guarding all creation of atoms.
- Fixed *CVE-2010-2234: Apache CouchDB Cross Site Request Forgery Attack*

16.17.3 Version 0.11.1

Build and System Integration

- Output of *couchdb –help* has been improved.
- Fixed compatibility with the Erlang R14 series.
- Fixed warnings on Linux builds.
- Fixed build error when aclocal needs to be called during the build.
- Require ICU 4.3.1.
- Fixed compatibility with Solaris.

Configuration System

- Fixed timeout with large .ini files.

Futon

- Use "expando links" for over-long document values in Futon.
- Added continuous replication option.
- Added option to replicating test results anonymously to a community CouchDB instance.
- Allow creation and deletion of config entries.
- Fixed display issues with doc ids that have escaped characters.
- Fixed various UI issues.

HTTP Interface

- Mask passwords in active tasks and logging.
- Update mochijson2 to allow output of BigNums not in float form.
- Added support for X-HTTP-METHOD-OVERRIDE.
- Better error message for database names.
- Disable jsonp by default.
- Accept gzip encoded standalone attachments.
- Made max_concurrent_connections configurable.
- Made changes API more robust.
- Send newly generated document rev to callers of an update function.

JavaScript Clients

- Added tests for couch.js and jquery.couch.js
- Added changes handler to jquery.couch.js.
- Added cache busting to jquery.couch.js if the user agent is msie.
- Added support for multi-document-fetch (via _all_docs) to jquery.couch.js.
- Added attachment versioning to jquery.couch.js.
- Added option to control ensure_full_commit to jquery.couch.js.
- Added list functionality to jquery.couch.js.
- Fixed issues where bulkSave() wasn't sending a POST body.

Log System

- Log HEAD requests as HEAD, not GET.
- Keep massive JSON blobs out of the error log.
- Fixed a timeout issue.

Replication System

- Refactored various internal APIs related to attachment streaming.
- Fixed hanging replication.
- Fixed keepalive issue.

Security

- Added authentication redirect URL to log in clients.
- Fixed query parameter encoding issue in oauth.js.
- Made authentication timeout configurable.
- Temporary views are now admin-only resources.

Storage System

- Don't require a revpos for attachment stubs.
- Added checking to ensure when a revpos is sent with an attachment stub, it's correct.
- Make file deletions async to avoid pauses during compaction and db deletion.
- Fixed for wrong offset when writing headers and converting them to blocks, only triggered when header is larger than 4k.
- Preserve _revs_limit and instance_start_time after compaction.

Test Suite

- Made the test suite overall more reliable.

View Server

- Provide a UUID to update functions (and all other functions) that they can use to create new docs.
- Upgrade CommonJS modules support to 1.1.1.
- Fixed erlang filter funs and normalize filter fun API.
- Fixed hang in view shutdown.

URL Rewriter & Vhosts

- Allow more complex keys in rewriter.
- Allow global rewrites so system defaults are available in vhosts.
- Allow isolation of databases with vhosts.
- Fix issue with passing variables to query parameters.

16.17.4 Version 0.11.0

Build and System Integration

- Updated and improved source documentation.
- Fixed distribution preparation for building on Mac OS X.
- Added support for building a Windows installer as part of 'make dist'.
- Bug fix for building couch.app's module list.
- ETap tests are now run during make distcheck. This included a number of updates to the build system to properly support VPATH builds.
- Gavin McDonald set up a build-bot instance. More info can be found at http://ci.apache.org/buildbot.html

Futon

- Added a button for view compaction.
- JSON strings are now displayed as-is in the document view, without the escaping of new-lines and quotes. That dramatically improves readability of multi-line strings.
- Same goes for editing of JSON string values. When a change to a field value is submitted, and the value is not valid JSON it is assumed to be a string. This improves editing of multi-line strings a lot.
- Hitting tab in textareas no longer moves focus to the next form field, but simply inserts a tab character at the current caret position.
- Fixed some font declarations.

HTTP Interface

- Provide Content-MD5 header support for attachments.
- Added URL Rewriter handler.
- Added virtual host handling.

Replication

- Added option to implicitly create replication target databases.
- Avoid leaking file descriptors on automatic replication restarts.
- Added option to replicate a list of documents by id.
- Allow continuous replication to be cancelled.

Runtime Statistics

- Statistics are now calculated for a moving window instead of non-overlapping timeframes.
- Fixed a problem with statistics timers and system sleep.
- Moved statistic names to a term file in the priv directory.

Security

- Fixed CVE-2010-0009: Apache CouchDB Timing Attack Vulnerability.
- Added default cookie-authentication and users database.
- Added Futon user interface for user signup and login.
- Added per-database reader access control lists.
- Added per-database security object for configuration data in validation functions.
- Added proxy authentication handler

Storage System

- Adds batching of multiple updating requests, to improve throughput with many writers. Removed the now redundant couch_batch_save module.
- Adds configurable compression of attachments.

View Server

- Added optional 'raw' binary collation for faster view builds where Unicode collation is not important.
- Improved view index build time by reducing ICU collation callouts.
- Improved view information objects.
- Bug fix for partial updates during view builds.
- Move query server to a design-doc based protocol.
- Use json2.js for JSON serialization for compatibility with native JSON.
- Major refactoring of couchjs to lay the groundwork for disabling cURL support. The new HTTP interaction acts like a synchronous XHR. Example usage of the new system is in the JavaScript CLI test runner.

16.18 0.10.x Branch

- *Upgrade Notes*
- *Version 0.10.2*
- *Version 0.10.1*
- *Version 0.10.0*

16.18.1 Upgrade Notes

Warning: *Version 0.10.2* contains important security fixes. Previous *0.10.x* releases are not recommended for regular usage.

Modular Configuration Directories

CouchDB now loads configuration from the following places (glob(7) syntax) in order:

- PREFIX/default.ini
- PREFIX/default.d/*
- PREFIX/local.ini
- PREFIX/local.d/*

The configuration options for *couchdb* script have changed to:

```
-a FILE     add configuration FILE to chain
-A DIR      add configuration DIR to chain
-n          reset configuration file chain (including system default)
-c          print configuration file chain and exit
```

Show and List API change

Show and List functions must have a new structure in 0.10. See Formatting_with_Show_and_List for details.

Stricter enforcing of reduciness in reduce-functions

Reduce functions are now required to reduce the number of values for a key.

View query reduce parameter strictness

CouchDB now considers the parameter `reduce=false` to be an error for queries of map-only views, and responds with status code 400.

16.18.2 Version 0.10.2

Build and System Integration

- Fixed distribution preparation for building on Mac OS X.

Security

- Fixed *CVE-2010-0009: Apache CouchDB Timing Attack Vulnerability*

Replicator

- Avoid leaking file descriptors on automatic replication restarts.

16.18.3 Version 0.10.1

Build and System Integration

- Test suite now works with the distcheck target.

Replicator

- Stability enhancements regarding redirects, timeouts, OAuth.

Query Server

- Avoid process leaks
- Allow list and view to span languages

Stats

- Eliminate new process flood on system wake

16.18.4 Version 0.10.0

Build and System Integration

- Changed *couchdb* script configuration options.
- Added default.d and local.d configuration directories to load sequence.

HTTP Interface

- Added optional cookie-based authentication handler.
- Added optional two-legged OAuth authentication handler.

Storage Format

- Add move headers with checksums to the end of database files for extra robust storage and faster storage.

View Server

- Added native Erlang views for high-performance applications.

16.19 0.9.x Branch

- *Upgrade Notes*
- *Version 0.9.2*
- *Version 0.9.1*
- *Version 0.9.0*

16.19.1 Upgrade Notes

Response to Bulk Creation/Updates

The response to a bulk creation / update now looks like this

```
[
    {"id": "0", "rev": "3682408536"},
    {"id": "1", "rev": "3206753266"},
    {"id": "2", "error": "conflict", "reason": "Document update conflict."}
]
```

Database File Format

The database file format has changed. CouchDB itself does yet not provide any tools for migrating your data. In the meantime, you can use third-party scripts to deal with the migration, such as the dump/load tools that come with the development version (trunk) of couchdb-python.

Renamed "count" to "limit"

The view query API has been changed: `count` has become `limit`. This is a better description of what the parameter does, and should be a simple update in any client code.

Moved View URLs

The view URLs have been moved to design document resources. This means that paths that used to be like:

```
http://hostname:5984/mydb/_view/designname/viewname?limit=10
```

will now look like:

```
http://hostname:5984/mydb/_design/designname/_view/viewname?limit=10.
```

See the REST, Hypermedia, and CouchApps thread on dev for details.

Attachments

Names of attachments are no longer allowed to start with an underscore.

Error Codes

Some refinements have been made to error handling. CouchDB will send 400 instead of 500 on invalid query parameters. Most notably, document update conflicts now respond with *409 Conflict* instead of *412 Precondition Failed*. The error code for when attempting to create a database that already exists is now 412 instead of 409.

ini file format

CouchDB 0.9 changes sections and configuration variable names in configuration files. Old .ini files won't work. Also note that CouchDB now ships with two .ini files where 0.8 used couch.ini there are now *default.ini* and *local.ini*. *default.ini* contains CouchDB's standard configuration values. local.ini is meant for local changes. *local.ini* is not overwritten on CouchDB updates, so your edits are safe. In addition, the new runtime configuration system persists changes to the configuration in *local.ini*.

16.19.2 Version 0.9.2

Build and System Integration

- Remove branch callbacks to allow building couchjs against newer versions of Spidermonkey.

Replication

- Fix replication with 0.10 servers initiated by an 0.9 server (COUCHDB-559).

16.19.3 Version 0.9.1

Build and System Integration

- PID file directory is now created by the SysV/BSD daemon scripts.
- Fixed the environment variables shown by the configure script.
- Fixed the build instructions shown by the configure script.
- Updated ownership and permission advice in *README* for better security.

Configuration and stats system

- Corrected missing configuration file error message.
- Fixed incorrect recording of request time.

Database Core

- Document validation for underscore prefixed variables.
- Made attachment storage less sparse.
- Fixed problems when a database with delayed commits pending is considered idle, and subject to losing changes when shutdown. (COUCHDB-334)

External Handlers

- Fix POST requests.

Futon

- Redirect when loading a deleted view URI from the cookie.

HTTP Interface

- Attachment requests respect the "rev" query-string parameter.

JavaScript View Server

- Useful JavaScript Error messages.

Replication

- Added support for Unicode characters transmitted as UTF-16 surrogate pairs.
- URL-encode attachment names when necessary.
- Pull specific revisions of an attachment, instead of just the latest one.
- Work around a rare chunk-merging problem in ibrowse.
- Work with documents containing Unicode characters outside the Basic Multilingual Plane.

16.19.4 Version 0.9.0

Build and System Integration

- The *couchdb* script now supports system chainable configuration files.
- The Mac OS X daemon script now redirects STDOUT and STDERR like SysV/BSD.
- The build and system integration have been improved for portability.
- Added COUCHDB_OPTIONS to etc/default/couchdb file.
- Remove COUCHDB_INI_FILE and COUCHDB_PID_FILE from etc/default/couchdb file.
- Updated *configure.ac* to manually link *libm* for portability.
- Updated *configure.ac* to extended default library paths.
- Removed inets configuration files.
- Added command line test runner.
- Created dev target for make.

Configuration and stats system

- Separate default and local configuration files.
- HTTP interface for configuration changes.
- Statistics framework with HTTP query API.

Database Core

- Faster B-tree implementation.
- Changed internal JSON term format.
- Improvements to Erlang VM interactions under heavy load.
- User context and administrator role.
- Update validations with design document validation functions.
- Document purge functionality.
- Ref-counting for database file handles.

Design Document Resource Paths

- Added httpd_design_handlers config section.
- Moved _view to httpd_design_handlers.
- Added ability to render documents as non-JSON content-types with _show and _list functions, which are also httpd_design_handlers.

Futon Utility Client

- Added pagination to the database listing page.
- Implemented attachment uploading from the document page.
- Added page that shows the current configuration, and allows modification of option values.
- Added a JSON "source view" for document display.
- JSON data in view rows is now syntax highlighted.
- Removed the use of an iframe for better integration with browser history and bookmarking.
- Full database listing in the sidebar has been replaced by a short list of recent databases.
- The view editor now allows selection of the view language if there is more than one configured.
- Added links to go to the raw view or document URI.
- Added status page to display currently running tasks in CouchDB.
- JavaScript test suite split into multiple files.
- Pagination for reduce views.

HTTP Interface

- Added client side UUIDs for idempotent document creation
- HTTP COPY for documents
- Streaming of chunked attachment PUTs to disk
- Remove negative count feature
- Add include_docs option for view queries
- Add multi-key view post for views
- Query parameter validation
- Use stale=ok to request potentially cached view index
- External query handler module for full-text or other indexers.
- Etags for attachments, views, shows and lists
- Show and list functions for rendering documents and views as developer controlled content-types.
- Attachment names may use slashes to allow uploading of nested directories (useful for static web hosting).
- Option for a view to run over design documents.
- Added newline to JSON responses. Closes bike-shed.

Replication

- Using ibrowse.
- Checkpoint replications so failures are less expensive.
- Automatically retry of failed replications.
- Stream attachments in pull-replication.

16.20 0.8.x Branch

- *Version 0.8.1-incubating*
- *Version 0.8.0-incubating*

16.20.1 Version 0.8.1-incubating

Build and System Integration

- The *couchdb* script no longer uses *awk* for configuration checks as this was causing portability problems.
- Updated *sudo* example in *README* to use the *-i* option, this fixes problems when invoking from a directory the *couchdb* user cannot access.

Database Core

- Fix for replication problems where the write queues can get backed up if the writes aren't happening fast enough to keep up with the reads. For a large replication, this can exhaust memory and crash, or slow down the machine dramatically. The fix keeps only one document in the write queue at a time.
- Fix for databases sometimes incorrectly reporting that they contain 0 documents after compaction.
- CouchDB now uses ibrowse instead of inets for its internal HTTP client implementation. This means better replication stability.

Futon

- The view selector dropdown should now work in Opera and Internet Explorer even when it includes opt-groups for design documents. (COUCHDB-81)

JavaScript View Server

- Sealing of documents has been disabled due to an incompatibility with SpiderMonkey 1.9.
- Improve error handling for undefined values emitted by map functions. (COUCHDB-83)

HTTP Interface

- Fix for chunked responses where chunks were always being split into multiple TCP packets, which caused problems with the test suite under Safari, and in some other cases.
- Fix for an invalid JSON response body being returned for some kinds of views. (COUCHDB-84)
- Fix for connections not getting closed after rejecting a chunked request. (COUCHDB-55)
- CouchDB can now be bound to IPv6 addresses.
- The HTTP *Server* header now contains the versions of CouchDB and Erlang.

16.20.2 Version 0.8.0-incubating

Build and System Integration

- CouchDB can automatically respawn following a server crash.
- Database server no longer refuses to start with a stale PID file.
- System logrotate configuration provided.
- Improved handling of ICU shared libraries.
- The *couchdb* script now automatically enables SMP support in Erlang.
- The *couchdb* and *couchjs* scripts have been improved for portability.
- The build and system integration have been improved for portability.

Database Core

- The view engine has been completely decoupled from the storage engine. Index data is now stored in separate files, and the format of the main database file has changed.
- Databases can now be compacted to reclaim space used for deleted documents and old document revisions.
- Support for incremental map/reduce views has been added.
- To support map/reduce, the structure of design documents has changed. View values are now JSON objects containing at least a *map* member, and optionally a *reduce* member.
- View servers are now identified by name (for example *javascript*) instead of by media type.
- Automatically generated document IDs are now based on proper UUID generation using the crypto module.
- The field *content-type* in the JSON representation of attachments has been renamed to *content_type* (underscore).

Futon

- When adding a field to a document, Futon now just adds a field with an autogenerated name instead of prompting for the name with a dialog. The name is automatically put into edit mode so that it can be changed immediately.
- Fields are now sorted alphabetically by name when a document is displayed.
- Futon can be used to create and update permanent views.
- The maximum number of rows to display per page on the database page can now be adjusted.
- Futon now uses the XMLHTTPRequest API asynchronously to communicate with the CouchDB HTTP server, so that most operations no longer block the browser.

- View results sorting can now be switched between ascending and descending by clicking on the *Key* column header.
- Fixed a bug where documents that contained a @ character could not be viewed. (COUCHDB-12)
- The database page now provides a *Compact* button to trigger database compaction. (COUCHDB-38)
- Fixed portential double encoding of document IDs and other URI segments in many instances. (COUCHDB-39)
- Improved display of attachments.
- The JavaScript Shell has been removed due to unresolved licensing issues.

JavaScript View Server

- SpiderMonkey is no longer included with CouchDB, but rather treated as a normal external dependency. A simple C program (*_couchjs*) is provided that links against an existing SpiderMonkey installation and uses the interpreter embedding API.
- View functions using the default JavaScript view server can now do logging using the global *log(message)* function. Log messages are directed into the CouchDB log at *INFO* level. (COUCHDB-59)
- The global *map(key, value)* function made available to view code has been renamed to *emit(key, value)*.
- Fixed handling of exceptions raised by view functions.

HTTP Interface

- CouchDB now uses MochiWeb instead of inets for the HTTP server implementation. Among other things, this means that the extra configuration files needed for inets (such as *couch_httpd.conf*) are no longer used.
- The HTTP interface now completely supports the *HEAD* method. (COUCHDB-3)
- Improved compliance of *Etag* handling with the HTTP specification. (COUCHDB-13)
- Etags are no longer included in responses to document *GET* requests that include query string parameters causing the JSON response to change without the revision or the URI having changed.
- The bulk document update API has changed slightly on both the request and the response side. In addition, bulk updates are now atomic.
- CouchDB now uses *TCP_NODELAY* to fix performance problems with persistent connections on some platforms due to nagling.
- Including a *?descending=false* query string parameter in requests to views no longer raises an error.
- Requests to unknown top-level reserved URLs (anything with a leading underscore) now return a *unknown_private_path* error instead of the confusing *illegal_database_name*.
- The Temporary view handling now expects a JSON request body, where the JSON is an object with at least a *map* member, and optional *reduce* and *language* members.
- Temporary views no longer determine the view server based on the Content-Type header of the *POST* request, but rather by looking for a *language* member in the JSON body of the request.
- The status code of responses to *DELETE* requests is now 200 to reflect that that the deletion is performed synchronously.

CHAPTER

SEVENTEEN

SECURITY ISSUES / CVES

17.1 CVE-2010-0009: Apache CouchDB Timing Attack Vulnerability

Date
31.03.2010

Affected
Apache CouchDB 0.8.0 to 0.10.1

Severity
Important

Vendor
The Apache Software Foundation

17.1.1 Description

Apache CouchDB versions prior to version *0.11.0* are vulnerable to timing attacks, also known as side-channel information leakage, due to using simple break-on-inequality string comparisons when verifying hashes and passwords.

17.1.2 Mitigation

All users should upgrade to CouchDB *0.11.0*. Upgrades from the *0.10.x* series should be seamless. Users on earlier versions should consult with *upgrade notes*.

17.1.3 Example

A canonical description of the attack can be found in http://codahale.com/a-lesson-in-timing-attacks/

17.1.4 Credit

This issue was discovered by *Jason Davies* of the Apache CouchDB development team.

17.2 CVE-2010-2234: Apache CouchDB Cross Site Request Forgery Attack

Date
21.02.2010

Affected
Apache CouchDB 0.8.0 to 0.11.1

Severity
Important

Vendor
The Apache Software Foundation

17.2.1 Description

Apache CouchDB versions prior to version *0.11.1* are vulnerable to Cross Site Request Forgery (CSRF) attacks.

17.2.2 Mitigation

All users should upgrade to CouchDB *0.11.2* or *1.0.1*.

Upgrades from the *0.11.x* and *0.10.x* series should be seamless.

Users on earlier versions should consult with upgrade notes.

17.2.3 Example

A malicious website can *POST* arbitrary JavaScript code to well known CouchDB installation URLs (like http://localhost:5984/) and make the browser execute the injected JavaScript in the security context of CouchDB's admin interface Futon.

Unrelated, but in addition the JSONP API has been turned off by default to avoid potential information leakage.

17.2.4 Credit

This CSRF issue was discovered by a source that wishes to stay anonymous.

17.3 CVE-2010-3854: Apache CouchDB Cross Site Scripting Issue

Date
28.01.2011

Affected
Apache CouchDB 0.8.0 to 1.0.1

Severity
Important

Vendor
The Apache Software Foundation

17.3.1 Description

Apache CouchDB versions prior to version *1.0.2* are vulnerable to Cross Site Scripting (XSS) attacks.

17.3.2 Mitigation

All users should upgrade to CouchDB *1.0.2*.

Upgrades from the *0.11.x* and *0.10.x* series should be seamless.

Users on earlier versions should consult with upgrade notes.

17.3.3 Example

Due to inadequate validation of request parameters and cookie data in Futon, CouchDB's web-based administration UI, a malicious site can execute arbitrary code in the context of a user's browsing session.

17.3.4 Credit

This XSS issue was discovered by a source that wishes to stay anonymous.

17.4 CVE-2012-5641: Information disclosure via unescaped backslashes in URLs on Windows

Date
14.01.2013

Affected
All Windows-based releases of Apache CouchDB, up to and including 1.0.3, 1.1.1, and 1.2.0 are vulnerable.

Severity
Moderate

Vendor
The Apache Software Foundation

17.4.1 Description

A specially crafted request could be used to access content directly that would otherwise be protected by inbuilt CouchDB security mechanisms. This request could retrieve in binary form any CouchDB database, including the *_users* or *_replication* databases, or any other file that the user account used to run CouchDB might have read access to on the local filesystem. This exploit is due to a vulnerability in the included MochiWeb HTTP library.

17.4.2 Mitigation

Upgrade to a supported CouchDB release that includes this fix, such as:

- *1.0.4*
- *1.1.2*
- *1.2.1*
- *1.3.x*

All listed releases have included a specific fix for the MochiWeb component.

17.4.3 Work-Around

Users may simply exclude any file-based web serving components directly within their configuration file, typically in *local.ini*. On a default CouchDB installation, this requires amending the *httpd_global_handlers/favicon.ico* and *httpd_global_handlers/_utils* lines within *httpd_global_handlers*:

```
[httpd_global_handlers]
favicon.ico = {couch_httpd_misc_handlers, handle_welcome_req, <<"Forbidden">>}
_utils = {couch_httpd_misc_handlers, handle_welcome_req, <<"Forbidden">>}
```

If additional handlers have been added, such as to support Adobe's Flash *crossdomain.xml* files, these would also need to be excluded.

17.4.4 Acknowledgement

The issue was found and reported by Sriram Melkote to the upstream MochiWeb project.

17.4.5 References

- https://github.com/melkote/mochiweb/commit/ac2bf

17.5 CVE-2012-5649: JSONP arbitrary code execution with Adobe Flash

Date
14.01.2013

Affected
Releases up to and including 1.0.3, 1.1.1, and 1.2.0 are vulnerable, if administrators have enabled JSONP.

Severity
Moderate

Vendor
The Apache Software Foundation

17.5.1 Description

A hand-crafted JSONP callback and response can be used to run arbitrary code inside client-side browsers via Adobe Flash.

17.5.2 Mitigation

Upgrade to a supported CouchDB release that includes this fix, such as:

- *1.0.4*
- *1.1.2*
- *1.2.1*
- *1.3.x*

All listed releases have included a specific fix.

17.5.3 Work-Around

Disable JSONP or don't enable it since it's disabled by default.

17.6 CVE-2012-5650: DOM based Cross-Site Scripting via Futon UI

Date
14.01.2013

Affected
Apache CouchDB releases up to and including 1.0.3, 1.1.1, and 1.2.0 are vulnerable.

Severity
Moderate

Vendor
The Apache Software Foundation

17.6.1 Description

Query parameters passed into the browser-based test suite are not sanitised, and can be used to load external resources. An attacker may execute JavaScript code in the browser, using the context of the remote user.

17.6.2 Mitigation

Upgrade to a supported CouchDB release that includes this fix, such as:

- *1.0.4*
- *1.1.2*
- *1.2.1*
- *1.3.x*

All listed releases have included a specific fix.

17.6.3 Work-Around

Disable the Futon user interface completely, by adapting *local.ini* and restarting CouchDB:

```
[httpd_global_handlers]
_utils = {couch_httpd_misc_handlers, handle_welcome_req, <<"Forbidden">>}
```

Or by removing the UI test suite components:

- share/www/verify_install.html
- share/www/couch_tests.html
- share/www/custom_test.html

17.6.4 Acknowledgement

This vulnerability was discovered & reported to the Apache Software Foundation by Frederik Braun.

17.7 CVE-2014-2668: DoS (CPU and memory consumption) via the count parameter to /_uuids

Date
26.03.2014

Affected
Apache CouchDB releases up to and including 1.3.1, 1.4.0, and 1.5.0 are vulnerable.

Severity
Moderate

Vendor
The Apache Software Foundation

17.7.1 Description

The */_uuids* resource's *count* query parameter is able to take unreasonable huge numeric value which leads to exhaustion of server resources (CPU and memory) and to DoS as the result.

17.7.2 Mitigation

Upgrade to a supported CouchDB release that includes this fix, such as:

- *1.5.1*
- *1.6.0*

All listed releases have included a specific fix to

17.7.3 Work-Around

Disable the */_uuids* handler completely, by adapting *local.ini* and restarting CouchDB:

```
[httpd_global_handlers]
_uuids =
```

17.8 CVE-2017-12635: Apache CouchDB Remote Privilege Escalation

Date
14.11.2017

Affected
All Versions of Apache CouchDB

Severity
Critical

Vendor
The Apache Software Foundation

17.8.1 Description

Due to differences in CouchDB's Erlang-based JSON parser and JavaScript-based JSON parser, it is possible to submit _users documents with duplicate keys for *roles* used for access control within the database, including the special case *_admin* role, that denotes administrative users. In combination with *CVE-2017-12636* (Remote Code Execution), this can be used to give non-admin users access to arbitrary shell commands on the server as the database system user.

17.8.2 Mitigation

All users should upgrade to CouchDB *1.7.1* or *2.1.1*.

Upgrades from previous 1.x and 2.x versions in the same series should be seamless.

Users on earlier versions, or users upgrading from 1.x to 2.x should consult with upgrade notes.

17.8.3 Example

The JSON parser differences result in behaviour that if two *roles* keys are available in the JSON, the second one will be used for authorising the document write, but the first *roles* key is used for subsequent authorisation for the newly created user. By design, users can not assign themselves roles. The vulnerability allows non-admin users to give themselves admin privileges.

We addressed this issue by updating the way CouchDB parses JSON in Erlang, mimicking the JavaScript behaviour of picking the last key, if duplicates exist.

17.8.4 Credit

This issue was discovered by Max Justicz.

17.9 CVE-2017-12636: Apache CouchDB Remote Code Execution

Date
14.11.2017

Affected
All Versions of Apache CouchDB

Severity
Critical

Vendor
The Apache Software Foundation

17.9.1 Description

CouchDB administrative users can configure the database server via HTTP(S). Some of the configuration options include paths for operating system-level binaries that are subsequently launched by CouchDB. This allows a CouchDB admin user to execute arbitrary shell commands as the CouchDB user, including downloading and executing scripts from the public internet.

17.9.2 Mitigation

All users should upgrade to CouchDB *1.7.1* or *2.1.1*.

Upgrades from previous 1.x and 2.x versions in the same series should be seamless.

Users on earlier versions, or users upgrading from 1.x to 2.x should consult with upgrade notes.

17.9.3 Credit

This issue was discovered by Joan Touzet of the CouchDB Security team during the investigation of *CVE-2017-12635*.

17.10 CVE-2018-11769: Apache CouchDB Remote Code Execution

Date
08.08.2018

Affected
Apache CouchDB 1.x and 2.1.2

Severity
Low

Vendor
The Apache Software Foundation

17.10.1 Description

CouchDB administrative users can configure the database server via HTTP(S). Due to insufficient validation of administrator-supplied configuration settings via the HTTP API, it is possible for a CouchDB administrator user to escalate their privileges to that of the operating system's user under which CouchDB runs, by bypassing the blacklist of configuration settings that are not allowed to be modified via the HTTP API.

This privilege escalation effectively allows a CouchDB admin user to gain arbitrary remote code execution, bypassing mitigations for *CVE-2017-12636* and *CVE-2018-8007*.

17.10.2 Mitigation

All users should upgrade to CouchDB *2.2.0*.

Upgrades from previous 2.x versions in the same series should be seamless.

Users still on CouchDB 1.x should be advised that the Apache CouchDB team no longer support 1.x.

In-place mitigation (on any 1.x release, or 2.x prior to 2.2.0) is possible by removing the `_config` route from the `default.ini` file, as follows:

```
[httpd_global_handlers]
;_config = {couch_httpd_misc_handlers, handle_config_req}
```

or by blocking access to the */_config* (1.x) or */_node/*/_config* routes at a reverse proxy in front of the service.

17.11 CVE-2018-17188: Apache CouchDB Remote Privilege Escalations

Date
17.12.2018

Affected
All Versions of Apache CouchDB

Severity
Medium

Vendor
The Apache Software Foundation

17.11.1 Description

Prior to CouchDB version 2.3.0, CouchDB allowed for runtime-configuration of key components of the database. In some cases, this lead to vulnerabilities where CouchDB admin users could access the underlying operating system as the CouchDB user. Together with other vulnerabilities, it allowed full system entry for unauthenticated users.

These vulnerabilities were fixed and disclosed in the following CVE reports:

- *CVE-2018-11769: Apache CouchDB Remote Code Execution*
- *CVE-2018-8007: Apache CouchDB Remote Code Execution*
- *CVE-2017-12636: Apache CouchDB Remote Code Execution*
- *CVE-2017-12635: Apache CouchDB Remote Privilege Escalation*

Rather than waiting for new vulnerabilities to be discovered, and fixing them as they come up, the CouchDB development team decided to make changes to avoid this entire class of vulnerabilities.

With CouchDB version 2.3.0, CouchDB no longer can configure key components at runtime. While some flexibility is needed for speciality configurations of CouchDB, the configuration was changed from being available at runtime to start-up time. And as such now requires shell access to the CouchDB server.

This closes all future paths for vulnerabilities of this type.

17.11.2 Mitigation

All users should upgrade to CouchDB *2.3.0*.

Upgrades from previous 2.x versions in the same series should be seamless.

Users on earlier versions should consult with upgrade notes.

17.11.3 Credit

This issue was discovered by the Apple Information Security team.

17.12 CVE-2018-8007: Apache CouchDB Remote Code Execution

Date
30.04.2018

Affected
All Versions of Apache CouchDB

Severity
Low

Vendor
The Apache Software Foundation

17.12.1 Description

CouchDB administrative users can configure the database server via HTTP(S). Due to insufficient validation of administrator-supplied configuration settings via the HTTP API, it is possible for a CouchDB administrator user to escalate their privileges to that of the operating system's user that CouchDB runs under, by bypassing the backlist of configuration settings that are not allowed to be modified via the HTTP API.

This privilege escalation effectively allows a CouchDB admin user to gain arbitrary remote code execution, bypassing *CVE-2017-12636*

17.12.2 Mitigation

All users should upgrade to CouchDB *1.7.2* or *2.1.2*.

Upgrades from previous 1.x and 2.x versions in the same series should be seamless.

Users on earlier versions, or users upgrading from 1.x to 2.x should consult with upgrade notes.

17.12.3 Credit

This issue was discovered by Francesco Oddo of MDSec Labs.

17.13 CVE-2020-1955: Apache CouchDB Remote Privilege Escalation

Date
19.05.2020

Affected
3.0.0

Severity
Medium

Vendor
The Apache Software Foundation

17.13.1 Description

CouchDB version 3.0.0 shipped with a new configuration setting that governs access control to the entire database server called *require_valid_user_except_for_up*. It was meant as an extension to the long-standing setting *require_valid_user*, which in turn requires that any and all requests to CouchDB will have to be made with valid credentials, effectively forbidding any anonymous requests.

The new *require_valid_user_except_for_up* is an off-by-default setting that was meant to allow requiring valid credentials for all endpoints except for the */_up* endpoint.

However, the implementation of this made an error that lead to not enforcing credentials on any endpoint, when enabled.

CouchDB versions *3.0.1* and *3.1.0* fix this issue.

17.13.2 Mitigation

Users who have not enabled *require_valid_user_except_for_up* are not affected.

Users who have it enabled can either disable it again, or upgrade to CouchDB versions *3.0.1* and *3.1.0*

17.13.3 Credit

This issue was discovered by Stefan Klein.

17.14 CVE-2021-38295: Apache CouchDB Privilege Escalation

Date
12.10.2021

Affected
3.1.1 and below

Severity
Low

Vendor
The Apache Software Foundation

17.14.1 Description

A malicious user with permission to create documents in a database is able to attach a HTML attachment to a document. If a CouchDB admin opens that attachment in a browser, e.g. via the CouchDB admin interface Fauxton, any JavaScript code embedded in that HTML attachment will be executed within the security context of that admin. A similar route is available with the already deprecated *_show* and *_list* functionality.

This *privilege escalation* vulnerability allows an attacker to add or remove data in any database or make configuration changes.

17.14.2 Mitigation

CouchDB *3.2.0* and onwards adds *Content-Security-Policy* headers for all attachment, *_show* and *_list* requests. This breaks certain niche use-cases and there are configuration options to restore the previous behaviour for those who need it.

CouchDB *3.1.2* defaults to the previous behaviour, but adds configuration options to turn *Content-Security-Policy* headers on for all affected requests.

17.14.3 Credit

This issue was identified by Cory Sabol of Secure Ideas.

17.15 CVE-2022-24706: Apache CouchDB Remote Privilege Escalation

Date
25.04.2022

Affected
3.2.1 and below

Severity
Critical

Vendor
The Apache Software Foundation

17.15.1 Description

An attacker can access an improperly secured default installation without authenticating and gain admin privileges.

1. CouchDB opens a random network port, bound to all available interfaces in anticipation of clustered operation and/or runtime introspection. A utility process called *epmd* advertises that random port to the network. *epmd* itself listens on a fixed port.
2. CouchDB packaging previously chose a default *cookie* value for single-node as well as clustered installations. That cookie authenticates any communication between Erlang nodes.

The CouchDB documentation has always made recommendations for properly securing an installation, but not all users follow the advice.

We recommend a firewall in front of all CouchDB installations. The full CouchDB api is available on registered port *5984* and this is the only port that needs to be exposed for a single-node install. Installations that do not expose the separate distribution port to external access are not vulnerable.

17.15.2 Mitigation

CouchDB *3.2.2* and onwards will refuse to start with the former default erlang cookie value of *monster*. Installations that upgrade to this versions are forced to choose a different value.

In addition, all binary packages have been updated to bind *epmd* as well as the CouchDB distribution port to *127.0.0.1* and/or *::1* respectively.

17.15.3 Credit

This issue was identified by Alex Vandiver.

17.16 CVE-2023-26268: Apache CouchDB: Information sharing via couchjs processes

Date
02.05.2023

Affected
3.3.1 and below, 3.2.2 and below

Severity
Medium

Vendor
The Apache Software Foundation

17.16.1 Description

Design documents with matching document IDs, from databases on the same cluster, may share a mutable Javascript environment when using these design document functions:

- validate_doc_update
- list
- filter
- filter views (using view functions as filters)
- rewrite
- update

This doesn't affect map/reduce or search (Dreyfus) index functions.

17.16.2 Mitigation

CouchDB *3.3.2* and *3.2.3* and onwards matches Javascript execution processes by database names in addition to design document IDs when processing the affected design document functions.

17.16.3 Workarounds

Avoid using design documents from untrusted sources which may attempt to cache or store data in the Javascript environment.

17.16.4 Credit

This issue was identified by Nick Vatamaniuc

17.17 CVE-2023-45725: Apache CouchDB: Privilege Escalation Using Design Documents

Date
12.12.2023

Affected
3.3.2 and below

Severity
Medium

Vendor
The Apache Software Foundation

17.17.1 Description

Design document functions which receive a user http request object may expose authorization or session cookie headers of the user who accesses the document.

These design document functions are:

- list
- show
- rewrite
- update

An attacker can leak the session component using an HTML-like output, insert the session as an external resource (such as an image), or store the credential in a `_local` document with an "update" function.

For the attack to succeed the attacker has to be able to insert the design documents into the database, then manipulate a user to access a function from that design document.

17.17.2 Mitigation

CouchDB *3.3.3* scrubs the sensitive headers from http request objects passed to the query server execution environment.

For versions older than *3.3.3* this patch applied to the `loop.js` file would also mitigate the issue:

```
diff --git a/share/server/loop.js b/share/server/loop.js
--- a/share/server/loop.js
+++ b/share/server/loop.js
@@ -49,6 +49,20 @@ function create_nouveau_sandbox() {
   return sandbox;
 }

+function scrubReq(args) {
+  var req = args.pop()
+  if (req.method && req.headers && req.peer && req.userCtx) {
+    delete req.cookie
+    for (var p in req.headers) {
+      if (req.headers.hasOwnProperty(p) && ["authorization", "cookie"].indexOf(p.
↪toLowerCase()) !== -1) {
+        delete req.headers[p]
+      }
+    }
+  }
+  args.push(req)
+  return args
+}
+
 // Commands are in the form of json arrays:
 // ["commandname",..optional args...]\n
 //
@@ -85,7 +99,7 @@ var DDoc = (function() {
         var funPath = args.shift();
         var cmd = funPath[0];
         // the first member of the fun path determines the type of operation
-        var funArgs = args.shift();
+        var funArgs = scrubReq(args.shift());
         if (ddoc_dispatch[cmd]) {
           // get the function, call the command with it
           var point = ddoc;
```

17.17.3 Workarounds

Avoid using design documents from untrusted sources which may attempt to access or manipulate request object's headers.

17.17.4 Credit

This issue was found by Natan Nehorai and reported by Or Peles from the JFrog Vulnerability Research Team. It was also independently found by Richard Ellis and Mike Rhodes from IBM/Cloudant.

CHAPTER

EIGHTEEN

REPORTING NEW SECURITY PROBLEMS WITH APACHE COUCHDB

The Apache Software Foundation takes a very active stance in eliminating security problems and denial of service attacks against Apache CouchDB.

We strongly encourage folks to report such problems to our private security mailing list first, before disclosing them in a public forum.

Please note that the security mailing list should only be used for reporting undisclosed security vulnerabilities in Apache CouchDB and managing the process of fixing such vulnerabilities. We cannot accept regular bug reports or other queries at this address. All mail sent to this address that does not relate to an undisclosed security problem in the Apache CouchDB source code will be ignored.

If you need to report a bug that isn't an undisclosed security vulnerability, please use the bug reporting page.

Questions about:

- How to configure CouchDB securely
- If a vulnerability applies to your particular application
- Obtaining further information on a published vulnerability
- Availability of patches and/or new releases

should be address to the users mailing list. Please see the mailing lists page for details of how to subscribe.

The private security mailing address is: security@couchdb.apache.org

Please read how the Apache Software Foundation handles security reports to know what to expect.

Note that all networked servers are subject to denial of service attacks, and we cannot promise magic workarounds to generic problems (such as a client streaming lots of data to your server, or re-requesting the same URL repeatedly). In general our philosophy is to avoid any attacks which can cause the server to consume resources in a non-linear relationship to the size of inputs.

CHAPTER

NINETEEN

ABOUT COUCHDB DOCUMENTATION

19.1 License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
```

(continues on next page)

(continued from previous page)

```
    form, that is based on (or derived from) the Work and for which the
    editorial revisions, annotations, elaborations, or other modifications
    represent, as a whole, an original work of authorship. For the purposes
    of this License, Derivative Works shall not include works that remain
    separable from, or merely link (or bind by name) to the interfaces of,
    the Work and Derivative Works thereof.

    "Contribution" shall mean any work of authorship, including
    the original version of the Work and any modifications or additions
    to that Work or Derivative Works thereof, that is intentionally
    submitted to Licensor for inclusion in the Work by the copyright owner
    or by an individual or Legal Entity authorized to submit on behalf of
    the copyright owner. For the purposes of this definition, "submitted"
    means any form of electronic, verbal, or written communication sent
    to the Licensor or its representatives, including but not limited to
    communication on electronic mailing lists, source code control systems,
    and issue tracking systems that are managed by, or on behalf of, the
    Licensor for the purpose of discussing and improving the Work, but
    excluding communication that is conspicuously marked or otherwise
    designated in writing by the copyright owner as "Not a Contribution."

    "Contributor" shall mean Licensor and any individual or Legal Entity
    on behalf of whom a Contribution has been received by Licensor and
    subsequently incorporated within the Work.

 2. Grant of Copyright License. Subject to the terms and conditions of
    this License, each Contributor hereby grants to You a perpetual,
    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
    copyright license to reproduce, prepare Derivative Works of,
    publicly display, publicly perform, sublicense, and distribute the
    Work and such Derivative Works in Source or Object form.

 3. Grant of Patent License. Subject to the terms and conditions of
    this License, each Contributor hereby grants to You a perpetual,
    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
    (except as stated in this section) patent license to make, have made,
    use, offer to sell, sell, import, and otherwise transfer the Work,
    where such license applies only to those patent claims licensable
    by such Contributor that are necessarily infringed by their
    Contribution(s) alone or by combination of their Contribution(s)
    with the Work to which such Contribution(s) was submitted. If You
    institute patent litigation against any entity (including a
    cross-claim or counterclaim in a lawsuit) alleging that the Work
    or a Contribution incorporated within the Work constitutes direct
    or contributory patent infringement, then any patent licenses
    granted to You under this License for that Work shall terminate
    as of the date such litigation is filed.

 4. Redistribution. You may reproduce and distribute copies of the
    Work or Derivative Works thereof in any medium, with or without
    modifications, and in Source or Object form, provided that You
    meet the following conditions:

    (a) You must give any other recipients of the Work or
        Derivative Works a copy of this License; and
```

(continues on next page)

(continued from previous page)

```
    (b) You must cause any modified files to carry prominent notices
        stating that You changed the files; and

    (c) You must retain, in the Source form of any Derivative Works
        that You distribute, all copyright, patent, trademark, and
        attribution notices from the Source form of the Work,
        excluding those notices that do not pertain to any part of
        the Derivative Works; and

    (d) If the Work includes a "NOTICE" text file as part of its
        distribution, then any Derivative Works that You distribute must
        include a readable copy of the attribution notices contained
        within such NOTICE file, excluding those notices that do not
        pertain to any part of the Derivative Works, in at least one
        of the following places: within a NOTICE text file distributed
        as part of the Derivative Works; within the Source form or
        documentation, if provided along with the Derivative Works; or,
        within a display generated by the Derivative Works, if and
        wherever such third-party notices normally appear. The contents
        of the NOTICE file are for informational purposes only and
        do not modify the License. You may add Your own attribution
        notices within Derivative Works that You distribute, alongside
        or as an addendum to the NOTICE text from the Work, provided
        that such additional attribution notices cannot be construed
        as modifying the License.

    You may add Your own copyright statement to Your modifications and
    may provide additional or different license terms and conditions
    for use, reproduction, or distribution of Your modifications, or
    for any such Derivative Works as a whole, provided Your use,
    reproduction, and distribution of the Work otherwise complies with
    the conditions stated in this License.

  5. Submission of Contributions. Unless You explicitly state otherwise,
    any Contribution intentionally submitted for inclusion in the Work
    by You to the Licensor shall be under the terms and conditions of
    this License, without any additional terms or conditions.
    Notwithstanding the above, nothing herein shall supersede or modify
    the terms of any separate license agreement you may have executed
    with Licensor regarding such Contributions.

  6. Trademarks. This License does not grant permission to use the trade
    names, trademarks, service marks, or product names of the Licensor,
    except as required for reasonable and customary use in describing the
    origin of the Work and reproducing the content of the NOTICE file.

  7. Disclaimer of Warranty. Unless required by applicable law or
    agreed to in writing, Licensor provides the Work (and each
    Contributor provides its Contributions) on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied, including, without limitation, any warranties or conditions
    of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
    PARTICULAR PURPOSE. You are solely responsible for determining the
    appropriateness of using or redistributing the Work and assume any
    risks associated with Your exercise of permissions under this License.
```

(continues on next page)

(continued from previous page)

```
 8. Limitation of Liability. In no event and under no legal theory,
    whether in tort (including negligence), contract, or otherwise,
    unless required by applicable law (such as deliberate and grossly
    negligent acts) or agreed to in writing, shall any Contributor be
    liable to You for damages, including any direct, indirect, special,
    incidental, or consequential damages of any character arising as a
    result of this License or out of the use or inability to use the
    Work (including but not limited to damages for loss of goodwill,
    work stoppage, computer failure or malfunction, or any and all
    other commercial damages or losses), even if such Contributor
    has been advised of the possibility of such damages.

 9. Accepting Warranty or Additional Liability. While redistributing
    the Work or Derivative Works thereof, You may choose to offer,
    and charge a fee for, acceptance of support, warranty, indemnity,
    or other liability obligations and/or rights consistent with this
    License. However, in accepting such obligations, You may act only
    on Your own behalf and on Your sole responsibility, not on behalf
    of any other Contributor, and only if You agree to indemnify,
    defend, and hold each Contributor harmless for any liability
    incurred by, or claims asserted against, such Contributor by reason
    of your accepting any such warranty or additional liability.

 END OF TERMS AND CONDITIONS

 APPENDIX: How to apply the Apache License to your work.

    To apply the Apache License to your work, attach the following
    boilerplate notice, with the fields enclosed by brackets "[]"
    replaced with your own identifying information. (Don't include
    the brackets!)  The text should be enclosed in the appropriate
    comment syntax for the file format. We also recommend that a
    file or class name and description of purpose be included on the
    same "printed page" as the copyright notice for easier
    identification within third-party archives.

 Copyright 2020 The Apache Foundation

 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
```

CHAPTER

TWENTY

CONTRIBUTING TO THIS DOCUMENTATION

The documentation lives in its own source tree. We'll start by forking and cloning the CouchDB documentation GitHub mirror. That will allow us to send the contribution to CouchDB with a pull request.

If you don't have a GitHub account yet, it is a good time to get one, they are free. If you don't want to use GitHub, there are alternate ways to contributing back, that we'll cover next time.

Go to https://github.com/apache/couchdb and click the "fork" button in the top right. This will create a fork of CouchDB in your GitHub account. If your account is *username*, your fork lives at https://github.com/username/couchdb. In the header, it tells me my "GitHub Clone URL". We need to copy that and start a terminal:

```
$ git clone https://github.com/username/couchdb.git
$ cd couchdb/src/docs
$ subl .
```

I'm opening the whole CouchDB documentation source tree in my favourite editor. It gives me the usual directory listing:

```
ebin/
ext/
.git/
.gitignore
images/
LICENSE
make.bat
Makefile
NOTICE
rebar.config
src/
static/
templates/
themes/
.travis.yml
```

The documentation sources live in *src/docs/src*, you can safely ignore all the other files and directories.

First we should determine where we want to document this inside the documentation. We can look through http://docs.couchdb.org/en/latest/ for inspiration. The JSON Structure Reference looks like a fine place to write this up.

The current state includes mostly tables describing the JSON structure (after all, that's the title of this chapter), but some prose about the number representation can't hurt. For future reference, since the topic in the thread includes views and different encoding in views (as opposed to the storage engine), we should remember to make a note in the views documentation as well, but we'll leave this for later.

Let's try and find the source file that builds the file http://docs.couchdb.org/en/latest/json-structure.html – we are in luck, under *share/doc/src* we find the file *json-structure.rst*. That looks promising. *.rst* stands for ReStructured Text (see http://thomas-cokelaer.info/tutorials/sphinx/rest_syntax.html for a markup reference), which is an ASCII format for writing documents, documentation in this case. Let's have a look and open it.

We see ASCII tables with some additional formatting, all looking like the final HTML. So far so easy. For now, let's just add to the bottom of this. We can worry about organising this better later.

We start by adding a new headline:

```
Number Handling
===============
```

Now we paste in the rest of the main email of the thread. It is mostly text, but it includes some code listings. Let's mark them up. We'll turn:

```
ejson:encode(ejson:decode(<<"1.1">>)).
<<"1.1000000000000000888">>
```

Into:

```
.. code-block:: erlang

    ejson:encode(ejson:decode(<<"1.1">>)).
    <<"1.1000000000000000888">>
```

And we follow along with the other code samples. We turn:

```
Spidermonkey

$ js -h 2>&1 | head -n 1
JavaScript-C 1.8.5 2011-03-31
$ js
js> JSON.stringify(JSON.parse("1.01234567890123456789012345678901234567890"))
"1.0123456789012346"
js> var f = JSON.stringify(JSON.parse("1.01234567890123456789012345678901234567890"))
js> JSON.stringify(JSON.parse(f))
"1.0123456789012346"
```

into:

```
Spidermonkey::

    $ js -h 2>&1 | head -n 1
    JavaScript-C 1.8.5 2011-03-31
    $ js
    js> JSON.stringify(JSON.parse("1.01234567890123456789012345678901234567890"))
    "1.0123456789012346"
    js> var f = JSON.stringify(JSON.parse("1.01234567890123456789012345678901234567890
↪"))
    js> JSON.stringify(JSON.parse(f))
    "1.0123456789012346"
```

And then follow all the other ones.

I cleaned up the text a little but to make it sound more like a documentation entry as opposed to a post on a mailing list.

The next step would be to validate that we got all the markup right. I'll leave this for later. For now we'll contribute our change back to CouchDB.

First, we commit our changes:

```
$ > git commit -am 'document number encoding'
[main a84b2cf] document number encoding
1 file changed, 199 insertions(+)
```

Then we push the commit to our CouchDB fork:

```
$ git push origin main
```

Next, we go back to our GitHub page https://github.com/username/couchdb and click the "Pull Request" button. Fill in the description with something useful and hit the "Send Pull Request" button.

And we're done!

20.1 Style Guidelines for this Documentation

When you make a change to the documentation, you should make sure that you follow the style. Look through some files and you will see that the style is quite straightforward. If you do not know if your formatting is in compliance with the style, ask yourself the following question:

```
Is it needed for correct syntax?
```

If the answer is `No.` then it is probably not.

These guidelines strive be simple, without contradictions and exceptions. The best style is the one that is followed because it seems to be the natural way of doing it.

20.1.1 The guidelines

The guidelines are in descending priority.

1. Syntax
   - Correct syntax is always more important than style. This includes configuration files, HTML responses, etc.
2. Encoding
   - All files are `UTF-8`.
3. Line ending
   - All lines end with `\n`.
   - No trailing whitespace.
4. Line length
   - The maximum line length is `90` characters.
5. Links
   - All internal links are relative.
6. Indentation
   - `4` spaces.
7. Titles
   - The highest level titles in a file is over and underlined with =.
   - Lower level titles are underlined with the following characters in descending order:

     ```
     = - ^ *  + # ` : . " ~ _
     ```

   - Over and underline match the title length.
8. Empty lines
   - No empty line at the end of the file.

- Lists may separated each item with an empty line.

API QUICK REFERENCE

/

GET /, 307

/_active_tasks

GET /_active_tasks, 308

/_all_dbs

GET /_all_dbs, 310

/_cluster_setup

GET /_cluster_setup, 314
POST /_cluster_setup, 315

/_db_updates

GET /_db_updates, 316

/_dbs_info

GET /_dbs_info, 311
POST /_dbs_info, 312

/_membership

GET /_membership, 318

/_node

GET /_node/{node-name}, 334
GET /_node/{node-name}/_config, 364
GET /_node/{node-name}/_config/{section}, 366
GET /_node/{node-name}/_config/{section}/{key}, 367
GET /_node/{node-name}/_prometheus, 337
GET /_node/{node-name}/_smoosh/status, 340
GET /_node/{node-name}/_stats, 335
GET /_node/{node-name}/_system, 342
GET /_node/{node-name}/_versions, 343
POST /_node/{node-name}/_config/_reload, 369
POST /_node/{node-name}/_restart, 343
PUT /_node/{node-name}/_config/{section}/{key}, 368
DELETE /_node/{node-name}/_config/{section}/{key}, 368

/_nouveau_analyze

POST /_nouveau_analyze, 345

/_replicate

POST /_replicate, 319

/_reshard

GET /_reshard, 348
GET /_reshard/jobs, 351
GET /_reshard/jobs/{jobid}, 352
GET /_reshard/jobs/{jobid}/state, 355
GET /_reshard/state, 349
POST /_reshard/jobs, 353
PUT /_reshard/jobs/{jobid}/state, 356
PUT /_reshard/state, 350
DELETE /_reshard/jobs/{jobid}, 354

/_scheduler

GET /_scheduler/docs, 327
GET /_scheduler/docs/{replicator_db}, 330
GET /_scheduler/docs/{replicator_db}/{docid}, 332
GET /_scheduler/jobs, 325

/_search_analyze

POST /_search_analyze, 344

/_session

GET /_session, 359
POST /_session, 358
DELETE /_session, 360

/_up

GET /_up, 346

/_utils

GET /_utils, 346
GET /_utils/, 346

/_uuids

GET /_uuids, 347

/favicon.ico

GET /favicon.ico, 348

/{db}

HEAD /{db}, 370
HEAD /{db}/_design/{ddoc}, 483

HEAD /{db}/_design/{ddoc}/{attname}, 484
HEAD /{db}/{docid}, 458
HEAD /{db}/{docid}/{attname}, 478
GET /{db}, 370
GET /{db}/_all_docs, 378
GET /{db}/_changes, 426
GET /{db}/_design/{ddoc}, 483
GET /{db}/_design/{ddoc}/_info, 485
GET /{db}/_design/{ddoc}/_list/{func}/{other-ddoc}/{view}, 511
GET /{db}/_design/{ddoc}/_list/{func}/{view}, 510
GET /{db}/_design/{ddoc}/_nouveau/{index}, 506
GET /{db}/_design/{ddoc}/_nouveau_info/{index}, 507
GET /{db}/_design/{ddoc}/_search/{index}, 503
GET /{db}/_design/{ddoc}/_search_info/{index}, 505
GET /{db}/_design/{ddoc}/_show/{func}, 508
GET /{db}/_design/{ddoc}/_show/{func}/{docid}, 509
GET /{db}/_design/{ddoc}/_view/{view}, 487
GET /{db}/_design/{ddoc}/{attname}, 484
GET /{db}/_design_docs, 380
GET /{db}/_index, 414
GET /{db}/_local/{docid}, 526
GET /{db}/_local_docs, 521
GET /{db}/_partition/{partition_id}, 518
GET /{db}/_partition/{partition_id}/_all_docs, 518
GET /{db}/_partition/{partition_id}/_design/{ddoc}/_view/{view}, 519
GET /{db}/_purged_infos, 451
GET /{db}/_purged_infos_limit, 452
GET /{db}/_revs_limit, 456
GET /{db}/_security, 445
GET /{db}/_shards, 423
GET /{db}/_shards/{docid}, 424
GET /{db}/{docid}, 459
GET /{db}/{docid}/{attname}, 479
POST /{db}, 375
POST /{db}/_all_docs, 379
POST /{db}/_all_docs/queries, 383
POST /{db}/_bulk_docs, 391
POST /{db}/_bulk_get, 387
POST /{db}/_changes, 430
POST /{db}/_compact, 440
POST /{db}/_compact/{ddoc}, 442
POST /{db}/_design/{ddoc}/_list/{func}/{other-ddoc}/{view}, 511
POST /{db}/_design/{ddoc}/_list/{func}/{view}, 510
POST /{db}/_design/{ddoc}/_show/{func}, 508
POST /{db}/_design/{ddoc}/_show/{func}/{docid}, 509
POST /{db}/_design/{ddoc}/_update/{func}, 512
POST /{db}/_design/{ddoc}/_view/{view}, 489
POST /{db}/_design/{ddoc}/_view/{view}/queries, 500
POST /{db}/_design_docs, 382
POST /{db}/_design_docs/queries, 386
POST /{db}/_ensure_full_commit, 443
POST /{db}/_explain, 418
POST /{db}/_find, 397
POST /{db}/_index, 411
POST /{db}/_index/_bulk_delete, 416
POST /{db}/_local_docs, 523
POST /{db}/_local_docs/queries, 524
POST /{db}/_missing_revs, 453
POST /{db}/_partition/{partition_id}/_explain, 521
POST /{db}/_partition/{partition_id}/_find, 520
POST /{db}/_purge, 448
POST /{db}/_revs_diff, 455
POST /{db}/_sync_shards, 425
POST /{db}/_view_cleanup, 444
PUT /{db}, 372
PUT /{db}/_design/{ddoc}, 483
PUT /{db}/_design/{ddoc}/_update/{func}/{docid}, 513
PUT /{db}/_design/{ddoc}/{attname}, 485
PUT /{db}/_local/{docid}, 526
PUT /{db}/_purged_infos_limit, 452
PUT /{db}/_revs_limit, 457
PUT /{db}/_security, 446
PUT /{db}/_view/{viewid}, 461
PUT /{db}/{docid}/{attname}, 480
DELETE /{db}, 374
DELETE /{db}/_design/{ddoc}, 484
DELETE /{db}/_design/{ddoc}/{attname}, 485
DELETE /{db}/_index/{design_doc}/json/{name}, 415
DELETE /{db}/_local/{docid}, 526
DELETE /{db}/{docid}, 462
DELETE /{db}/{docid}/{attname}, 481
COPY /{db}/_design/{ddoc}, 484
COPY /{db}/_local/{docid}, 526
COPY /{db}/{docid}, 464
ANY /{db}/_design/{ddoc}/_rewrite/{path}, 514

CONFIGURATION QUICK REFERENCE

admins

admins, 231

attachments

attachments, 254
compressible_types, 254
compression_level, 254

chttpd

chttpd, 222
admin_only_all_dbs, 225
allow_jsonp, 223
authentication_handlers, 222
bind_address, 222
buffer_response, 223
bulk_get_use_batches, 225
changes_timeout, 223
config_whitelist, 223
disconnect_check_jitter_msec, 225
disconnect_check_msec, 225
enable_cors, 223
enable_xframe_options, 224
max_http_request_size, 224
port, 222
prefer_minimal, 222
require_valid_user, 233
require_valid_user_except_for_up, 233
secure_rewrites, 224
server_header_versions, 225
x_forwarded_host, 224
x_forwarded_proto, 224
x_forwarded_ssl, 224

chttpd_auth

chttpd_auth, 233
allow_persistent_cookies, 233
auth_cache_size, 233
authentication_redirect, 234
cookie_domain, 233
hash_algorithms, 234
iterations, 234
max_iterations, 235
min_iterations, 234
password_regexp, 235
proxy_use_secret, 235
public_fields, 235
same_site, 233
secret, 236
timeout, 236
users_db_public, 236
x_auth_roles, 236
x_auth_token, 236
x_auth_username, 236

cluster

cluster, 218
n, 218
placement, 218
q, 218
reconnect_interval_sec, 219
seedlist, 219

cors

cors, 228
credentials, 229
headers, 229
max_age, 229
methods, 229
origins, 229

couch_peruser

couch_peruser, 220
delete_dbs, 220
enable, 220
q, 220

couchdb

couchdb, 215
attachment_stream_buffer_size, 215
database_dir, 215
default_security, 215
enable_database_recovery, 215
file_compression, 215
maintenance_mode, 216
max_dbs_open, 216
max_document_size, 216
os_process_timeout, 217
single_node, 217
uri_file, 217
users_db_security_editable, 217
users_db_suffix, 217
util_driver_dir, 217

uuid, 217
view_index_dir, 217
write_xxhash_checksums, 218

csp

csp, 257
attachments_enable, 257
attachments_header_value, 257
enable, 257
header_value, 257
showlist_enable, 257
showlist_header_value, 257
utils_enable, 257
utils_header_value, 257

database_compaction

database_compaction, 239
checkpoint_after, 239
doc_buffer_size, 239

disk_monitor

disk_monitor, 221
background_view_indexing_threshold, 221
enable, 221
interactive_database_writes_threshold, 221
interactive_view_indexing_threshold, 221

dreyfus

dreyfus, 252
limit, 252
limit_partitions, 252
max_limit, 253
max_limit_partitions, 253
name, 252
retry_limit, 252

httpd

httpd, 225
server_options, 225
socket_options, 226

ioq

ioq, 242
concurrency, 242
ratio, 242

ioq.bypass

ioq.bypass, 242
compaction, 242
os_process, 242
read, 242
reshard, 242
shard_sync, 242
view_update, 242
write, 242

jwt_auth

jwt_auth, 236
required_claims, 237
roles_claim_name, 237
roles_claim_path, 237

ken

ken, 241
batch_channels, 241
incremental_channels, 241
max_incremental_updates, 241

ken.ignore

ken.ignore, 241

log

log, 243
file, 243
include_sasl, 244
level, 243
syslog_appid, 244
syslog_facility, 244
syslog_host, 244
syslog_port, 244
write_buffer, 243
write_delay, 243
writer, 243

mango

mango, 253
default_limit, 253
index_all_disabled, 253
index_scan_warning_threshold, 253

native_query_servers

native_query_servers, 252

nouveau

nouveau, 253
enable, 253
max_pipeline_size, 253
max_sessions, 253
url, 253

prometheus

prometheus, 258
additional_port, 258
bind_address, 258
port, 258

purge

purge, 258
index_lag_warn_seconds, 258
max_document_id_number, 258
max_revisions_number, 258

query_server_config

query_server_config, 251
commit_freq, 251

os_process_limit, 251
os_process_soft_limit, 251
reduce_limit, 251

replicator

replicator, 245
auth_plugins, 249
cert_file, 248
checkpoint_interval, 248
connection_timeout, 246
http_connections, 246
ibrowse_options, 247
interval, 245
key_file, 248
max_churn, 245
max_history, 245
max_jobs, 245
password, 248
priority_coeff, 249
retries_per_request, 246
socket_options, 246
ssl_certificate_max_depth, 249
ssl_trusted_certificates_file, 249
update_docs, 246
usage_coeff, 249
use_bulk_get, 248
use_checkpoints, 248
valid_endpoint_protocols, 247
valid_endpoint_protocols_log, 247
valid_ibrowse_options, 247
valid_proxy_protocols, 247
valid_socket_options, 247
verify_ssl_certificates, 249
verify_ssl_certificates_log, 247
worker_batch_size, 246
worker_processes, 246

replicator.shares

replicator.shares, 250
{replicator_db}, 250

resharding

resharding, 259
delete_source, 259
max_history, 259
max_jobs, 259
max_retries, 259
require_node_param, 259
require_range_param, 260
retry_interval_sec, 259
source_close_timeout_sec, 259
update_shard_map_timeout_sec, 259

rexi

rexi, 219
buffer_count, 219
server_per_node, 219
stream_limit, 219

smoosh

smoosh, 240
cleanup_index_files, 240
db_channels, 240
staleness, 240
view_channels, 240
wait_secs, 240

smoosh.{channel-name}

smoosh.{channel-name}, 240
capacity, 240
concurrency, 240
from, 240
max_priority, 240
max_size, 240
min_changes, 241
min_priority, 241
min_size, 241
priority, 241
strict_window, 240
to, 240

ssl

ssl, 226
cacert_file, 227
cert_file, 227
ciphers, 228
fail_if_no_peer_cert, 228
key_file, 227
password, 227
secure_renegotiate, 228
ssl_certificate_max_depth, 227
tls_versions, 228
verify_fun, 227
verify_ssl_certificates, 228

stats

stats, 254
interval, 254

uuids

uuids, 254
algorithm, 254
max_count, 256
utc_id_suffix, 256

vendor

vendor, 256

vhosts

vhosts, 230

view_compaction

view_compaction, 239
keyvalue_buffer_size, 239

INDEX

Symbols

_approx_count_distinct (*global variable or constant*), 101
_count (*global variable or constant*), 101
_stats (*global variable or constant*), 102
_sum (*global variable or constant*), 102

E

Emit() (*built-in function*), 559
emit() (*built-in function*), 556

F

filterfun() (*built-in function*), 107
FoldRows() (*built-in function*), 560

G

GetRow() (*built-in function*), 560
getRow() (*built-in function*), 556

I

isArray() (*built-in function*), 556

J

JSON (*global variable or constant*), 556

L

listfun() (*built-in function*), 106
Log() (*built-in function*), 560
log() (*built-in function*), 556

M

mapfun() (*built-in function*), 100

P

provides() (*built-in function*), 557

R

redfun() (*built-in function*), 101
registerType() (*built-in function*), 557
require() (*built-in function*), 557
RFC
    RFC 1738, 68
    RFC 2109, 358
    RFC 2119, 68
    RFC 2616, 34, 631
    RFC 2616#section-14.27, 483
    RFC 2617, 357
    RFC 2618, 229
    RFC 2817, 229
    RFC 2822, 320
    RFC 3986, 68
    RFC 4122, 68
    RFC 4627, 68
    RFC 5322, 77
    RFC 5789, 229
    RFC 6454, 230

S

Send() (*built-in function*), 561
send() (*built-in function*), 557
showfun() (*built-in function*), 104
Start() (*built-in function*), 561
start() (*built-in function*), 558
sum() (*built-in function*), 558

T

toJSON() (*built-in function*), 558

U

updatefun() (*built-in function*), 107

V

validatefun() (*built-in function*), 110