Pinot Documentation

Release 0.016

Pinot development team

Jan 25, 2019

Contents

1 About Pinot 1

2 Architecture 3

3 Quick Demo 7

4 Reference 9

5 Customizing Pinot 23

6 Design Documents 35

7 Design Proposals 59

CHAPTER 1

About Pinot

Pinot is a realtime distributed OLAP datastore, which is used at LinkedIn to deliver scalable real time analytics with low latency. It can ingest data from offline data sources (such as Hadoop and flat files) as well as streaming events (such as Kafka). Pinot is designed to scale horizontally, so that it can scale to larger data sets and higher query rates as needed.

1.1 What is it for (and not)?

Pinot is well suited for analytical use cases on immutable append-only data that require low latency between an event being ingested and it being available to be queried.

1.2 Key Features

- A column-oriented database with various compression schemes such as Run Length, Fixed Bit Length
- Pluggable indexing technologies - Sorted Index, Bitmap Index, Inverted Index
- Ability to optimize query/execution plan based on query and segment metadata .
- Near real time ingestion from streams and batch ingestion from Hadoop
- SQL like language that supports selection, aggregation, filtering, group by, order by, distinct queries on data.
- Support for multivalued fields
- Horizontally scalable and fault tolerant

Because of the design choices we made to achieve these goals, there are certain limitations in Pinot:

- Pinot is not a replacement for database i.e it cannot be used as source of truth store, cannot mutate data
- Not a replacement for search engine i.e Full text search, relevance not supported
- Query cannot span across multiple tables.

Pinot works very well for querying time series data with lots of Dimensions and Metrics. For example:

```
SELECT sum(clicks), sum(impressions) FROM AdAnalyticsTable
  WHERE ((daysSinceEpoch >= 17849 AND daysSinceEpoch <= 17856)) AND accountId IN␣
↪(123456789)
  GROUP BY daysSinceEpoch TOP 100
```

```
SELECT sum(impressions) FROM AdAnalyticsTable
  WHERE (daysSinceEpoch >= 17824 and daysSinceEpoch <= 17854) AND adveriserId =
↪'1234356789'
  GROUP BY daysSinceEpoch,advertiserId TOP 100
```

```
SELECT sum(cost) FROM AdAnalyticsTable GROUP BY advertiserId TOP 50
```

CHAPTER 2

Architecture

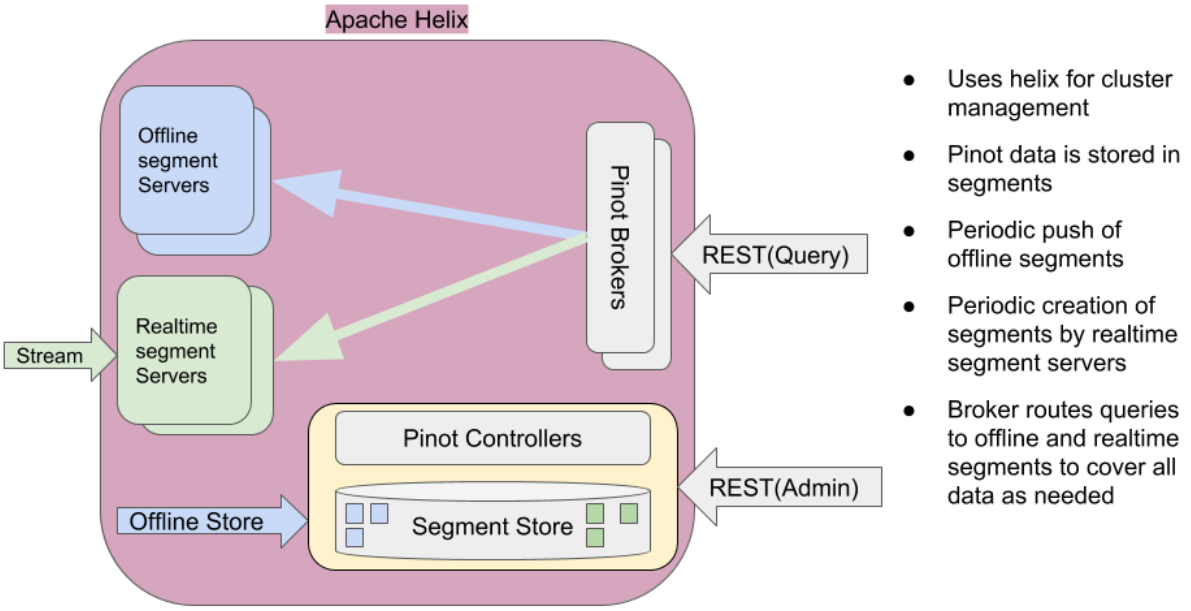


Fig. 1: Pinot Architecture Overview

2.1 Terminology

Table A table is a logical abstraction to refer to a collection of related data. It consists of columns and rows (documents).

Segment Data in table is divided into (horizontal) shards referred to as segments.

2.2 Pinot Components

Pinot Controller Manages other pinot components (brokers, servers) as well as controls assignment of tables/segments to servers.

Pinot Server Hosts one or more segments and serves queries from those segments

Pinot Broker Accepts queries from clients and routes them to one or more servers, and returns consolidated response to the client.

Pinot leverages Apache Helix for cluster management. Helix is a cluster management framework to manage replicated, partitioned resources in a distributed system. Helix uses Zookeeper to store cluster state and metadata.

Briefly, Helix divides nodes into three logical components based on their responsibilities:

Participant The nodes that host distributed, partitioned resources

Spectator The nodes that observe the current state of each Participant and use that information to access the resources. Spectators are notified of state changes in the cluster (state of a participant, or that of a partition in a participant).

Controller The node that observes and controls the Participant nodes. It is responsible for coordinating all transitions in the cluster and ensuring that state constraints are satisfied while maintaining cluster stability

Pinot Controller hosts Helix Controller, in addition to hosting REST APIs for Pinot cluster administration and data ingestion. There can be multiple instances of Pinot controller for redundancy. If there are multiple controllers, Pinot expects that all of them are configured with the same back-end storage system so that they have a common view of the segments (*e.g.* NFS). Pinot can use other storage systems such as HDFS or ADLS.

Pinot Servers are modeled as Helix Participants, hosting Pinot tables (referred to as *resources* in helix terminology). Segments of a table are modeled as Helix partitions (of a resource). Thus, a Pinot server hosts one or more helix partitions of one or more helix resources (*i.e.* one or more segments of one or more tables).

Pinot Brokers are modeled as Spectators. They need to know the location of each segment of a table (and each replica of the segments) and route requests to the appropriate server that hosts the segments of the table being queried. The broker ensures that all the rows of the table are queried exactly once so as to return correct, consistent results for a query. The brokers (or servers) may optimize to prune some of the segments as long as accuracy is not satisfied. In case of hybrid tables, the brokers ensure that the overlap between realtime and offline segment data is queried exactly once. Helix provides the framework by which spectators can learn the location (*i.e.* participant) in which each partition of a resource resides. The brokers use this mechanism to learn the servers that host specific segments of a table.

2.3 Pinot Tables

Pinot supports realtime, or offline, or hybrid tables. Data in Pinot tables is contained in the segments belonging to that table. A Pinot table is modeled as a Helix resource. Each segment of a table is modeled as a Helix Partition,

Table Schema defines column names and their metadata. Table configuration and schema is stored in zookeeper.

Offline tables ingest pre-built pinot-segments from external data stores, whereas Reatime tables ingest data from streams (such as Kafka) and build segments.

A hybrid Pinot table essentially has both realtime as well as offline tables. In such a table, offline segments may be pushed periodically (say, once a day). The retention on the offline table can be set to a high value (say, a few years) since segments are coming in on a periodic basis, whereas the retention on the realtime part can be small (say, a few days). Once an offline segment is pushed to cover a recent time period, the brokers automatically switch to using the

offline table for segments in _that_ time period, and use realtime table only to cover later segments for which offline data may not be available yet.

Note that the query does not know the existence of offline or realtime tables. It only specifies the table name in the query.

2.3.1 Ingesting Offline data

Segments for offline tables are constructed outside of Pinot, typically in Hadoop via map-reduce jobs and ingested into Pinot via REST API provided by the Controller. Pinot provides libraries to create Pinot segments out of input files in AVRO, JSON or CSV formats in a hadoop job, and push the constructed segments to the controlers via REST APIs.

When an Offline segment is ingested, the controller looks up the table's configuration and assigns the segment to the servers that host the table. It may assign multiple servers for each servers depending on the number of replicas configured for that table.

Pinot supports different segment assignment strategies that are optimized for various use cases.

Once segments are assigned, Pinot servers get notified via Helix to "host" the segment. The servers download the segments (as a cached local copy to serve queries) and load them into local memory. All segment data is maintained in memory as long as the server hosts that segment.

Once the server has loaded the segment, Helix notifies brokers of the availability of these segments. The brokers start include the new segments for queries. Brokers support different routing strategies depending on the type of table, the segment assignment strategy and the use case.

Data in offline segments are immmutable (Rows cannot be added, deleted, or modified). However, segments may be replaced modified data.

2.3.2 Ingesting Realtime Data

Segments for realtime tables are constructed by Pinot servers. The servers ingest rows from realtime streams (such as Kafka) until some completion threshold (such as number of rows, or a time threshold) and build a segment out of those rows. Depending on the type of ingestion mechanism used (stream or partition level), segments may be locally stored in the servers or in the controller's segment store.

Multiple servers may ingest the same data to increase availability and share query load.

Once a realtime segment is built and loaded the servers continue to consume from where they left off.

Realtime segments are immutable once they are completed. While realtime segments are being consumed they are mutable, in the sense that new rows can be added to them. Rows cannot be deleted from segments.

See *realtime design* for details.

2.4 Pinot Segments

A segment is laid out in a columnar format so that it can be directly mapped into memory for serving queries. Columns may be single or multi-valued. Column types may be STRING, INT, LONG, FLOAT, DOUBLE or BYTES. Columns may be declared to be metric or dimension (or specifically as a time dimension) in the schema.

Pinot uses dictionary encoding to store values as a dictionary ID. Columns may be configured to be "no-dictionary" column in which case raw values are stored. Dictionary IDs are encoded using minimum number of bits for efficient storage (_e.g._ a column with cardinality of 3 will use only 3 bits for each dictionary ID).

There is a forward index for each column compressed appropriately for efficient memory use. In addition, optional inverted indices can be configured for any set of columns. Inverted indices, while taking up more storage, offer better query performance.

Specialized indexes like StartTree index is also supported.

CHAPTER 3

Quick Demo

A quick way to get familiar with Pinot is to run the Pinot examples. The examples can be run either by compiling the code or by running the prepackaged Docker images.

To demonstrate Pinot, let's start a simple one node cluster, along with the required Zookeeper. This demo setup also creates a table, generates some Pinot segments, then uploads them to the cluster in order to make them queryable.

All of the setup is automated, so the only thing required at the beginning is to start the demonstration cluster.

3.1 Compiling the code

Note: You can skip this step if you are planning to run the pre-built docker image. Make sure you have Docker installed. Some of the newer query features may not be available in docker as of this writing

One can also run the Pinot demonstration by checking out the code on GitHub, compiling it, and running it. Compiling Pinot requires JDK 8 or later and Apache Maven 3.

1. Check out the code from GitHub (https://github.com/apache/incubator-pinot)
2. With Maven installed, run `mvn install package -DskipTests` in the directory in which you checked out Pinot.
3. Make the generated scripts executable `cd pinot-distribution/target/pinot-0.016-pkg; chmod +x bin/*.sh`

3.2 Trying out Offline quickstart demo

To run the demo with docker `docker run -it -p 9000:9000 linkedin/pinot-quickstart-offline`

To run the demo with compiled code: `bin/quick-start-offline.sh`

Once the Pinot cluster is running, you can query it by going to http://localhost:9000/query/

You can also use the REST API to query Pinot, as well as the Java client. As this is outside of the scope of this introduction, the reference documentation to use the Pinot client APIs is in the *Executing queries via REST API on the Broker* section.

Pinot uses PQL, a SQL-like query language, to query data. Here are some sample queries:

```
/*Total number of documents in the table*/
SELECT count(*) FROM baseballStats LIMIT 0

/*Top 5 run scorers of all time*/
SELECT sum('runs') FROM baseballStats GROUP BY playerName TOP 5 LIMIT 0

/*Top 5 run scorers of the year 2000*/
SELECT sum('runs') FROM baseballStats WHERE yearID=2000 GROUP BY playerName TOP 5␣
↪LIMIT 0

/*Top 10 run scorers after 2000*/
SELECT sum('runs') FROM baseballStats WHERE yearID>=2000 GROUP BY playerName

/*Select playerName,runs,homeRuns for 10 records from the table and order them by␣
↪yearID*/
SELECT playerName,runs,homeRuns FROM baseballStats ORDER BY yearID LIMIT 10
```

The full reference for the PQL query language is present in the *PQL* section of the Pinot documentation.

3.3 Trying out Realtime quickstart demo

Pinot can ingest data from streaming sources such as Kafka.

To run the demo with docker `docker run -it -p 9000:9000 linkedin/pinot-quickstart-realtime`

To run the demo with compiled code: `bin/quick-start-realtime.sh`

Once started, the demo will start Kafka, create a Kafka topic, and create a realtime Pinot table. Once created, Pinot will start ingesting events from the Kafka topic into the table. The demo also starts a consumer that consumes events from the Meetup API and pushes them into the Kafka topic that was created, causing new events modified on Meetup to show up in Pinot.

To show new events appearing, one can run `SELECT * FROM meetupRsvp ORDER BY mtime DESC LIMIT 50` repeatedly, which shows the last events that were ingested by Pinot.

CHAPTER 4

Reference

4.1 PQL

- PQL is a derivative of SQL derivative that supports selection, projection, aggregation, grouping aggregation. There is no support for Joins.
- Specifically, for Pinot:
  - Grouping keys always appear in query results, even if not requested
  - Aggregations are computed in parallel
  - Results of aggregations with large amounts of group keys (>1M) are approximated
  - `ORDER BY` only works for selection queries, for aggregations one must use the `TOP` keyword

4.1.1 PQL Examples

The Pinot Query Language (PQL) is very similar to standard SQL:

```
SELECT COUNT(*) FROM myTable
```

4.1.2 Aggregation

```
SELECT COUNT(*), MAX(foo), SUM(bar) FROM myTable
```

4.1.3 Grouping on Aggregation

```
SELECT MIN(foo), MAX(foo), SUM(foo), AVG(foo) FROM myTable
  GROUP BY bar, baz TOP 50
```

4.1.4 Filtering

```
SELECT COUNT(*) FROM myTable
  WHERE foo = 'foo'
  AND bar BETWEEN 1 AND 20
  OR (baz < 42 AND quux IN ('hello', 'goodbye') AND quuux NOT IN (42, 69))
```

4.1.5 Selection (Projection)

```
SELECT * FROM myTable
  WHERE quux < 5
  LIMIT 50
```

4.1.6 Ordering on Selection

```
SELECT foo, bar FROM myTable
  WHERE baz > 20
  ORDER BY bar DESC
  LIMIT 100
```

4.1.7 Pagination on Selection

Note: results might not be consistent if column ordered by has same value in multiple rows.

```
SELECT foo, bar FROM myTable
  WHERE baz > 20
  ORDER BY bar DESC
  LIMIT 50, 100
```

4.1.8 Wild-card match (in WHERE clause only)

To count rows where the column `airlineName` starts with `U`

```
SELECT count(*) FROM SomeTable
  WHERE regexp_like(airlineName, '^U.*')
  GROUP BY airlineName TOP 10
```

4.1.9 Examples with UDF

As of now, functions have to be implemented within Pinot. Injecting functions is not allowed yet. The examples below demonstrate the use of UDFs

```
SELECT count(*) FROM myTable
  GROUP BY timeConvert(timeColumnName, 'SECONDS', 'DAYS')

SELECT count(*) FROM myTable
  GROUP BY div(tim
```

4.1.10 PQL Specification

SELECT

The select statement is as follows

```
SELECT <outputColumn> (, outputColumn, outputColumn,...)
  FROM <tableName>
   (WHERE ... | GROUP BY ... | ORDER BY ... | TOP ... | LIMIT ...)
```

`outputColumn` can be `*` to project all columns, columns (`foo`, `bar`, `baz`) or aggregation functions like (`MIN(foo)`, `MAX(bar)`, `AVG(baz)`).

Supported aggregations on single-value columns

- `COUNT`
- `MIN`
- `MAX`
- `SUM`
- `AVG`
- `MINMAXRANGE`
- `DISTINCTCOUNT`
- `DISTINCTCOUNTHLL`
- `FASTHLL`
- `PERCENTILE[0-100]`: e.g. `PERCENTILE5`, `PERCENTILE50`, `PERCENTILE99`, etc.
- `PERCENTILEEST[0-100]`: e.g. `PERCENTILEEST5`, `PERCENTILEEST50`, `PERCENTILEEST99`, etc.

Supported aggregations on multi-value columns

- `COUNTMV`
- `MINMV`
- `MAXMV`
- `SUMMV`
- `AVGMV`
- `MINMAXRANGEMV`
- `DISTINCTCOUNTMV`
- `DISTINCTCOUNTHLLMV`
- `FASTHLLMV`
- `PERCENTILE[0-100]MV`: e.g. `PERCENTILE5MV`, `PERCENTILE50MV`, `PERCENTILE99MV`, etc.
- `PERCENTILEEST[0-100]MV`: e.g. `PERCENTILEEST5MV`, `PERCENTILEEST50MV`, `PERCENTILEEST99MV`, etc.

WHERE

Supported predicates are comparisons with a constant using the standard SQL operators (=, <, <=, >, >=, <>, '!=') , range comparisons using `BETWEEN` (`foo BETWEEN 42 AND 69`), set membership (`foo IN (1, 2, 4, 8)`) and exclusion (`foo NOT IN (1, 2, 4, 8)`). For `BETWEEN`, the range is inclusive.

Comparison with a regular expression is supported using the regexp_like function, as in `WHERE regexp_like(columnName, 'regular expression')`

GROUP BY

The `GROUP BY` clause groups aggregation results by a list of columns, or transform functions on columns (see below)

ORDER BY

The `ORDER BY` clause orders selection results by a list of columns. PQL supports ordering `DESC` or `ASC`.

TOP

The `TOP n` clause causes the 'n' largest group results to be returned. If not specified, the top 10 groups are returned.

LIMIT

The `LIMIT n` clause causes the selection results to contain at most 'n' results. The `LIMIT a, b` clause paginate the selection results from the 'a' th results and return at most 'b' results.

Transform Function in Aggregation and Grouping

In aggregation and grouping, each column can be transformed from one or multiple columns. For example, the following query will calculate the maximum value of column `foo` divided by column `bar` grouping on the column `time` converted form time unit `MILLISECONDS` to `SECONDS`:

```
SELECT MAX(DIV(foo, bar) FROM myTable
  GROUP BY TIMECONVERT(time, 'MILLISECONDS', 'SECONDS')
```

Supported transform functions

ADD Sum of at least two values

SUB Difference between two values

MULT Product of at least two values

DIV Quotient of two values

TIMECONVERT Takes 3 arguments, converts the value into another time unit. E.g. `TIMECONVERT(time, 'MILLISECONDS', 'SECONDS')`

DATETIMECONVERT Takes 4 arguments, converts the value into another date time format, and buckets time based on the given time granularity. *e.g.* `DATETIMECONVERT(date, '1:MILLISECONDS:EPOCH', '1:SECONDS:EPOCH', '15:MINUTES')`

VALUEIN Takes at least 2 arguments, where the first argument is a multi-valued column, and the following arguments are constant values. The transform function will filter the value from the multi-valued column with the given constant values. The VALUEIN transform function is especially useful when the same multi-valued column is both filtering column and grouping column. *e.g.* VALUEIN(mvColumn, 3, 5, 15)

4.1.11 Differences with SQL

- JOIN is not supported
- Use TOP instead of LIMIT for truncation
- LIMIT n has no effect in grouping queries, should use TOP n instead. If no TOP n defined, PQL will use TOP 10 as default truncation setting.
- No need to select the columns to group with.

The following two queries are both supported in PQL, where the non-aggregation columns are ignored.

```
SELECT MIN(foo), MAX(foo), SUM(foo), AVG(foo) FROM mytable
  GROUP BY bar, baz
  TOP 50

SELECT bar, baz, MIN(foo), MAX(foo), SUM(foo), AVG(foo) FROM mytable
  GROUP BY bar, baz
  TOP 50
```

- The results will always order by the aggregated value (descending).

The results for query:

```
SELECT MIN(foo), MAX(foo) FROM myTable
  GROUP BY bar
  TOP 50
```

will be the same as the combining results from the following queries:

```
SELECT MIN(foo) FROM myTable
  GROUP BY bar
  TOP 50
SELECT MAX(foo) FROM myTable
  GROUP BY bar
  TOP 50
```

where we don't put the results for the same group together.

4.2 Executing queries via REST API on the Broker

The Pinot REST API can be accessed by invoking POST operation witha a JSON body containing the parameter pql to the /query URI endpoint on a broker. Depending on the type of query, the results can take different shapes. The examples below use curl.

4.2.1 Aggregation

```
curl -X POST -d '{"pql":"select count(*) from flights"}' http://localhost:8099/query


{
 "traceInfo":{},
 "numDocsScanned":17,
 "aggregationResults":[
    {
       "function":"count_star",
       "value":"17"
    }
 ],
 "timeUsedMs":27,
 "segmentStatistics":[],
 "exceptions":[],
 "totalDocs":17
}
```

4.2.2 Aggregation with grouping

```
curl -X POST -d '{"pql":"select count(*) from flights group by Carrier"}' http://
↪localhost:8099/query


{
 "traceInfo":{},
 "numDocsScanned":23,
 "aggregationResults":[
    {
       "groupByResult":[
          {
             "value":"10",
             "group":["AA"]
          },
          {
             "value":"9",
             "group":["VX"]
          },
          {
             "value":"4",
             "group":["WN"]
          }
       ],
       "function":"count_star",
       "groupByColumns":["Carrier"]
    }
 ],
 "timeUsedMs":47,
 "segmentStatistics":[],
 "exceptions":[],
 "totalDocs":23
}
```

4.2.3 Selection

```
curl -X POST -d '{"pql":"select * from flights limit 3"}' http://localhost:8099/query


{
 "selectionResults":{
    "columns":[
       "Cancelled",
       "Carrier",
       "DaysSinceEpoch",
       "Delayed",
       "Dest",
       "DivAirports",
       "Diverted",
       "Month",
       "Origin",
       "Year"
    ],
    "results":[
       [
          "0",
          "AA",
          "16130",
          "0",
          "SFO",
          [],
          "0",
          "3",
          "LAX",
          "2014"
       ],
       [
          "0",
          "AA",
          "16130",
          "0",
          "LAX",
          [],
          "0",
          "3",
          "SFO",
          "2014"
       ],
       [
          "0",
          "AA",
          "16130",
          "0",
          "SFO",
          [],
          "0",
          "3",
          "LAX",
          "2014"
       ]
    ]
```

(continues on next page)

(continued from previous page)

```
 },
 "traceInfo":{},
 "numDocsScanned":3,
 "aggregationResults":[],
 "timeUsedMs":10,
 "segmentStatistics":[],
 "exceptions":[],
 "totalDocs":102
}
```

4.2.4 Java

The Pinot client API is similar to JDBC, although there are some differences, due to how Pinot behaves. For example, a query with multiple aggregation function will return one result set per aggregation function, as they are computed in parallel.

Connections to Pinot are created using the ConnectionFactory class' utility methods to create connections to a Pinot cluster given a Zookeeper URL, a Java Properties object or a list of broker addresses to connect to.

```
Connection connection = ConnectionFactory.fromZookeeper
  ("some-zookeeper-server:2191/zookeeperPath");

Connection connection = ConnectionFactory.fromProperties("demo.properties");

Connection connection = ConnectionFactory.fromHostList
  ("some-server:1234", "some-other-server:1234", ...);
```

Queries can be sent directly to the Pinot cluster using the Connection.execute(java.lang.String) and Connection.executeAsync(java.lang.String) methods of Connection.

```
ResultSetGroup resultSetGroup = connection.execute("select * from foo...");
Future<ResultSetGroup> futureResultSetGroup = connection.executeAsync
  ("select * from foo...");
```

Queries can also use a PreparedStatement to escape query parameters:

```
PreparedStatement statement = connection.prepareStatement
  ("select * from foo where a = ?");
statement.setString(1, "bar");

ResultSetGroup resultSetGroup = statement.execute();
Future<ResultSetGroup> futureResultSetGroup = statement.executeAsync();
```

In the case of a selection query, results can be obtained with the various get methods in the first ResultSet, obtained through the getResultSet(int) method:

```
ResultSet resultSet = connection.execute
  ("select foo, bar from baz where quux = 'quuux'").getResultSet(0);

for (int i = 0; i < resultSet.getRowCount(); ++i) {
  System.out.println("foo: " + resultSet.getString(i, 0));
  System.out.println("bar: " + resultSet.getInt(i, 1));
}

resultSet.close();
```

In the case of aggregation, each aggregation function is within its own ResultSet:

```
ResultSetGroup resultSetGroup = connection.execute("select count(*) from foo");

ResultSet resultSet = resultSetGroup.getResultSet(0);
System.out.println("Number of records: " + resultSet.getInt(0));
resultSet.close();
```

There can be more than one ResultSet, each of which can contain multiple results grouped by a group key.

```
ResultSetGroup resultSetGroup = connection.execute
    ("select min(foo), max(foo) from bar group by baz");

System.out.println("Number of result groups:" +
    resultSetGroup.getResultSetCount(); // 2, min(foo) and max(foo)

ResultSet minResultSet = resultSetGroup.getResultSet(0);
for(int i = 0; i < minResultSet.length(); ++i) {
    System.out.println("Minimum foo for " + minResultSet.getGroupKeyString(i, 1) +
        ": " + minResultSet.getInt(i));
}

ResultSet maxResultSet = resultSetGroup.getResultSet(1);
for(int i = 0; i < maxResultSet.length(); ++i) {
    System.out.println("Maximum foo for " + maxResultSet.getGroupKeyString(i, 1) +
        ": " + maxResultSet.getInt(i));
}

resultSet.close();
```

4.3 Managing Pinot via REST API on the Controller

TODO : Remove this section altogether and find a place somewhere for a pointer to the management API. Maybe in the 'Running pinot in production' section?

There is a REST API which allows management of tables, tenants, segments and schemas. It can be accessed by going to `http://[controller host]/help` which offers a web UI to do these tasks, as well as document the REST API.

It can be used instead of the `pinot-admin.sh` commands to automate the creation of tables and tenants.

4.4 Creating Pinot segments

Pinot segments can be created offline on Hadoop, or via command line from data files. Controller REST endpoint can then be used to add the segment to the table to which the segment belongs.

4.4.1 Creating segments using hadoop

To create Pinot segments on Hadoop, a workflow can be created to complete the following steps:

1. Pre-aggregate, clean up and prepare the data, writing it as Avro format files in a single HDFS directory
2. Create segments

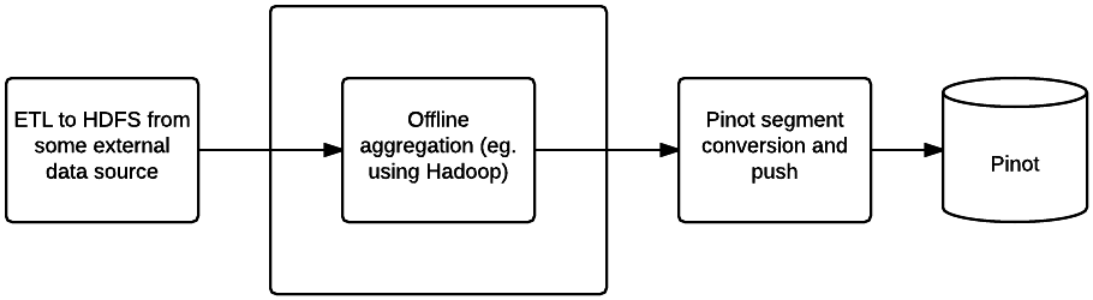


Fig. 1: Offline Pinot workflow

3. Upload segments to the Pinot cluster

Step one can be done using your favorite tool (such as Pig, Hive or Spark), Pinot provides two MapReduce jobs to do step two and three.

Configuring the job

Create a job properties configuration file, such as one below:

```
# === Index segment creation job config ===

# path.to.input: Input directory containing Avro files
path.to.input=/user/pinot/input/data

# path.to.output: Output directory containing Pinot segments
path.to.output=/user/pinot/output

# path.to.schema: Schema file for the table, stored locally
path.to.schema=flights-schema.json

# segment.table.name: Name of the table for which to generate segments
segment.table.name=flights

# === Segment tar push job config ===

# push.to.hosts: Comma separated list of controllers host names to which to push
push.to.hosts=controller_host_0,controller_host_1

# push.to.port: The port on which the controller runs
push.to.port=8888
```

Executing the job

The Pinot Hadoop module contains a job that you can incorporate into your workflow to generate Pinot segments.

```
mvn clean install -DskipTests -Pbuild-shaded-jar
hadoop jar pinot-hadoop-0.016-shaded.jar SegmentCreation job.properties
```

You can then use the SegmentTarPush job to push segments via the controller REST API.

```
hadoop jar pinot-hadoop-0.016-shaded.jar SegmentTarPush job.properties
```

4.4.2 Creating Pinot segments outside of Hadoop

Here is how you can create Pinot segments from standard formats like CSV/JSON.

1. Follow the steps described in the section on *Compiling the code* to build pinot. Locate `pinot-admin.sh` in `pinot-tools/target/pinot-tools-pkg/bin/pinot-admin.sh`.
2. Create a top level directory containing all the CSV/JSON files that need to be converted into segments.
3. The file name extensions are expected to be the same as the format name (*i.e* `.csv`, or `.json`), and are case insensitive. Note that the converter expects the `.csv` extension even if the data is delimited using tabs or spaces instead.
4. Prepare a schema file describing the schema of the input data. The schema needs to be in JSON format. See example later in this section.
5. **Specifically for CSV format, an optional csv config file can be provided (also in JSON format). This is used to configure pa**
   A detailed description of this follows below.

Run the pinot-admin command to generate the segments. The command can be invoked as follows. Options within "[ ]" are optional. For -format, the default value is AVRO.

```
bin/pinot-admin.sh CreateSegment -dataDir <input_data_dir> [-format [CSV/JSON/AVRO]]␣
↪[-readerConfigFile <csv_config_file>] [-generatorConfigFile <generator_config_file>
↪] -segmentName <segment_name> -schemaFile <input_schema_file> -tableName <table_
↪name> -outDir <output_data_dir> [-overwrite]
```

To configure various parameters for CSV a config file in JSON format can be provided. This file is optional, as are each of its parameters. When not provided, default values used for these parameters are described below:

1. fileFormat: Specify one of the following. Default is EXCEL.

   ##. EXCEL ##. MYSQL ##. RFC4180 ##. TDF

1. header: If the input CSV file does not contain a header, it can be specified using this field. Note, if this is specified, then the input file is expected to not contain the header row, or else it will result in parse error. The columns in the header must be delimited by the same delimiter character as the rest of the CSV file.
2. delimiter: Use this to specify a delimiter character. The default value is ",".
3. dateFormat: If there are columns that are in date format and need to be converted into Epoch (in milliseconds), use this to specify the format. Default is "mm-dd-yyyy".
4. dateColumns: If there are multiple date columns, use this to list those columns.

Below is a sample config file.

```
{
  "fileFormat" : "EXCEL",
  "header" : "col1,col2,col3,col4",
  "delimiter" : "\t",
  "dateFormat" : "mm-dd-yy"
  "dateColumns" : ["col1", "col2"]
}
```

Sample Schema:

```
{
  "dimensionFieldSpecs" : [
    {
      "dataType" : "STRING",
      "delimiter" : null,
      "singleValueField" : true,
      "name" : "name"
    },
    {
      "dataType" : "INT",
      "delimiter" : null,
      "singleValueField" : true,
      "name" : "age"
    }
  ],
  "timeFieldSpec" : {
    "incomingGranularitySpec" : {
      "timeType" : "DAYS",
      "dataType" : "LONG",
      "name" : "incomingName1"
    },
    "outgoingGranularitySpec" : {
      "timeType" : "DAYS",
      "dataType" : "LONG",
      "name" : "outgoingName1"
    }
  },
  "metricFieldSpecs" : [
    {
      "dataType" : "FLOAT",
      "delimiter" : null,
      "singleValueField" : true,
      "name" : "percent"
    }
   ]
  },
  "schemaName" : "mySchema",
}
```

Pushing segments to Pinot

You can use curl to push a segment to pinot:

```
curl -X POST -F segment=@<segment-tar-file-path> http://controllerHost:controllerPort/
↪segments
```

Alternatively you can use the pinot-admin.sh utility to upload one or more segments:

```
pinot-tools/target/pinot-tools-pkg/bin//pinot-admin.sh UploadSegment -controllerHost
↪<hostname> -controllerPort <port> -segmentDir <segmentDirectoryPath>
```

The command uploads all the segments found in `segmentDirectoryPath`. The segments could be either tar-compressed (in which case it is a file under `segmentDirectoryPath`) or uncompressed (in which case it is a directory under `segmentDirectoryPath`).

4.5 Running Pinot in production

4.5.1 Installing Pinot

Requirements

- Java 8+
- Several nodes with enough memory
- A working installation of Zookeeper

Recommended environment

- Shared storage infrastructure (such as NFS)
- Regular Zookeeper backups
- HTTP load balancers (such as nginx/haproxy)

4.5.2 Deploying Pinot

Direct deployment of Pinot

Deployment of Pinot on Kubernetes

4.5.3 Managing Pinot

Creating tables

Updating tables

Uploading data

Configuring realtime data ingestion

Monitoring Pinot

CHAPTER 5

Customizing Pinot

5.1 Pluggable Streams

Prior to commit ba9f2d, Pinot was only able to support reading from Kafka stream.

Pinot now enables its users to write plug-ins to read from pub-sub streams other than Kafka. (Please refer to Issue #2583)

Some of the streams for which plug-ins can be added are:

- Amazon kinesis
- Azure Event Hubs
- LogDevice
- Pravega
- Pulsar

You may encounter some limitations either in Pinot or in the stream system while developing plug-ins. Please feel free to get in touch with us when you start writing a stream plug-in, and we can help you out. We are open to receiving PRs in order to improve these abstractions if they do not work for a certain stream implementation.

Refer to sections *High Level Consumers* and *Low Level Consumers* for details on how Pinot consumes streaming data.

5.1.1 Requirements to support Stream Level (High Level) consumers

The stream should provide the following guarantees:

- Exactly once delivery (unless restarting from a checkpoint) for each consumer of the stream.
- (Optionally) support mechanism to split events (in some arbitrary fashion) so that each event in the stream is delivered exactly to one host out of set of hosts.
- Provide ways to save a checkpoint for the data consumed so far. If the stream is partitioned, then this checkpoint is a vector of checkpoints for events consumed from individual partitions.

- The checkpoints should be recorded only when Pinot makes a call to do so.
- The consumer should be able to start consumption from one of:
  - latest avaialble data
  - earliest available data
  - last saved checkpoint

5.1.2 Requirements to support Partition Level (Low Level) consumers

While consuming rows at a partition level, the stream should support the following properties:

- Stream should provide a mechanism to get the current number of partitions.
- Each event in a partition should have a unique offset that is not more than 64 bits long.
- Refer to a partition as a number not exceeding 32 bits long.
- Stream should provide the following mechanisms to get an offset for a given partition of the stream:
  - get the offset of the oldest event available (assuming events are aged out periodically) in the partition.
  - get the offset of the most recent event published in the partition
  - (optionally) get the offset of an event that was published at a specified time
- Stream should provide a mechanism to consume a set of events from a partition starting from a specified offset.
- Events with higher offsets should be more recent (the offsets of events need not be contiguous)

In addition, we have an operational requirement that the number of partitions should not be reduced over time.

5.1.3 Stream plug-in implementation

In order to add a new type of stream (say,Foo) implement the following classes:

1. FooConsumerFactory extends StreamConsumerFactory
2. FooPartitionLevelConsumer implements PartitionLevelConsumer
3. FooStreamLevelConsumer implements StreamLevelConsumer
4. FooMetadataProvider implements StreamMetadataProvider
5. FooMessageDecoder implements StreamMessageDecoder

Depending on stream level or partition level, your implementation needs to include StreamLevelConsumer or PartitionLevelConsumer.

The properties for the stream implementation are to be set in the table configuration, inside streamConfigs section.

Use the `streamType` property to define the stream type. For example, for the implementation of stream `foo`, set the property `"streamType" : "foo"`.

The rest of the configuration properties for your stream should be set with the prefix `"stream.foo"`. Be sure to use the same suffix for: (see examples below):

- topic
- consumer type
- stream consumer factory
- offset

- decoder class name
- decoder properties
- connnection timeout
- fetch timeout

All values should be strings. For example:

```
"streamType" : "foo",
"stream.foo.topic.name" : "SomeTopic",
"stream.foo.consumer.type": "lowlevel",
"stream.foo.consumer.factory.class.name": "fully.qualified.pkg.
↪ConsumerFactoryClassName",
"stream.foo.consumer.prop.auto.offset.reset": "largest",
"stream.foo.decoder.class.name" : "fully.qualified.pkg.DecoderClassName",
"stream.foo.decoder.prop.a.decoder.property" : "decoderPropValue",
"stream.foo.connection.timeout.millis" : "10000", // default 30_000
"stream.foo.fetch.timeout.millis" : "10000" // default 5_000
```

You can have additional properties that are specific to your stream. For example:

```
"stream.foo.some.buffer.size" : "24g"
```

In addition to these properties, you can define thresholds for the consuming segments:

- rows threshold
- time threshold

The properties for the thresholds are as follows:

```
"realtime.segment.flush.threshold.size" : "100000"
"realtime.segment.flush.threshold.time" : "6h"
```

An example of this implementation can be found in the KafkaConsumerFactory, which is an implementation for the kafka stream.

5.2 Segment Fetchers

When pinot segment files are created in external systems (hadoop/spark/etc), there are several ways to push those data to pinot Controller and Server:

1. push segment to shared NFS and let pinot pull segment files from the location of that NFS.
2. push segment to a Web server and let pinot pull segment files from the Web server with http/https link.
3. push segment to HDFS and let pinot pull segment files from HDFS with hdfs location uri.
4. push segment to other system and implement your own segment fetcher to pull data from those systems.

The first two options should be supported out of the box with pinot package. As long your remote jobs send Pinot controller with the corresponding URI to the files it will pick up the file and allocate it to proper Pinot Servers and brokers. To enable Pinot support for HDFS, you will need to provide Pinot Hadoop configuration and proper Hadoop dependencies.

5.2.1 HDFS segment fetcher configs

In your Pinot controller/server configuration, you will need to provide the following configs:

```
pinot.controller.segment.fetcher.hdfs.hadoop.conf.path=`<file path to hadoop conf␣
↪folder>
```

or

```
pinot.server.segment.fetcher.hdfs.hadoop.conf.path=`<file path to hadoop conf folder>
```

This path should point the local folder containing `core-site.xml` and `hdfs-site.xml` files from your Hadoop installation

```
pinot.controller.segment.fetcher.hdfs.hadoop.kerberos.principle=`<your kerberos␣
↪principal>
pinot.controller.segment.fetcher.hdfs.hadoop.kerberos.keytab=`<your kerberos keytab>
```

or

```
pinot.server.segment.fetcher.hdfs.hadoop.kerberos.principle=`<your kerberos principal>
pinot.server.segment.fetcher.hdfs.hadoop.kerberos.keytab=`<your kerberos keytab>
```

These two configs should be the corresponding Kerberos configuration if your Hadoop installation is secured with Kerberos. Please check Hadoop Kerberos guide on how to generate Kerberos security identification.

You will also need to provide proper Hadoop dependencies jars from your Hadoop installation to your Pinot startup scripts.

5.2.2 Push HDFS segment to Pinot Controller

To push HDFS segment files to Pinot controller, you just need to ensure you have proper Hadoop configuration as we mentioned in the previous part. Then your remote segment creation/push job can send the HDFS path of your newly created segment files to the Pinot Controller and let it download the files.

For example, the following curl requests to Controller will notify it to download segment files to the proper table:

```
curl -X POST -H "UPLOAD_TYPE:URI" -H "DOWNLOAD_URI:hdfs://nameservice1/hadoop/path/to/
↪segment/file.gz" -H "content-type:application/json" -d '' localhost:9000/segments
```

5.2.3 Implement your own segment fetcher for other systems

You can also implement your own segment fetchers for other file systems and load into Pinot system with an external jar. All you need to do is to implement a class that extends the interface of SegmentFetcher and provides config to Pinot Controller and Server as follows:

```
pinot.controller.segment.fetcher.`<protocol>`.class =`<class path to your␣
↪implementation>
```

or

```
pinot.server.segment.fetcher.`<protocol>`.class =`<class path to your implementation>
```

You can also provide other configs to your fetcher under config-root `pinot.server.segment.fetcher.<protocol>`

5.3 Star-Tree: A Specialized Index for Fast Aggregations

One of the biggest challenges in realtime OLAP systems is achieving and maintaining tight SLA's on latency and throughput on large data sets.

Existing techniques such as sorted index or inverted index help improve query latencies, but speed-ups are still limited by number of documents necessary to process for computing the results. On the other hand, pre-aggregating the results ensures a constant upper bound on query latencies, but can lead to storage space explosion.

Here we introduce **Star-Tree** index to utilize the pre-aggregated documents in a smart way to achieve low query latencies but also use the storage space efficiently for aggregation/group-by queries.

5.3.1 Existing Solutions

Consider the following data set as an example to discuss the existing approaches:

Country	Browser	Locale	Impressions
CA	Chrome	en	400
CA	Firefox	fr	200
MX	Safari	es	300
MX	Safari	en	100
USA	Chrome	en	600
USA	Firefox	es	200
USA	Firefox	en	400

Sorted Index

In this approach, data is sorted on a primary key, which is likely to appear as filter in most queries in the query set.

This reduces the time to search the documents for a given primary key value from linear scan *O(n)* to binary search *O(logn)*, and also keeps good locality for the documents selected.

While this is a good improvement over linear scan, there are still a few issues with this approach:

- While sorting on one column does not require additional space, sorting on additional columns would require additional storage space to re-index the records for the various sort orders.
- While search time is reduced from *O(n)* to *O(logn)*, overall latency is still a function of total number of documents need to be processed to answer a query.

Inverted Index

In this approach, for each value of a given column, we maintain a list of document id's where this value appears.

Below are the inverted indexes for columns 'Browser' and 'Locale' for our example data set:

Browser	Doc Id
Firefox	1,5,6
Chrome	0,4
Safari	2,3

Locale	Doc Id
en	0,3,4,6
es	2,5
fr	1

For example, if we want to get all the documents where 'Browser' is 'Firefox', we can simply look up the inverted index for 'Browser' and identify that it appears in documents [1, 5, 6].

Using inverted index, we can reduce the search time to constant time *O(1)*. However, the query latency is still a function of the selectivity of the query, i.e. increases with the number of documents need to be processed to answer the query.

Pre-aggregation

In this technique, we pre-compute the answer for a given query set upfront.

In the example below, we have pre-aggregated the total impressions for each country:

Country	Impressions
CA	600
MX	400
USA	1200

Doing so makes answering queries about total impressions for a country just a value lookup, by eliminating the need of processing a large number of documents. However, to be able to answer with multiple predicates implies pre-aggregating for various combinations of different dimensions. This leads to exponential explosion in storage space.

5.3.2 Star-Tree Solution

On one end of the spectrum we have indexing techniques that improve search times with limited increase in space, but do not guarantee a hard upper bound on query latencies. On the other end of the spectrum we have pre-aggregation techniques that offer hard upper bound on query latencies, but suffer from exponential explosion of storage space.

We propose the Star-Tree data structure that offers a configurable trade-off between space and time and allows us to achieve hard upper bound for query latencies for a given use case. In the following sections we will define the Star-Tree data structure, and discuss how it is utilized within Pinot for achieving low latencies with high throughput.

Definition

Tree Structure

Star-Tree is a tree data structure that is consisted of the following properties:

- **Root Node** (Orange): Single root node, from which the rest of the tree can be traversed.
- **Leaf Node** (Blue): A leaf node can containing at most T records, where T is configurable.
- **Non-leaf Node** (Green): Nodes with more than T records are further split into children nodes.
- **Star-Node** (Yellow): Non-leaf nodes can also have a special child node called the Star-Node. This node contains the pre-aggregated records after removing the dimension on which the data was split for this level.

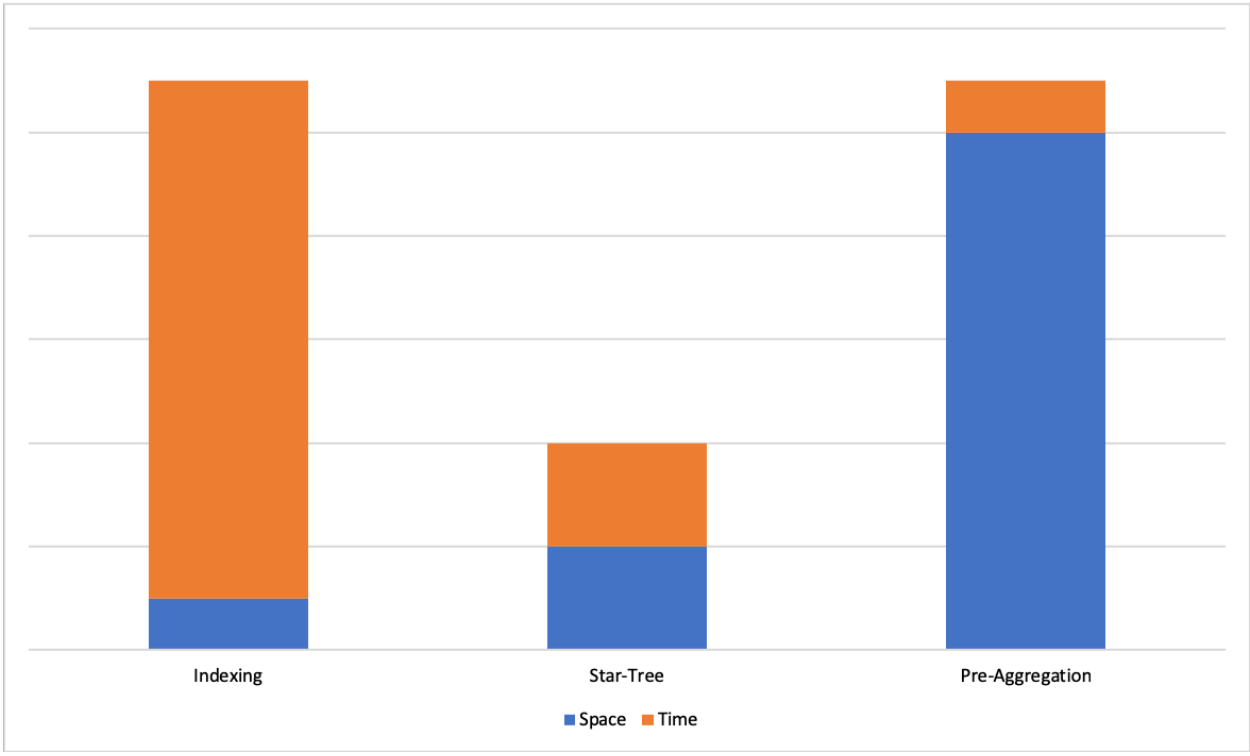


Fig. 1: Space-Time Trade Off Between Different Techniques

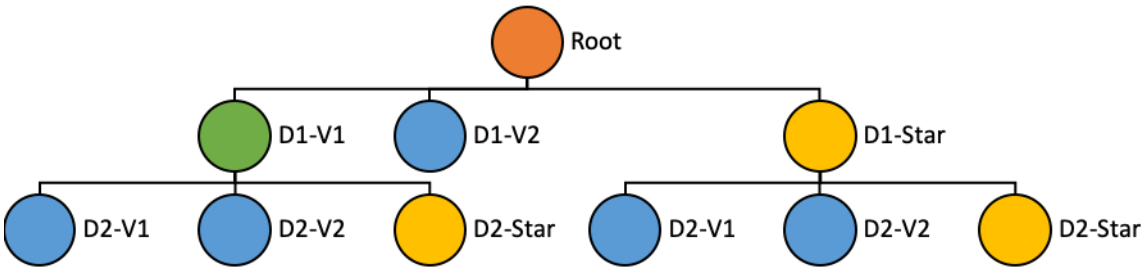


Fig. 2: Star-Tree Structure

- **Dimensions Split Order** ([D1, D2]): Nodes at a given level in the tree are split into children nodes on all values of a particular dimension. The dimensions split order is an ordered list of dimensions that is used to determine the dimension to split on for a given level in the tree.

Node Properties

The properties stored in each node are as follows:

- **Dimension**: The dimension which the node is split on
- **Start/End Document Id**: The range of documents this node points to
- **Aggregated Document Id**: One single document which is the aggregation result of all documents pointed by this node

Index Generation

Star-Tree index is generated in the following steps:

- The data is first projected as per the *dimensionsSplitOrder*. Only the dimensions from the split order are reserved, others are dropped. For each unique combination of reserved dimensions, metrics are aggregated per configuration. The aggregated documents are written to a file and served as the initial Star-Tree documents (separate from the original documents).
- Sort the Star-Tree documents based on the *dimensionsSplitOrder*. It is primary-sorted on the first dimension in this list, and then secondary sorted on the rest of the dimensions based on their order in the list. Each node in the tree points to a range in the sorted documents.
- The tree structure can be created recursively (starting at root node) as follows:
  - If a node has more than T records, it is split into multiple children nodes, one for each value of the dimension in the split order corresponding to current level in the tree.
  - A Star-Node can be created (per configuration) for the current node, by dropping the dimension being split on, and aggregating the metrics for rows containing dimensions with identical values. These aggregated documents are appended to the end of the Star-Tree documents.

    If there is only one value for the current dimension, Star-Node won't be created because the documents under the Star-Node are identical to the single node.
- The above step is repeated recursively until there are no more nodes to split.
- Multiple Star-Trees can be generated based on different configurations (*dimensionsSplitOrder*, *aggregations*, T)

Aggregation

Aggregation is configured as a pair of aggregation function and the column to apply the aggregation.

All types of aggregation function with bounded-sized intermediate result are supported.

Supported Functions

- COUNT
- MIN
- MAX

- SUM
- AVG
- MINMAXRANGE
- DISTINCTCOUNTHLL
- PERCENTILEEST
- PERCENTILETDIGEST

Unsupported Functions

- DISTINCTCOUNT: Intermediate result *Set* is unbounded
- PERCENTILE: Intermediate result *List* is unbounded

Index Generation Configuration

Multiple index generation configurations can be provided to generate multiple Star-Trees. Each configuration should contain the following properties:

- **dimensionsSplitOrder**: An ordered list of dimension names can be specified to configure the split order. Only the dimensions in this list are reserved in the aggregated documents. The nodes will be split based on the order of this list. For example, split at level *i* is performed on the values of dimension at index *i* in the list.
- **skipStarNodeCreationForDimensions** (Optional, default empty): A list of dimension names for which to not create the Star-Node.
- **functionColumnPairs**: A list of aggregation function and column pairs (split by double underscore "__"). E.g. **SUM__Impressions** (*SUM* of column *Impressions*)
- **maxLeafRecords** (Optional, default 10000): The threshold *T* to determine whether to further split each node.

Example

For our example data set, with the following example configuration, the tree and documents should be something like below.

StarTreeIndexConfig

```
{
  "dimensionsSplitOrder": [
    "Country",
    "Browser",
    "Locale"
  ],
  "skipStarNodeCreationForDimensions": [],
  "functionColumnPairs": [
    "SUM__Impressions"
  ],
  "maxLeafRecords": 1
}
```

Tree Structure

The values in the parentheses are the aggregated sum of *Impressions* for all the documents under the node.

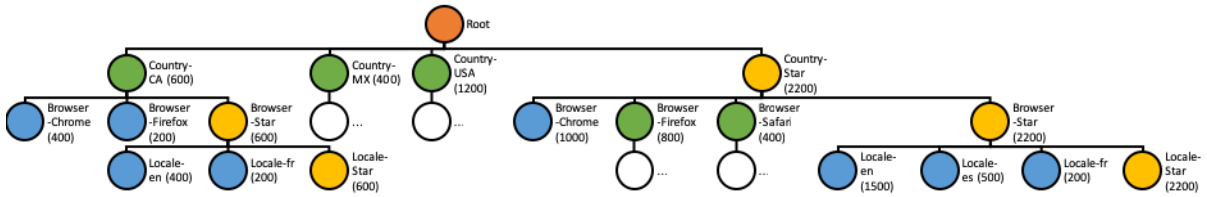


Fig. 3: Star-Tree Example

Star-Tree documents

Country	Browser	Locale	SUM__Impressions
CA	Chrome	en	400
CA	Firefox	fr	200
MX	Safari	en	100
MX	Safari	es	300
USA	Chrome	en	600
USA	Firefox	en	400
USA	Firefox	es	200
CA	*	en	400
CA	*	fr	200
CA	*	*	600
MX	Safari	*	400
USA	Firefox	*	600
USA	*	en	1000
USA	*	es	200
USA	*	*	1200
*	Chrome	en	1000
*	Firefox	en	400
*	Firefox	es	200
*	Firefox	fr	200
*	Firefox	*	800
*	Safari	en	100
*	Safari	es	300
*	Safari	*	400
*	*	en	1500
*	*	es	500
*	*	fr	200
*	*	*	2200

Query Execution

For query execution, the idea is to first check metadata to determine whether the query can be solved with the Star-Tree documents, then traverse the Star-Tree to identify documents that satisfy all the predicates. After applying any remain-

ing predicates that were missed while traversing the Star-Tree to the identified documents, apply aggregation/group-by on the qualified documents.

The algorithm to traverse the tree can be described as follows:

- Start from root node.
- For each level, what child node(s) to select depends on whether there are any predicates/group-by on the split dimension for the level in the query.
  - If there is no predicate or group-by on the split dimension, select the Star-Node if exists, or all child nodes to traverse further.
  - If there are predicate(s) on the split dimension, select the child node(s) that satisfy the predicate(s).
  - If there is no predicate, but there is a group-by on the split dimension, select all child nodes except Star-Node.
- Recursively repeat the previous step until all leaf nodes are reached, or all predicates are satisfied.
- Collect all the documents pointed to by the selected nodes.
  - If all predicates and group-bys are satisfied, pick the single aggregated document from each selected node.
  - Otherwise, collect all the documents in the document range from each selected node.

CHAPTER 6

Design Documents

6.1 Realtime Design

Pinot consumes rows from streaming data (such as Kafka) and serves queries on the data consumed thus far.

Two modes of consumption are supported in Pinot:

6.1.1 High Level Consumers

TODO: Add design description of how HLC realtime works

6.1.2 Low Level Consumers

The HighLevel Pinot consumer has the following properties:

- **Each consumer needs to consume from all partitions. So, if we run one consumer in a host, we are limited by the capacity**
  A stream may provide a way by which multiple hosts can consume the same topic, with each host receiving a subset of the messages. However in this mode the stream may duplicate rows that across the machines when the machines go down and come back up. Pinot cannot afford that.
- **A stream consumer has no control over the messages that are received. As a result, the consumers may have more or less s**
  If we have common realtime segments across servers, it allows the brokers to use different routing algorithms (like we do with offline segments). Otherwise, the broker needs to route a query to exactly one realtime server so that we do not see duplicate data in results.

6.1.3 Design

When a table is created, the controller determines the number of partitions for the table, and "creates" one segment per partition, spraying these segments evenly across all the tagged servers. The following steps are done as a part of creating each segment: * Segment metadata is created in Zookeeper. The segments are named as tableName__partitionNumber__segmentSeqNumber__Timestamp. For example: "myTable__6__0__20180801T1647Z"

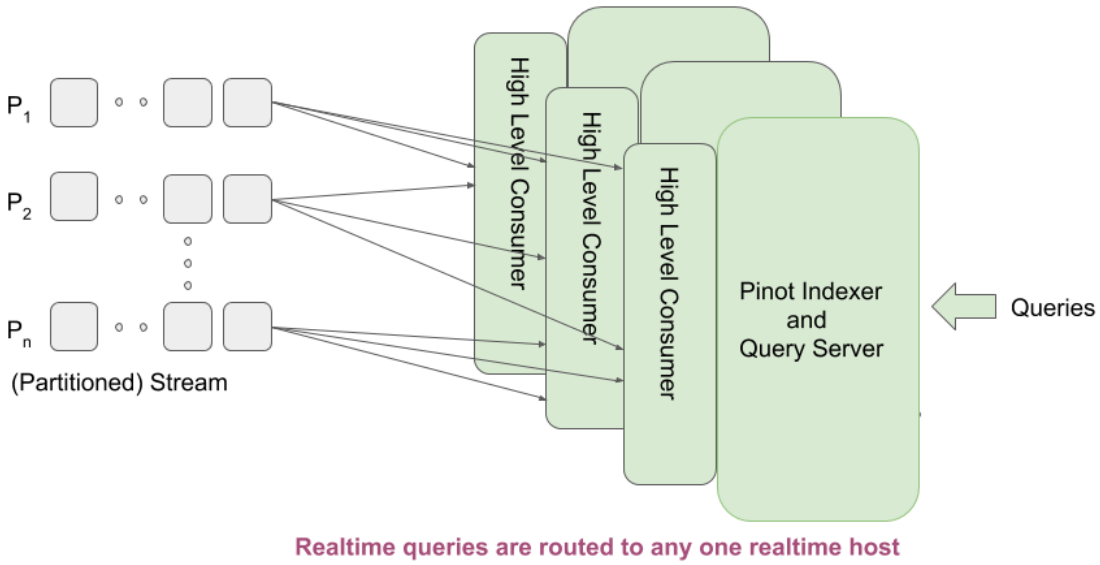


Fig. 1: High Level Stream Consumer Architecture

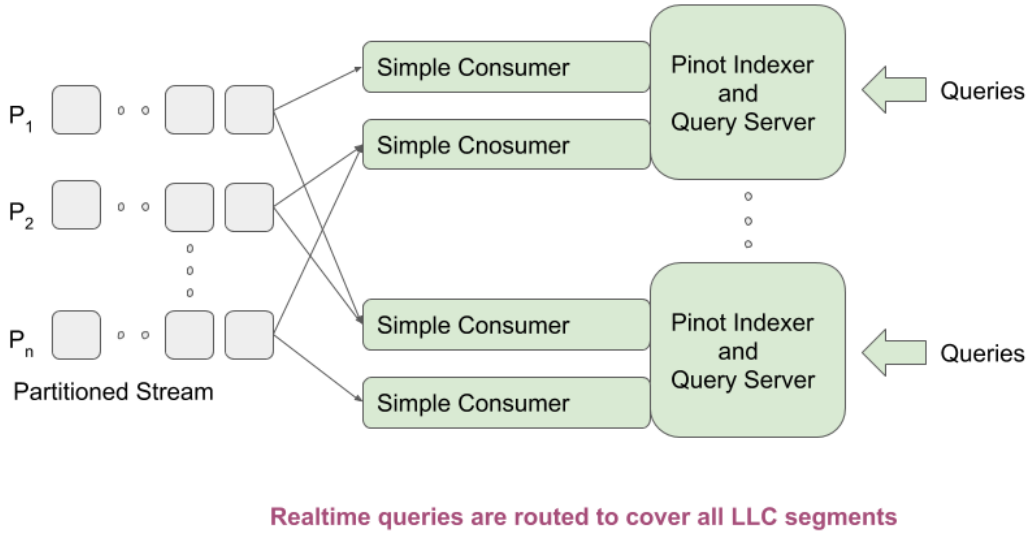


Fig. 2: Low Level Stream Consumer Architecture

* Segment metadata is set with the segment completion criteria – the number of rows. The controller computes this number by dividing the rows threshold set in table configuration by the total number of segments of the table on the same server. * Segment metadata is set with the offset from which to consume. Controller determines the offset by querying the stream. * Table Idealstate is set with these segment names and the appropriate server instances thay are hosted. The state is set to CONSUMING * Depending on the number of replicas set, each partition could be consumed in multiple servers.

When a server completes consuming the segment and reaches the end-criteria (either time or number of rows as per segment metadata), the server goes through a segment completion protocol sequence (described in diagrams below). The controller orchestrates all the replicas to reach the same consumption level, and allows one replica to "commit" a segment. The "commit" step involves:

- The server uploads the completed segment to the controller
- The controller updates that segments metadata to mark it completed, writes the end offset, end time, etc. in the metadata
- The controller creates a new segment for the same partition (e.g. "myTable__6__1__20180801T1805Z") and sets the metadata exactly like before, with the consumption offsets adjusted to reflect the end offset of the previous segmentSeqNumber.
- The controller updates the IdealState to change the state of the completing segment to ONLINE, and add the new segment in CONSUMING state.

As a first implementation, the end-criteria in the metadata points to the table config. It can be used at some point if we want to implement a more fancy end-criteria, perhaps based on traffic or other conditions, something that varies on a per-segment basis. The end-criteria could be number of rows, or time. If number of rows is specified, then the controller divides the specified number by the number of segments (of that table) on the same server, and sets the appropriate row threshold in the segment metadata. The consuming server simply obeys what is in segment metadata for row threshold.

We change the broker to introduce a new routing strategy that prefers ONLINE to CONSUMING segments, and ensures that there is at most one segment in CONSUMING state on a per partition basis in the segments that a query is to be routed to.

6.1.4 Important tuning parameters for Realtime Pinot

- replicasPerPartition: This number indicates how many replicas are needed for each partition to be consumed from the stream
- realtime.segment.flush.threshold.size: This parameter should be set to the total number of rows of a topic that a realtime consuming server can hold in memory. Default value is 5M. If the value is set to 0, then the number of rows is automatically adjusted such that the size of the segment generated is as per the setting realtime.segment.flush.desired.size
- realtime.segment.flush.desired.size: Default value is "200M". The setting is used only if realtime.segment.flush.threshold.size is set to 0
- realtime.segment.flush.threshold.size.llc: This parameter overrides realtime.segment.flush.threshold.size. Useful when migrating live from HLC to LLC
- pinot.server.instance.realtime.alloc.offheap: Default is false. Set it to true if you want off-heap allocation for dictionaries and no-dictionary column
- pinot.server.instance.realtime.alloc.offheap.direct: Default is false. Set it to true if you want off-heap allocation from DirectMemory (as opposed to MMAP)
- pinot.server.instance.realtime.max.parallel.segment.builds: Default is 0 (meaning infinite). Set it to a number if you want to limit number of segment builds. Segment builds take up heap memory, so it is useful to have a max setting and limit the number of simultaneous segment builds on a single server instance JVM.

6.1.5 Live migration of existing use cases from HLC to LLC

Preparation

- Set the new configurations as desired (replicasPerPartition, realtime.segment.flush.threshold.size.llc, realtime.segment.flush.threshold.time.llc). Note that the ".llc" versions of the configs are to be used only if you want to do a live migration of an existing table from HLC to LLC and need to have different thresholds for LLC than HLC.
- Set loadMode of segments to MMAP
- Set configurations to use offheap (either direct or MMAP) for dictionaries and no-dictinary items (realtime.alloc.offheap, realtime.alloc.offheap.direct)
- If your stream is Kafka, add `stream.kafka.broker.list` configurations for per-partition consumers
- Increase the heap size (doubling it may be useful) since we will be consuming both HLC and LLC on the same machines now. Restart the servers

Consuming the streams via both mechanisms

Configure two consumers but keep routing to be KafkaHighLevel

- Change the `stream.<your stream here>.consumer.type` setting to be `highLevel,simple`. This starts both LLC and HLC consumers on the nodes.
- Change `stream.<your stream here>.consumer.prop.auto.offset.reset` to have the value largest (otherwise, llc consumers will start consuming from the beginning which may interfere with HLC operations)
- Check memory consumption and query response times.
- Set the broker routingTableuilderName to be `KafkaHighLevel` so that queries are not routed to LLC until consumption is caught up.
- Apply the table config
- Restart brokers and servers
- wait for retention period of the table.

Disabling HLC

- Change the `stream.<your stream here>.consumer.type` setting to be "simple"
- Remove the routingTableuilderName setting
- Apply the table configs and restart the brokers and servers
- The HLC segments will slowly age out on their own.

6.1.6 Migration from HLC to LLC with downtime

If it is all right to take a down time, then the easiest way is to disable the table, do the last step of the previous migration steps, and restart the table once the consumption has caught up.

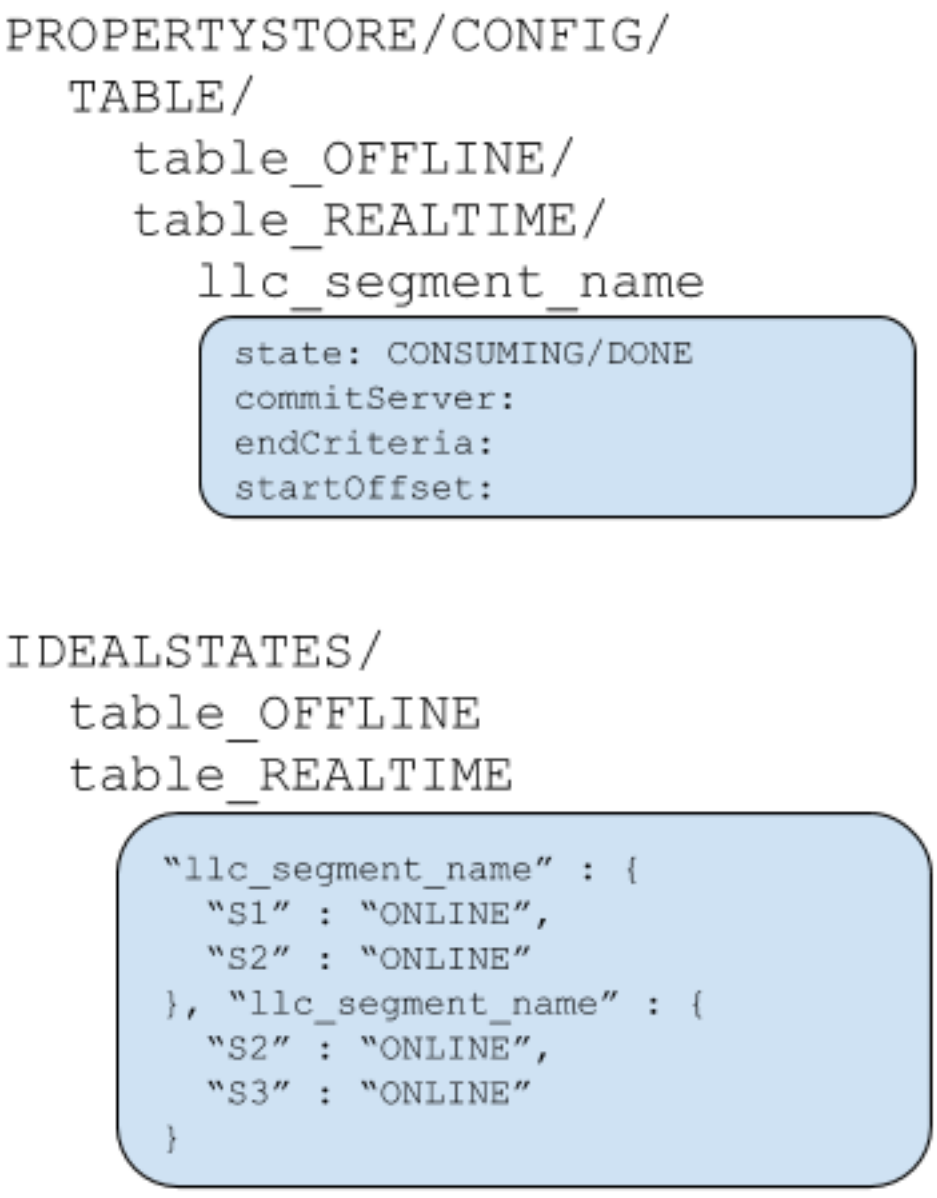


Fig. 3: Zookeeper setup

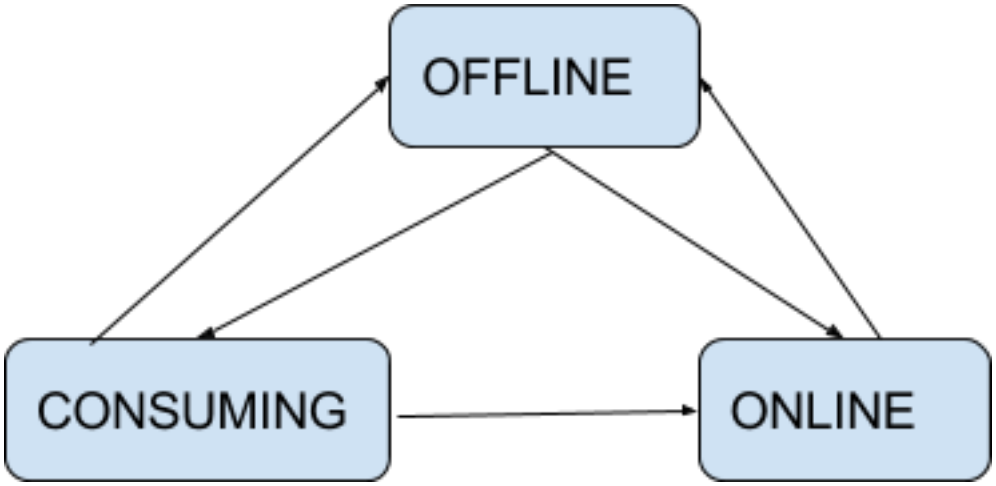


Fig. 4: Server-side Helix State Machine

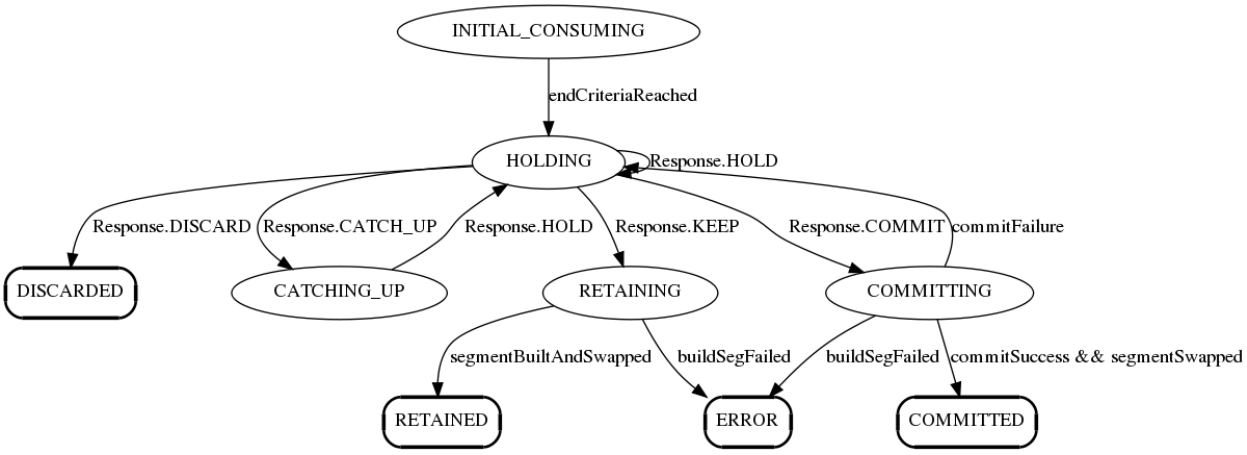


Fig. 5: Server-side Partition consumer state machine

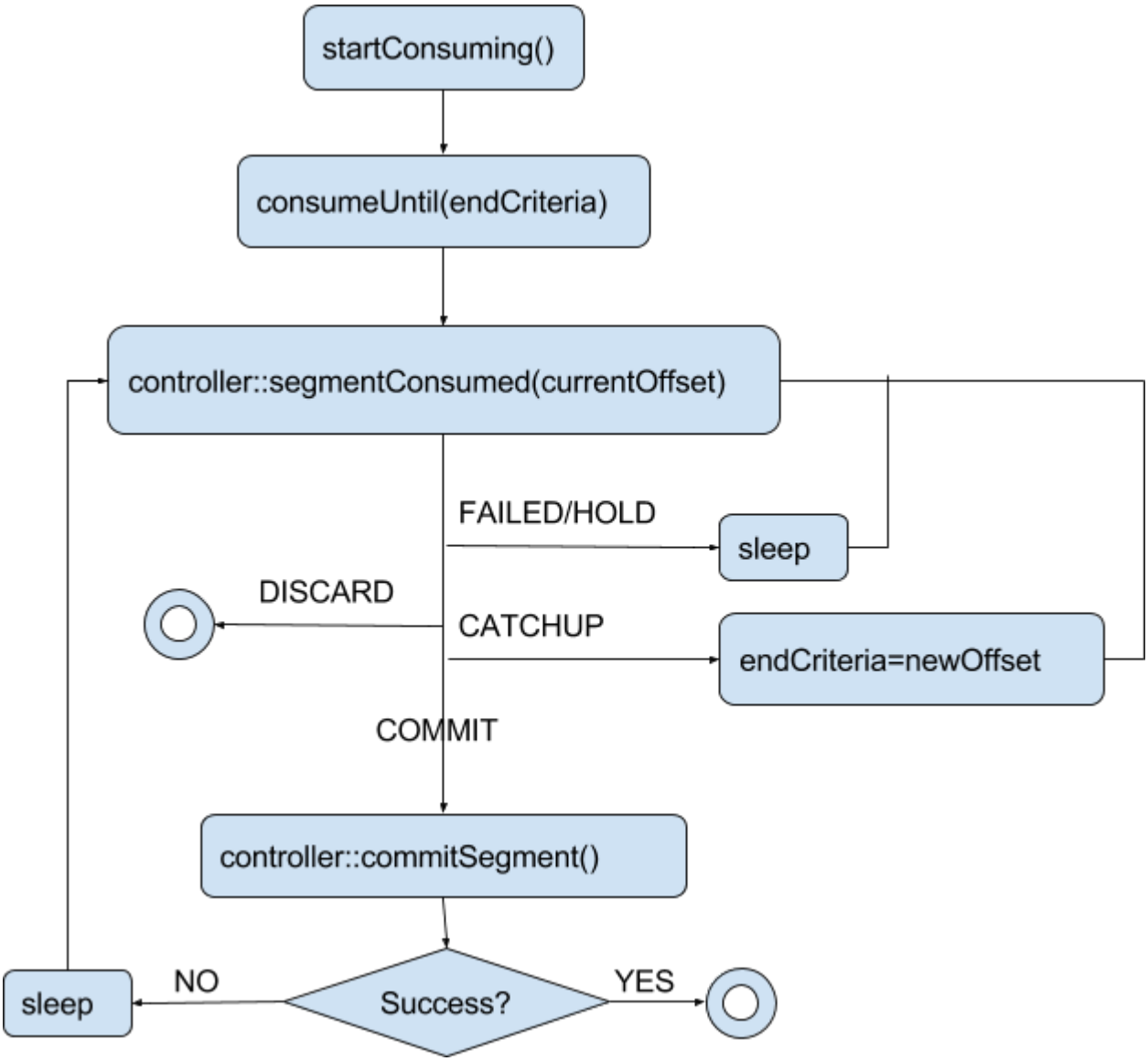


Fig. 6: Server-side control flow

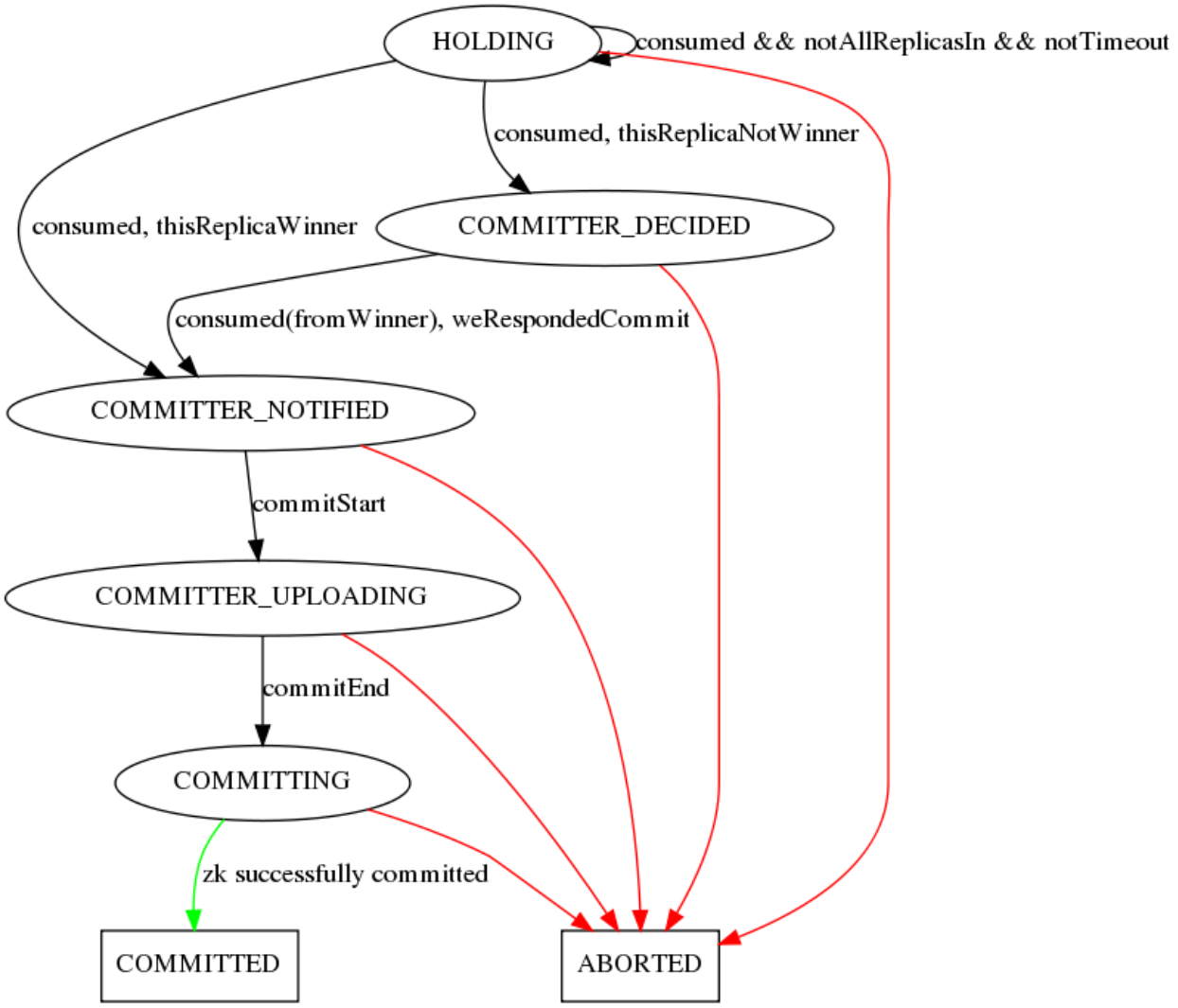


Fig. 7: Controller-side Segment completion state machine

6.1.7 LLC Zookeeper setup and Segment Completion Protocol

Scenarios

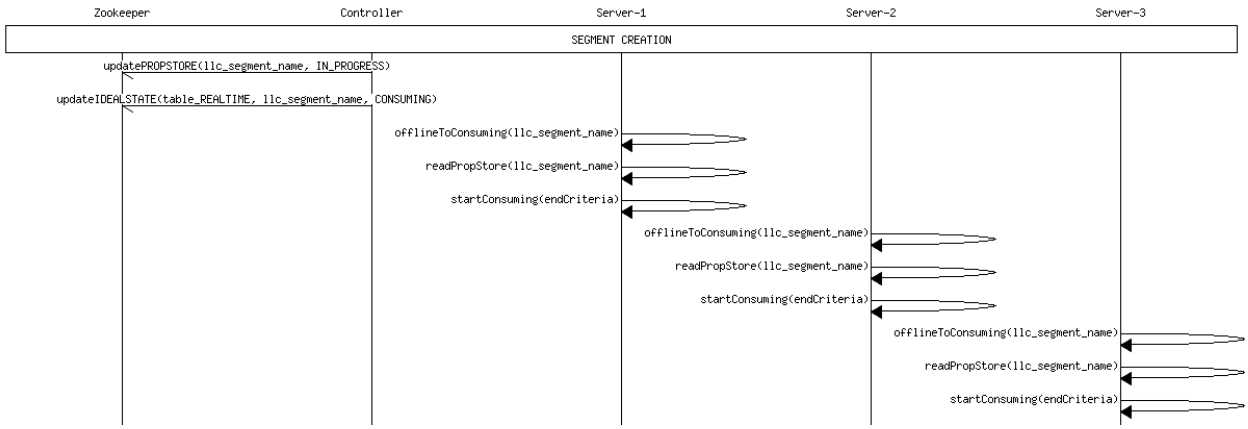


Fig. 8: Segment Creation

6.2 Partition Aware Query Routing

In ongoing efforts to support use cases with low latency high throughput requirements, we have started to notice scaling issues in Pinot broker where adding more replica sets does not improve throughput as expected beyond a certain point. One of the reason behind this is the fact that the broker does not perform any pruning of segments, and so every query ends up processing each segment in the data set. This processing of potentially unnecessary segments has been shown to impact throughput adversely.

6.2.1 Details

The Pinot broker component is responsible for receiving queries, fanning them out to Pinot servers, merging the responses from servers and finally sending the merged responses back to the client. The broker maintains multiple randomly generated routing tables that map each server to a subset of segments, such that one routing table covers one replica set (across various servers). This implies that for each query all segments of a replica set are processed for a server.

This becomes an overhead when the answer for the given query lies within a small subset of segments. One very common example is when queries have a narrow time filter (say 30 days), but the retention is two years (730 segments, at the rate of one segment per day). For each such query there are multiple overheads:

> Broker needs to use connections to servers that may not even be hosting any segments worth processing. On the server side, there is query planning as well as execution once per segment. This happens for hundreds (or even few thousands) of segments, when only a few need to be actually processed.

We observed through experiments as well as prototyping that while these overheads may (or may not) impact latency, they certainly impact throughput quite a bit. In an experiment with one SSD node with 500 GB of data (720 segments), we observed a baseline QPS of 150 without any pruning and pruning on broker side improves QPS to about 1500.

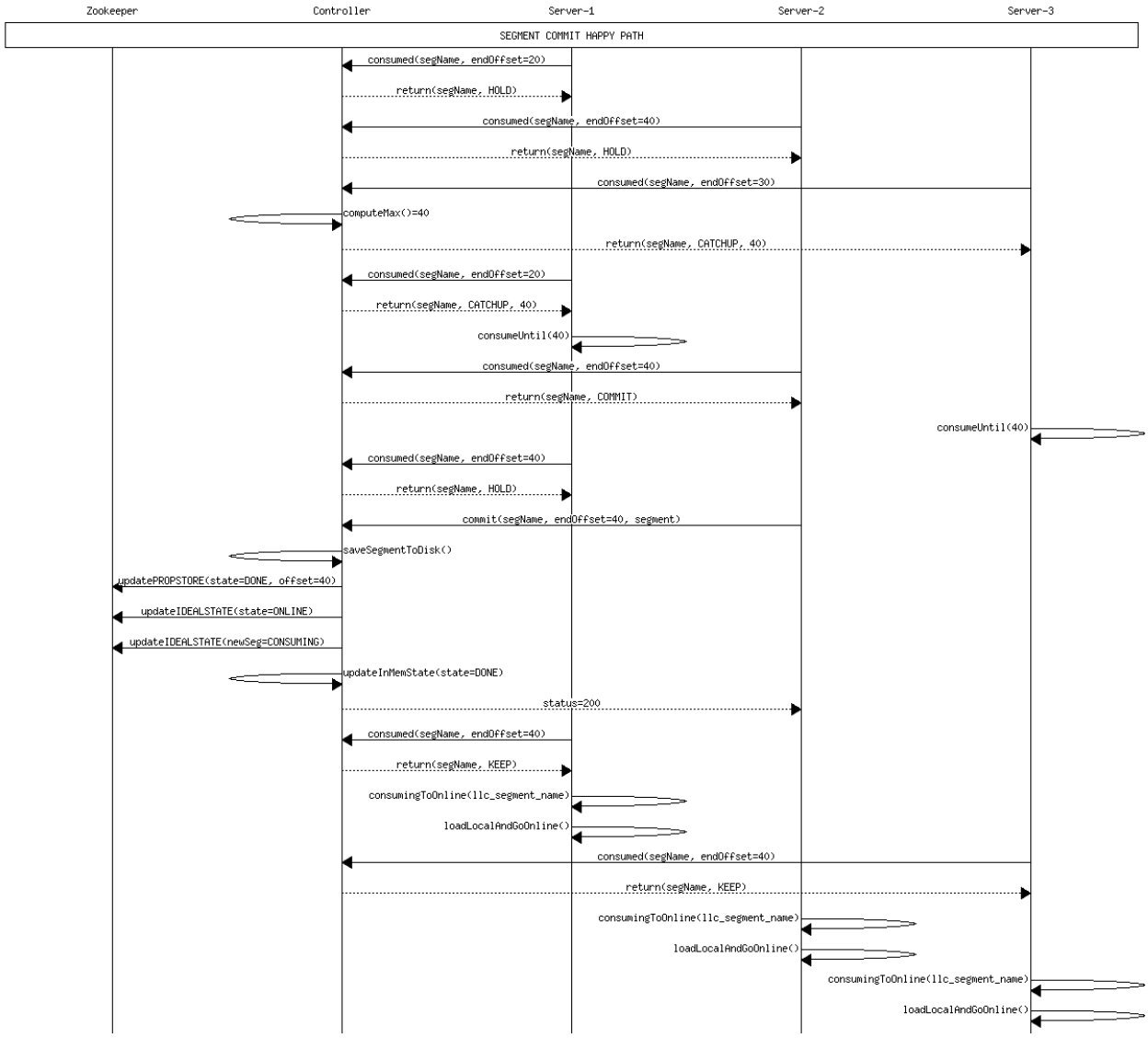


Fig. 9: Happy path commit 1

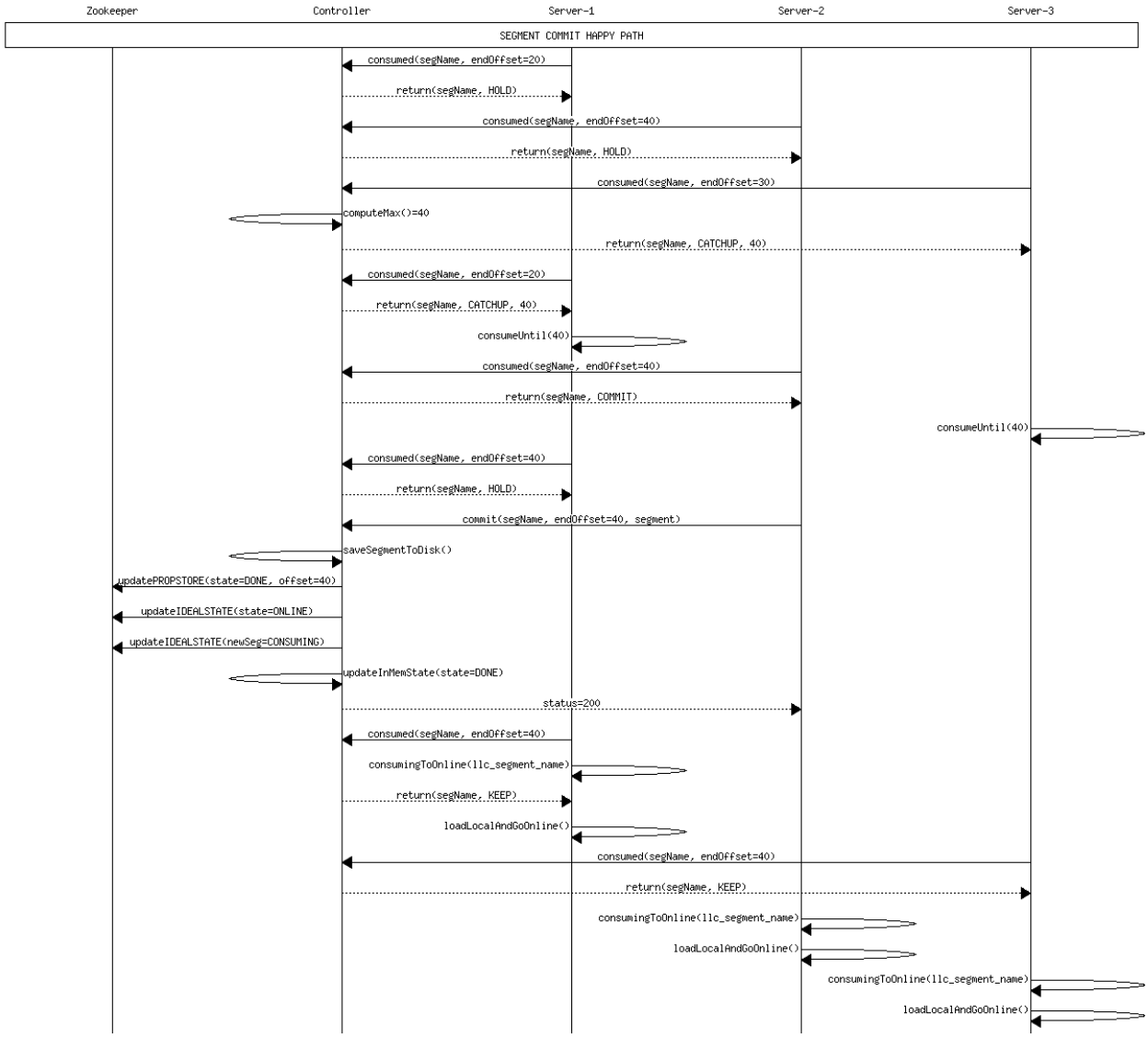


Fig. 10: Happy path commit 2

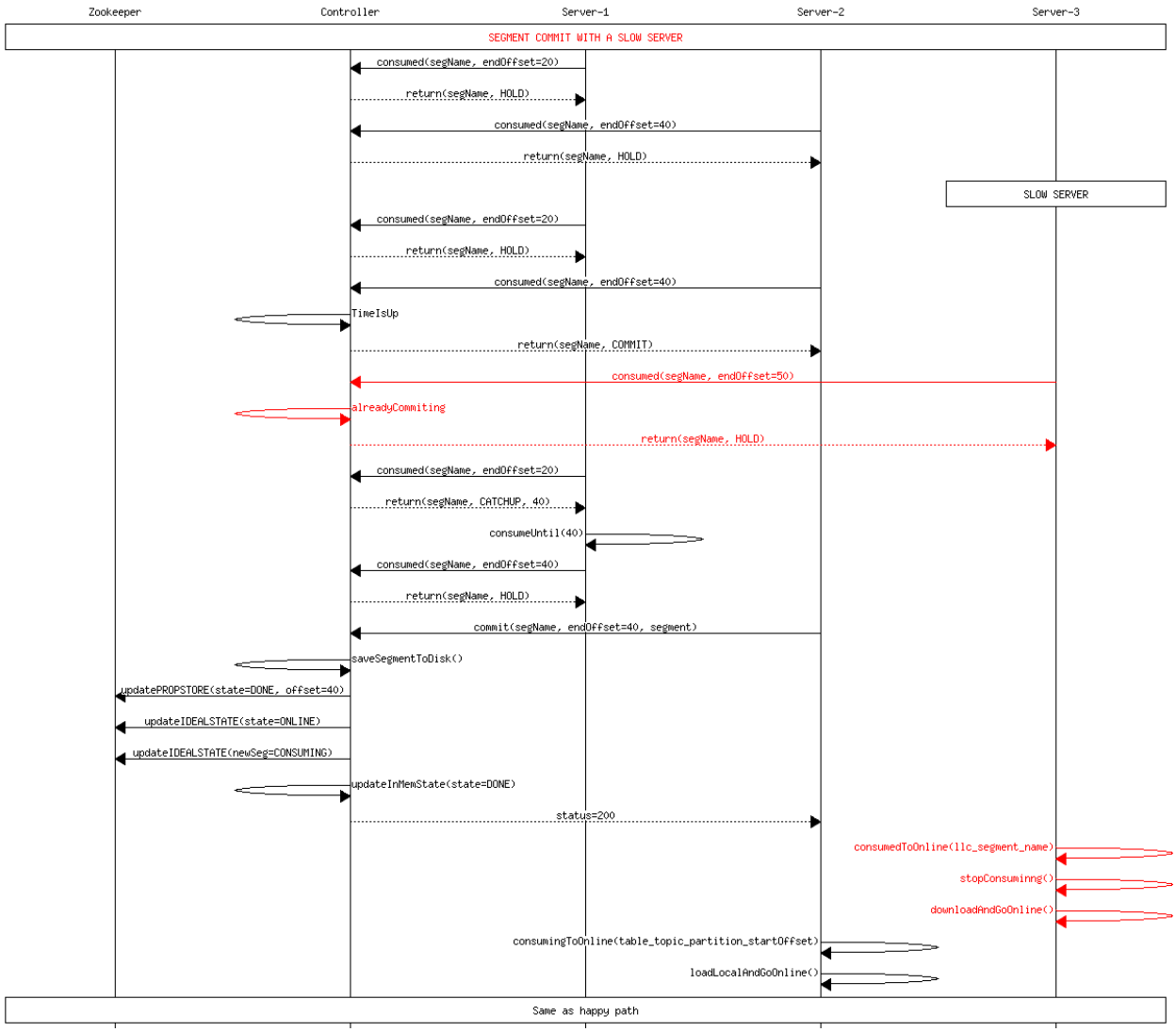


Fig. 11: Delayed Server

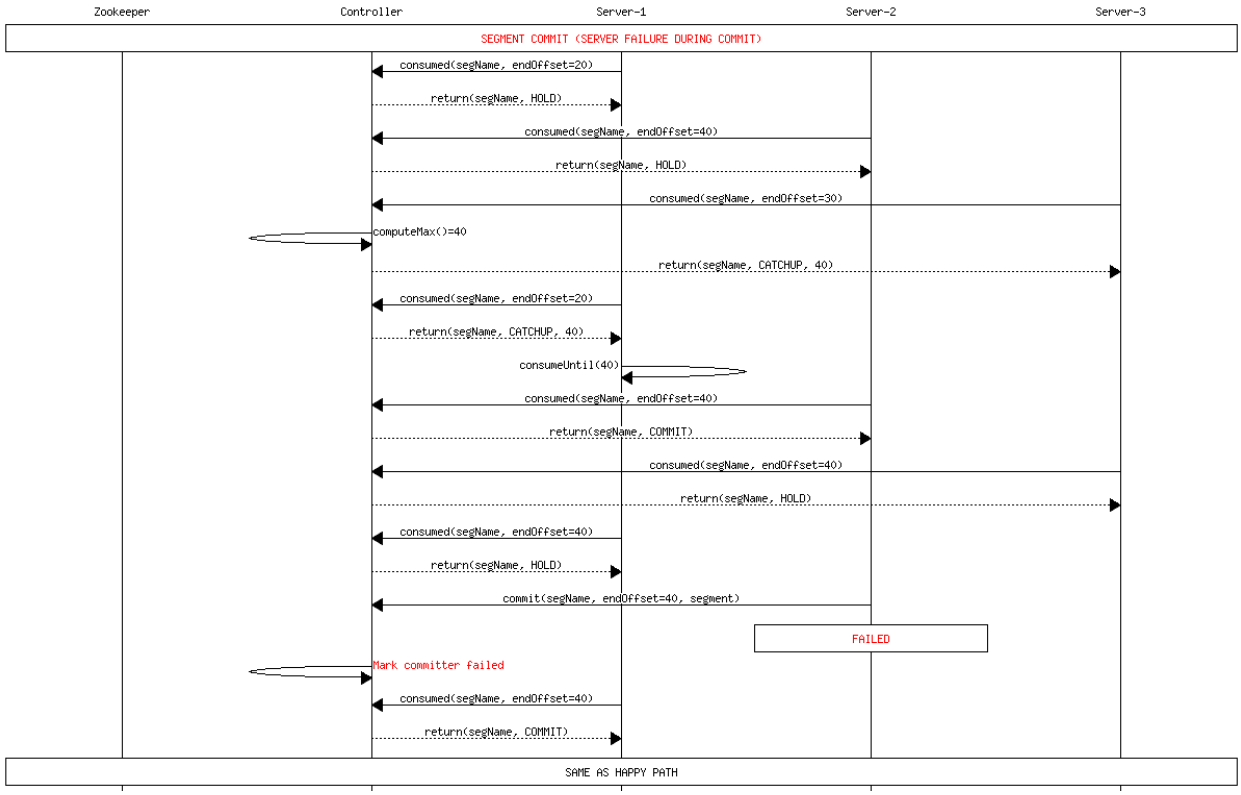


Fig. 12: Committer failure

6.2.2 Proposed Solution

We propose a solution that would allow the broker to quickly prune segments for a given query, reducing the overheads and improving throughput. The idea is to make information available in the segment metadata for broker to be able to quickly prune a segment for a given query. Once the broker has compiled the query, it has the filter query tree that represents the predicates in the query. If there are predicates on column(s) for which there is partition/range information in the metadata, the broker would be able to quickly prune segments that would not satisfy the predicates.

In our design, we propose two new components within the broker:

- **Segment ZK Metadata Manager**: This component will be responsible for caching the segment ZK metadata in memory within the broker. Its cache will be kept upto date by listening to external view changes.
- **Segment Pruner**: This component will be responsible pruning segments for a given query. Segments will be pruned based on the segment metadata and the predicates in the query.

6.2.3 Segment ZK Metadata Manager

While the broker does not have access to the segments themselves, it does have access to the ZK metadata for segments. The SegmentZKMetadataManager will be responsible for fetching and caching this metadata for each segment. Upon transition to online state, the SegmentZKMetadataManager will go over the segments of the table(s) it hosts and build its cache. It will use the existing External View change listener to update its cache. Since External View does not change when segments are refreshed, and setting watches for each segment in the table is too expensive, we are choosing to live with a limitation where this feature does not work for refresh usecase. This limitation can be revisited at a later time, when inexpensive solutions are available to detect segment level changes for refresh usecases.

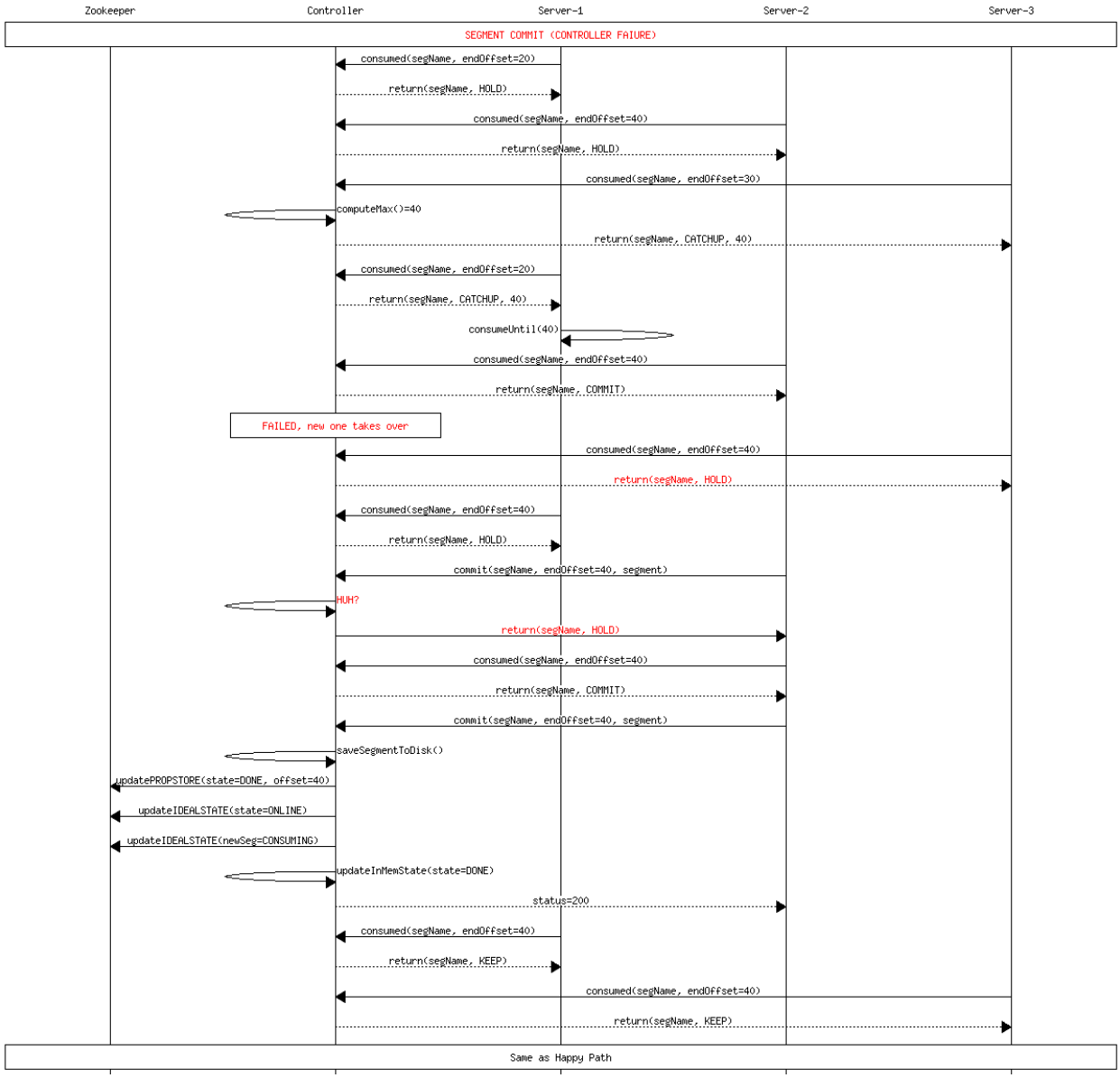


Fig. 13: Controller failure during commit

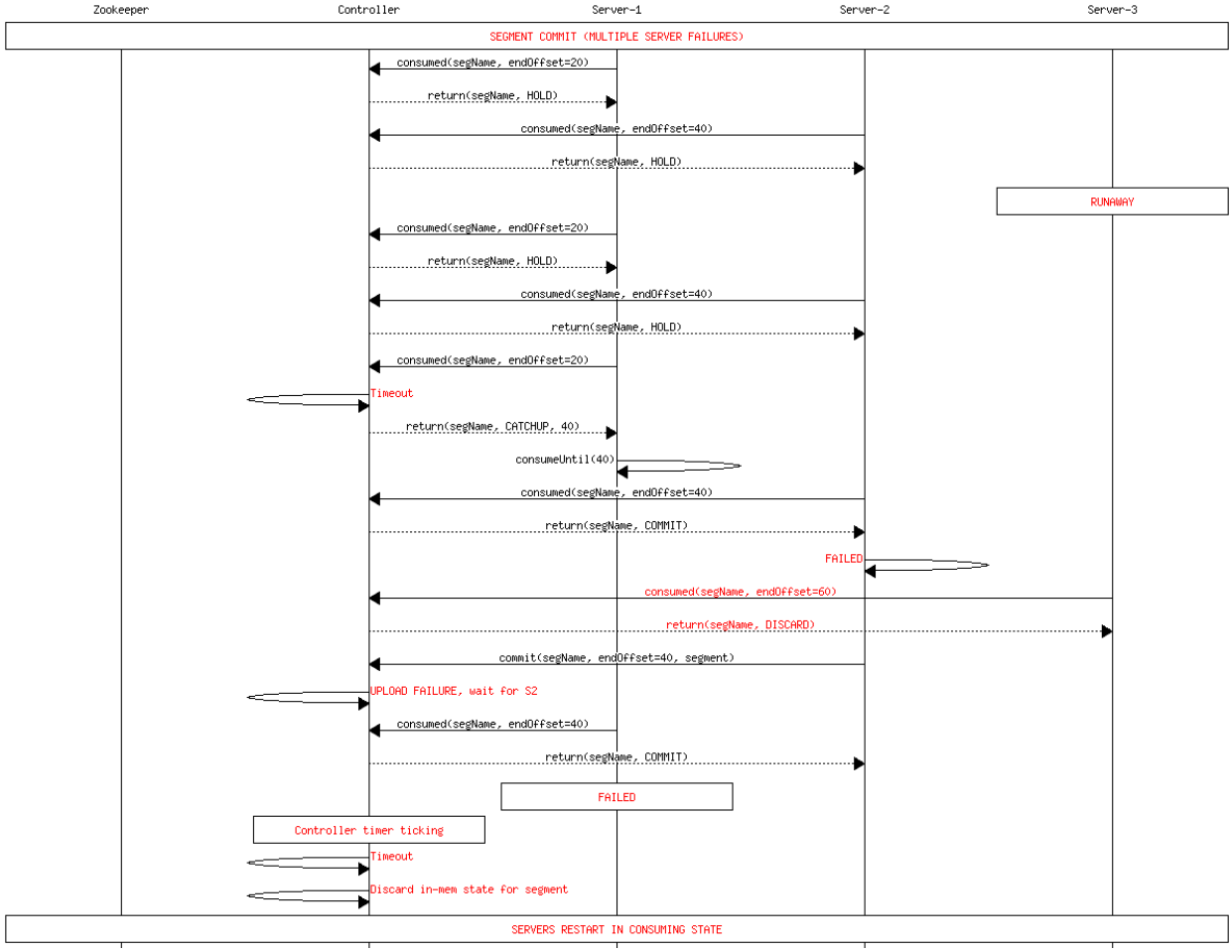


Fig. 14: Multiple failures

6.2.4 Table Level Partition Metadata

We will introduce a table level config to control enabling/disabling this feature for a table. This config can actually be the pruner class name, so that realtime segment generation can pick it up. Absence or incorrect specification of this table config would imply the feature is disabled for the table.

6.2.5 Segment Level Partition Metadata

The partition metadata would be a list of tuples, one for each partition column, where each tuple contains: Partition column: Column on which the data is partitioned. Partition value: A min-max range with [start, end]. For single value start == end. Prune function: Name of the class that will be used by the broker to prune a segment based on the predicate(s) in the query. It will take as argument the predicate value and the partition value, and return true if segment can be pruned, and false otherwise.

For example, let us consider a case where the data is naturally partitioned on time column 'daysSinceEpoch'. The segment zk metadata will have information like below:

```
{
  "partitionColumn"   : "daysSinceEpoch",
      "partitionStart"        : "17200",
   "partitionEnd"             : "17220",
       "pruneFunction"                : "TimePruner"
}
```

Now consider the following query comes in.

```
Select count(*) from myTable where daysSinceEpoch between 17100 and 17110
```

The broker will recognize the range predicate on the partition column, and call the TimePruner which will identify that the predicate cannot be satisfied and hence and return true, thus pruning the segment. If there is no segment metadata or there is no range predicate on partition column, the segment will not be pruned (return false).

Let's consider another example where the data is partitioned by memberId, where a hash function was applied on the memberId to compute a partition number.

```
{
  "partitionColumn"   : "memberId",
      "partitionStart"        : "10",
  "partitionEnd"              : "10",
      "pruneFunction"         : "HashPartitionPruner"
}

Select count(*) from myTable where memberId = 1000`
```

In this case, the HashPartitionPruner will compute the partition id for the memberId (1000) in the query. And if it turns out to anything other than 10, the segment would be pruned out. The prune function would contain the complete logic (and information) to be able to compute partition id's from meberId's.

6.2.6 Segment Pruner

The broker will perform segment pruning as follows. This is not an exact algorithm but meant for conveying the idea. Actual implementation might differ slightly (if needed).

1. Broker will compile the query and generate filter query tree as it does currently.
2. The broker will perform a DFS on the filter query tree and prune the segment as follows:

- If the current node is leaf and is not the partition column, return false (not prune).
- If the current node is leaf and is the partition column, return the result of calling prune function with predicate value from leaf node, and partition value from metadata.
- If the current node is AND, return true as long as one of its children returned true (prune).
- If the current node is OR, return true if all of its children returned true (prune).

6.2.7 Segment Generation

The segment generation code would be enhanced as follows: It already auto-detects sorted columns, but only writes out the min-max range for time column to metadata. It will be enhanced to write out the min-max range for all sorted columns in the metadata.

For columns with custom partitioning schemes, the name of partitioning (pruner) class will be specified in the segment generation config. Segment generator will ensure that the column values adhere to partitioning scheme and then will write out the partition information into the metadata. In case column values do not adhere to partition scheme, it will log a warning and will not write partition information in the metadata. The impact of this will be that broker would not able to perform any pruning on this segment.

This will apply to both offline as well as realtime segment generation, except that for realtime the pruner class name is specified in the table config. When the creation/annotation of segment ZK metadata from segment metadata happens in controller (when adding a new segment) the partition info will also be copied over.

6.2.8 Backward compatibility

This feature will be ensured to not create any backward compatibility issues. New code with old segments will not find any partition information and pruning will be skipped. Old code will not look for the new information in segment Zk metadata and will work as expected.

6.2.9 Query response impact

The total number of documents in the table is returned as part of the query response. This is computed by servers when they process segments. If some segments are pruned, their documents will not be accounted for in the query response.

To address this, we will enhance the Segment ZK metadata to also store the document count of the segment, which the broker has access to. The total document count will then be computed on the broker side instead.

6.2.10 Partitioning of existing data

In the scope of this project, existing segments would not be partitioned. This simply means that pruning would not apply to those segments. If there's a existing usecase that would benefit from this, then there are a few possibilities that can be explored (outside the scope of this project):

6.2.11 Data can be re-bootstrapped after partitioning

If the existing segments already comply to some partitioning, a utility can be created to to update the segment metadata and re-push the segments.

6.2.12 Results

With Partition aware segment assignment and query routing, we were able to demonstrate 6k qps with 99th %ile latency under 100ms, on a data size of 3TB, in production.

6.2.13 Limitations

The initial implementation will have the following limitations: It will not work for refresh usecases because currently there's no cheap way to detect segment ZK metadata change for segment refresh. The only available way is to set watches at segment level that will be too expensive. Segment generation will not partition the data itself, but assume and assert that input data is partitioned as specified in the config.

6.3 Expressions and UDFs

6.3.1 Requirements

The query language for Pinot (*PQL*) currently only supports *selection*, *aggregation* & *group by* operations on columns, and moreover, do not support nested operations. There are a growing number of use-cases of Pinot that require some sort of transformation on the column values, before and/or after performing *selection*, *aggregation* & *group by*. One very common example is when we would want to aggregate *metrics* over different granularity of times, without needing to pre-aggregate records on the individual granularity outside of Pinot. Another example would be when we would want to aggregate on *function* (say difference) of multiple columns.

The high level requirement here is to support *expressions* that represent a function on a set of columns in the queries, as opposed to just columns.

```
select <exp1> from myTable where ... [group by <exp2>]
```

Where exp1 and exp2 can be of the form:

```
func1(func2(col1, col2...func3(...)...), coln...)...
```

6.3.2 Proposal

We first propose the concept of a Transform Operator (*xform*) below. We then propose using these *xform* operators to perform transformations before/after *selection*, *aggregation* and *group by* operations.

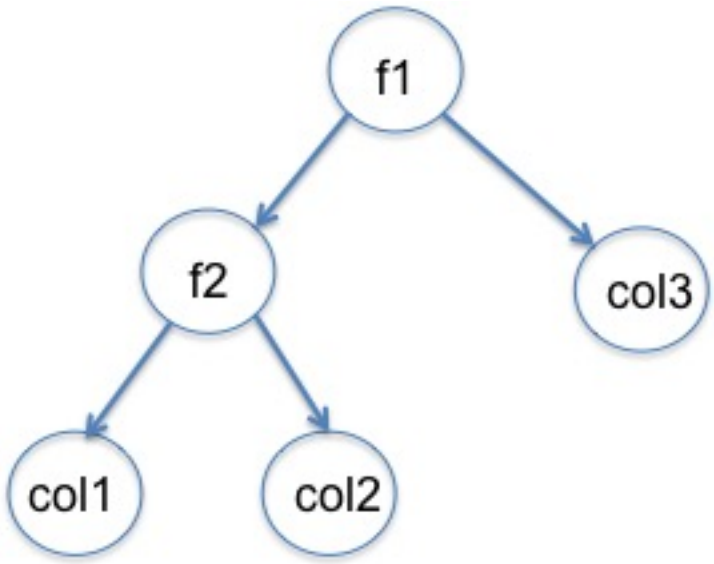
The *xform* operator takes following inputs:

1. An expression tree capturing *functions* and the *columns* they are applied on. The figure below shows one such tree for expression: `f1(f2(col1, col2), col3)`
2. A set of document Id's on which to perform *xform*

The *xform* produces the following output:

- For each document Id in the input, it evaluates the specified expression, and produces one value.
  - It is Many:1 for columns, i.e. many columns in the input produce one column value in the output.
  - It is 1:1 for document Id's, i.e. for each document in the input, it produces one value in the output.

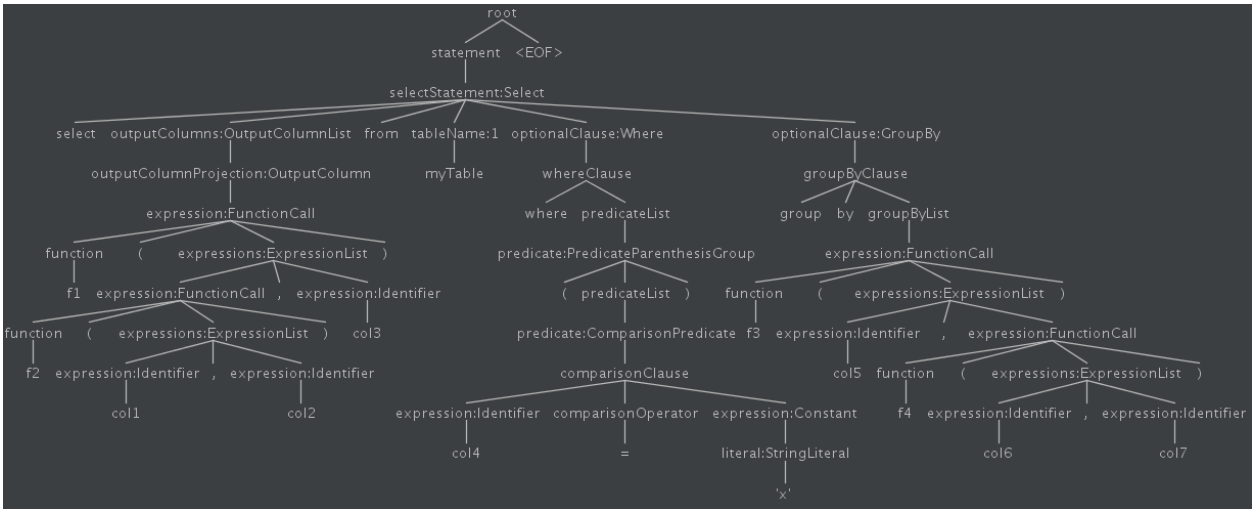
The *functions* in the *expression* can be either built-in into Pinot, or can be user-defined. We will discuss the mechanism for hooking up *UDF* and the manageability aspects in later sections.



6.3.3 Parser

The PQL parser is already capable of parsing expressions in the *selection*, *aggregation* and *group by* sections. Following is a sample query containing expression, and its parse tree shown in the image.

```
select f1(f2(col1, col2), col3) from myTable where (col4 = 'x') group by f3(col5,␣
↪f4(col6, col7))
```



6.3.4 BrokerRequest

We convert the Parse Tree from the parser into what we refer to as *BrokerRequest*, which captures the parse tree along with other information and is serialized from Pinot broker to server. While the parser does already recognize expressions in these sections, the *BrokerRequest* currently assumes these to be columns and not expressions. We propose the following enhancements here:

1. *BrokerRequest* needs to be enhanced to support not just *columns* but also *expressions* in the *selection*, *aggregation* & *group by* sections. *BrokerRequest* is currently implemented via 'Thrift'. We will need to enhance

request.thrift to be able to support expressions. There are a couple of options here:

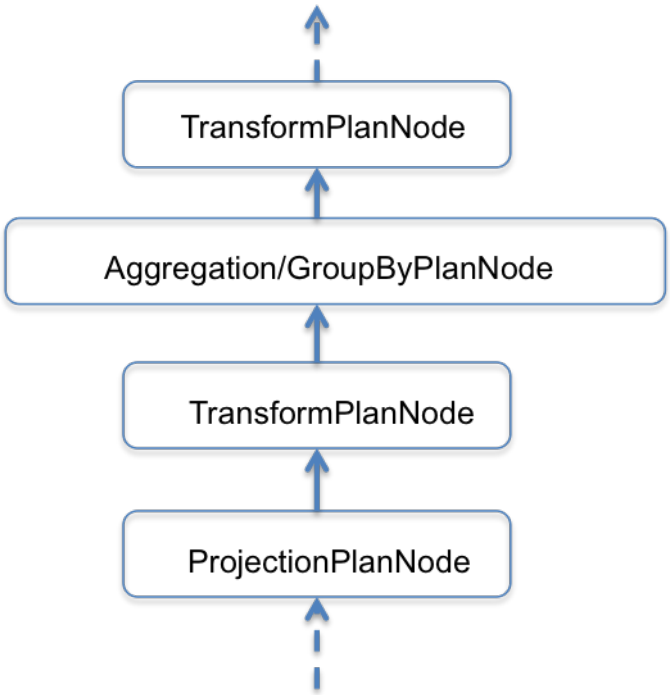
- We use the same mechanism as *FilterQuery* (which is how the predicates are represented).
- Evaluate other structures that may be more suitable for expression evaluation. (TBD).

1. The back-end of the parser generates *BrokerRequest* based on the parse tree of the query. We need to implement the functionality that takes the parse tree containing expressions in these sections and generates the new/enhanced *BrokerRequest* containing expressions.

6.3.5 Query Planning

In the query planning phase, Pinot server receives a *BrokerRequest* (per query) and parses it to build a query plan, where it hooks up various plan nodes for filtering, *Selection/Aggregation/GroupBy*, combining together.

A new *TransformPlanNode* class will be implemented that implements the *PlanNode* interface. The query planning phase will be enhanced to include new *xform* plan nodes if the *BrokerRequest* contains *expressions* for *selection*, *aggregation* & *group by*. These plan nodes will get hooked up appropriately during planning phase.



6.3.6 Query Execution

In the query execution phase, the *run* method for *TransformPlanNode* will return a new *TransformOperator*. This operator is responsible for applying a transformation to a given set of documents, as specified by the *expression* in the query. The output *block* of this operator will be fed into other operators as per the query plan.

6.3.7 UDFs

The functions in *expressions* can either be built-in functions in Pinot, or they can be user-defined. There are a couple of approaches for supporting hooking up of UDF's into Pinot:

1. If the function is generic enough and reusable by more than one clients, it might be better to include it as part of Pinot code base. In this case, the process for users would be to file a pull-request, which would then be reviewed and become part of Pinot code base.
2. Dynamic loading of user-defined functions:

- Users can specify jars containing their UDF's in the class path.
- List of UDF's can be specified in server config, and the server can ensure that it can find and load classes for each UDF specified in the config. This allows for a one-time static checking of availability of all specified UDF's.
- Alternatively, the server may do a dynamic check for each query to ensure all UDF's specified in the query are available and can be loaded.

6.3.8 Backward compatibility

Given that this proposal requires modifying *BrokerRequest*, we are exposed to backward compatibility issues where different versions of broker and server are running (one with the new feature and another without). We propose to address this as follows:

1. The changes to *BrokerRequest* to include *expressions* instead of *columns* would only take effect if a query containing *expression* is received. For the query just contains *columns* instead of *expressions*, we fall be to existing behavior and send the *columns* as they are being sent in the current design (ie not as a special case of an *expresion*).
2. This will warrant the following sequencing: * Broker upgraded before server. * New queries containing *expressions* should be sent only after both broker and server are upgraded.

6.3.9 Limitations

We see the following limitations in functionality currently:

1. Nesting of *aggregation* functions is not supported in the expression tree. This is because the number of documents after *aggregation* is reduced. In the expression below, *sum* of *col2* would yield one value, whereas *xform1* one *col1* would yield the same number of documents as in the input.

```
sum(xform1(col1), sum(col2))
```

1. The current parser does not support precedence/associativity of operators, it just builds parse tree from left to right. Addressing this is outside of the scope of this project. Once the parser is enhanced to support this, *expression* evaluation within query execution would work correctly without any code changes required.

6.4 Schema TimeSpec Refactoring

6.4.1 Problems with current schema design

The pinot schema timespec looks like this:

```
{
  "timeFieldSpec":
  {
    "name" : <name of time column>,
    "dataType" : <datatype of time column>,
    "timeFormat" : <format of time column, EPOCH or SIMPLE_DATE_FORMAT:format>,
```

(continues on next page)

(continued from previous page)

```
    "timeUnitSize" : <time column granularity size>,
    "timeType" : <time unit of time column>
  }
}
```

We are missing data granularity information in pinot schema. TimeUnitSize, timeType and timeFormat allow us to define the granularity of the time column, but don't provide a way for applications to know in what buckets the data granularity is. Currently, we can only have one time column in the table which is limiting some use cases. We should allow multiple time columns and even allow derived time columns. Derived columns can be useful in performing roll ups or generating star tree aggregate nodes.

6.4.2 Changes

We have added a List<DateTimeFieldSpec> _dateTimeFieldSpecs to the pinot schema

```
{
  "dateTimeFieldSpec":
    {
      "name" : <name of the date time column>,
      "dataType" : <datetype of the date time column>,
      "format" : <string for interpreting the datetime column>,
      "granularity" : <string for data granularity buckets>,
      "dateTimeType" : <DateTimeType enum PRIMARY,SECONDARY or DERIVED>
    }
}
```

1. name - this if the name of the date time column, similar to the older timeSpec
2. dataType - this is the DataType of the date time column, similar to the older timeSpec

#. format - defines how to interpret the numeric value in the date time column.
Format has to follow the pattern - size:timeunit:timeformat, where size and timeUnit together define the granularity of the time column value.
Size is the integer value of the granularity size.
TimeFormat tells us whether the time column value is expressed in epoch or is a simple date format pattern.
Consider 2 date time values for example 2017/07/01 00:00:00 and 2017/08/29 05:20:00: 1. If the time column value is defined in millisSinceEpoch (1498892400000, 1504009200000), this configuration will be 1:MILLISECONDS:EPOCH 2. If the time column value is defined in 5 minutes since epoch (4996308, 5013364), this configuration will be 5:MINUTES:EPOCH 3. If the time column value is defined in a simple date format of a day (e.g. 20170701, 20170829), this configuration will be 1:DAYS:SIMPLE_DATE_FORMAT:yyyyMMdd (the pattern can be configured as desired)

#. granularity - defines in what granularity the data is bucketed.
Granularity has to follow pattern- size:timeunit, where size and timeUnit together define the bucket granularity of the data. This is independent of the format, which is purely defining how to interpret the numeric value in the datetime column. 1. if a time column is defined in millisSinceEpoch (format=1:MILLISECONDS:EPOCH), but the data buckets are 5 minutes, the granularity will be 5:MINUTES. 2. if a time column is defined in hoursSinceEpoch (format=1:HOURS:EPOCH), and the data buckets are 1 hours, the granularity will be 1:HOURS

#. dateTimeType - this is an enum of values 1. PRIMARY: The primary date time column. This will be the date time column which keeps the milliseconds value. This will be used as the default time column, in references by pinot code (e.g. retention manager) 2. SECONDARY: The date time columns which are not the primary columns with milliseconds value. These can be date time columns in other granularity, put in by applications for their specific use cases 3. DERIVED: The date time columns which are derived, say using other columns, generated via rollups, etc

Examples:

```
"dateTimeFieldSpec":
{

  "name" : "Date",
  "dataType" : "LONG",
  "format" : "1:HOURS:EPOCH",
  "granularity" : "1:HOURS",
  "dateTimeType" : "PRIMARY"

}

"dateTimeFieldSpec":
{

  "name" : "Date",
  "dataType" : "LONG",
  "format" : "1:MILLISECONDS:EPOCH",
  "granularity" : "5:MINUTES",
  "dateTimeType" : "PRIMARY"

}

"dateTimeFieldSpec":
{

  "name" : "Date",
  "dataType" : "LONG",
  "format" : "1:DAYS:SIMPLE_DATE_FORMAT:yyyyMMdd",
  "granularity" : "1:DAYS",
  "dateTimeType" : "SECONDARY"

}
```

6.4.3 Migration

Once this change is pushed in, we will migrate all our clients to start populating the new DateTimeFieldSpec, along with the TimeSpec.
We can then go over all older schemas, and fill up the DateTimeFieldSpec referring to the TimeFieldSpec.
We then migrate our clients to start using DateTimeFieldSpec instead of TimeFieldSpec.
At this point, we can deprecate the TimeFieldSpec.

CHAPTER 7

Design Proposals

7.1 Multitenancy

7.1.1 Problems with Multiple cluster in Pinot 1.0

In Pinot 1.0, we created one cluster for every engagement. While this was good in the beginning, it causes maintenance head aches and also delays on boarding new engagements.

7.1.2 Engagement

Here is the typical process of on boarding a new engagement.

1. Capacity planning, estimate the number of nodes needed.
2. Place new hardware request and wait for allocation.
3. Once we get the the hardware, tag the nodes and deploy the software

The above steps take time and every new engagement causes disruption for developers and engagements don't understand the process and feel that infrastructure team is slowing them down. Our goal is to drastically reduce the on boarding time on Pinot and also minimize the involvement from developers in on boarding new engagements.

7.1.3 Cluster Maintenance

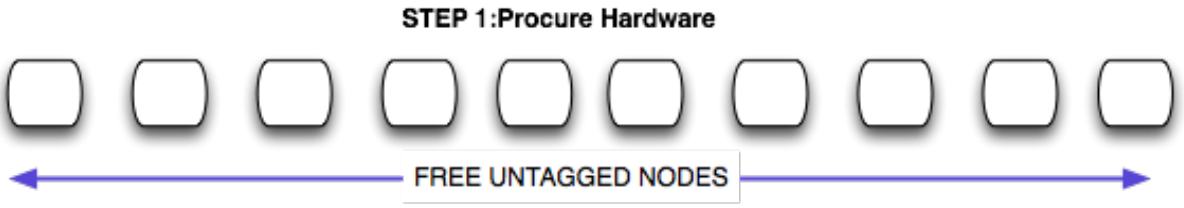
Even maintenance becomes harder as we have more number of clusters. Some of the problems with having multiple clusters

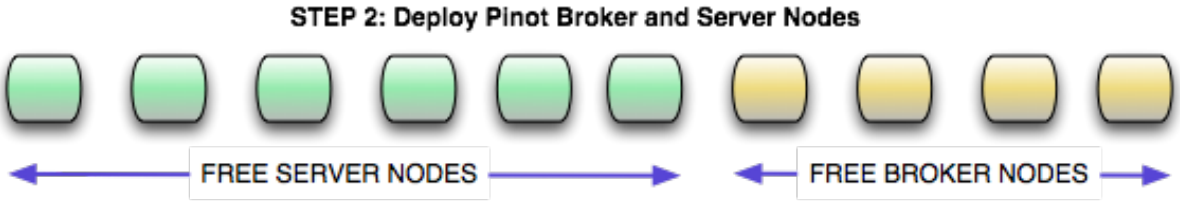
- Too many versions to manage. Lot of clusters continue to run with old version. When there is a bug, we upgrade one cluster because its urgent but never get to upgrade remaining clusters.
- Promotes per tenant configuration. Since each cluster can have its own tag based configuration. We end up having too many configuration parameters that are specific to a particular tenant.

7.1.4 Hardware utilization (cost to serve)

Having separate set of hardware for every client means we cannot use the hardware in a cost effective manner. Most of the boxes in Pinot are under utilized and can easily support multiple use cases. However because of the way it is designed in Pinot 1.0, we end up creating separate clusters for each tenant. Co-locating multiple tenants on same hardware can reduce the number of boxes needed. While this is risky for external/site facing use cases, this can be used for internal use cases.

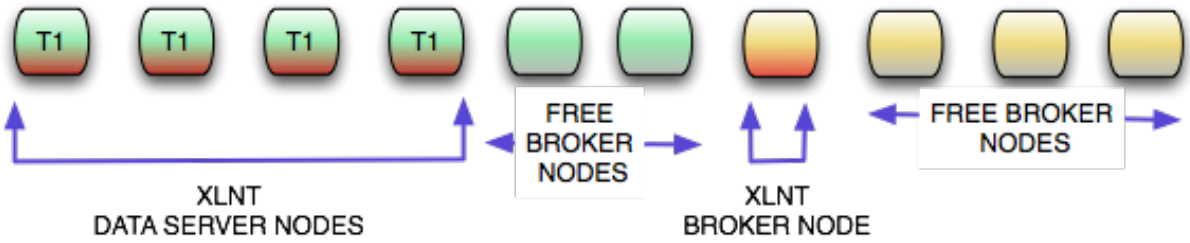
7.1.5 Multi tenancy in Pinot 2.0





STEP 3: ADD TENANT: CREATE RESOURCE COMMAND

resourceName	XLNT
tableName	T1
numberOfDataInstances	4
numberOfCopies	2
brokerTagName	XLNT
numberOfBrokerInstances	1



In Pinot 2.0, we designed the system assuming that it will be Multi tenant from day 1. We will have only one cluster for all tenants. Helix will be used to drive the multi tenancy in Pinot. The key ideas here are

1. Unlike Pinot 1.0 where we order nodes on a per tenant basis, we order hardware in bulk. SRE will install the same software version on all boxes, these boxes will start up and register in Helix. This allows us the SRE's to configure and deploy the software on these boxes in one go.
2. Make use of tagging feature provided by Helix. Helix allows one to Tag/untag a node dynamically. All instances are untagged when they join the cluster for the first time.
3. On boarding a new engagement is as simple as tagging a node in Helix and assigning segments to the appropriate nodes in the cluster.

7.1.6 Example flow

- Procure 100 nodes from Ops in the beginning of the quarter and deploy pinot code.
- All nodes will have the tag "untagged"
- Lets say we get an use case "XLNT". We do capacity planning and estimate that we need 10 nodes in total (Including replication factor). Pinot Controller automatically grabs 10 nodes from the pool and tags them as "XLNT". All segments arriving from Hadoop and real time segments will be assigned to one of these boxes.
- When we reach close to the capacity we get new hardware and add them to this cluster.

7.1.7 Cluster maintenance

- With this approach, all nodes in the cluster can be upgraded at once.
- We might however some times want to upgrade only a set of machines. Current tooling at LinkedIn does not understand Helix metadata, hence we will write a small wrapper script that reads the information from Helix and upgrades the boxes that belong to a particular tenant.
- Canary: we will tag some nodes as canary and deploy our golden data set on it. Every release will be first deployed to these canary nodes before deploying on rest of the nodes.

7.1.8 Monitoring

- With Pinot 1.0, we would have one in graph dashboard per tenant because we tag the nodes in svn when they are assigned to a tenant. With Pinot 2.0, we can dynamically assign a node to any tenant. This makes it hard to have a per tenant dashboard. We solve this problem by having a convention in naming our metrics. Every metric we log in Auto metrics will have tenant name as part of it. SRE can use regex feature in InGraphs to filter metrics that belong to a tenant and generate per tenent dashboard.
- Metric naming convention: `(pinot_server|pinot_broker|pinot_controller)_resourceName_tableName.metricName`

7.1.9 Pinot Broker

In Pinot 1.0, we had embedded pinot broker within every pinot server. While this simplified deployment, it made it hard to capacity plan appropriately. Pinot broker and Pinot Server differ quite a bit in resource usage and workload patterns. Often, we wanted to add additional servers without increasing the number of brokers but this was not possible since adding a server meant adding additional broker as well. In pinot 2.0, we separated pinot-broker and pinot-server into separate deployable. Note: we can still deploy them together. This allowed us to make our brokers multi tenant. Unlike pinot-servers, in case of pinot-brokers we can make them truly multi tenant since they are state less. Especially

for all internal use cases where Pinot serves as the back end for UI, the qps is pretty low and we can easily share brokers across multiple tenants.

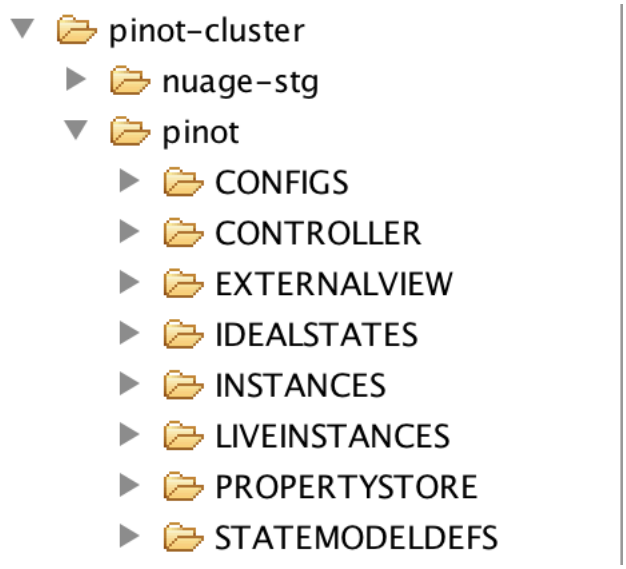
7.1.10 Helix Layout

All cluster state of Pinot is managed by Helix. The following links will help you understand the general layout of ZNodes in Helix.

- ZNode Layout in Helix
- Helix ZNode description

7.1.11 Pinot Cluster creation

When the cluster is created the Zookeeper ZNode layout looks as follows.



7.1.12 Adding Nodes to cluster

Adding node to cluster can be done in two ways, manual or automatic. This is controlled by a property set in cluster config called "allowPariticpantAutoJoin". If this is set to true, participants can join the cluster when they are started. If not, they need to be pre-registered in Helix via Helix Admin command addInstance.

```
{
 "id" : "PinotPerfTestCluster",
 "simpleFields" : {
 "allowParticipantAutoJoin" : "true"
 },
 "mapFields" : { },
 "listFields" : { }
}
```

In Pinot 2.0 we will set AUTO_JOIN to true. This means after the SRE's procure the hardware they can simply deploy the Pinot war and provide the cluster name. When the nodes start up, they join the cluster and registers themselves as server_untagged or broker_untagged. This is what one would see in Helix.

The znode `CONFIGS/PARTICIPANT/ServerInstanceName` looks lik below:

```
{
 "id":"Server_localhost_8098"
 ,"simpleFields":{
 "HELIX_ENABLED":"true"
 ,"HELIX_HOST":"Server_localhost"
 ,"HELIX_PORT":"8098"
 }
 ,"listFields":{
 "TAG_LIST":["server_untagged"]
 }
 ,"mapFields":{
 }
}
```

And the znode `CONFIGS/PARTICIPANT/BrokerInstanceName` looks like below:

```
{
 "id":"Broker_localhost_8099"
 ,"simpleFields":{
 "HELIX_ENABLED":"true"
 ,"HELIX_HOST":"Broker_localhost"
 ,"HELIX_PORT":"8099"
 }
 ,"listFields":{
 "TAG_LIST":["broker_untagged"]
 }
 ,"mapFields":{
 }
}
```

7.1.13 Adding Resources to Cluster

There is one resource idealstate created for Broker by default called broker_resource. This will contain the broker_tenant to broker assignment. Before creation of first a data resource, here is the content of brokerResource IdealState

CLUSTERNAME/IDEALSTATES/BrokerResource (Broker IdealState before adding data resource)

```
{
 "id" : "brokerResource",
 "simpleFields" : {
 "IDEAL_STATE_MODE" : "CUSTOMIZED",
 "MAX_PARTITIONS_PER_INSTANCE" : "2147483647",
 "NUM_PARTITIONS" : "2147483647",
 "REBALANCE_MODE" : "CUSTOMIZED",
 "REPLICAS" : "2147483647",
 "STATE_MODEL_DEF_REF" : "BrokerResourceOnlineOfflineStateModel",
 "STATE_MODEL_FACTORY_NAME" : "DEFAULT"
 },
 "mapFields" : { },
 "listFields" : { }
}
```

After adding a resource using the following data resource creation command, a resource name XLNT will be created under IDEALSTATE znode. We will also tag one of server nodes as **server_XLNT** and 1 broker as **broker_XLNT.**

7.1.14 Sample Curl request

```
curl -i -X POST -H 'Content-Type: application/json' -d '{"requestType":"create",
↪"resourceName":"XLNT","tableName":"T1", "timeColumnName":"daysSinceEpoch", "timeType
↪":"daysSinceEpoch","numberOfDataInstances":4,"numberOfCopies":2,"retentionTimeUnit":
↪"DAYS", "retentionTimeValue":"700","pushFrequency":"daily", "brokerTagName":"XLNT",
↪"numberOfBrokerInstances":1, "segmentAssignmentStrategy":
↪"BalanceNumSegmentAssignmentStrategy", "resourceType":"OFFLINE", "metadata":{}}'
```

This is how it looks in Helix after running the above command.

The znode `CONFIGS/PARTICIPANT/Broker_localhost_8099` looks as follows:

```
{
 "id":"Broker_localhost_8099"
 ,"simpleFields":{
 "HELIX_ENABLED":"true"
 ,"HELIX_HOST":"Broker_localhost"
 ,"HELIX_PORT":"8099"
 }
 ,"listFields":{
 "TAG_LIST":["broker_mirrorProfileViewOfflineEvents1"]
 }
 ,"mapFields":{
 }
}
```

And the znode `IDEALSTATES/brokerResource` looks like below after Data resource is created

```
{
 "id":"brokerResource"
 ,"simpleFields":{
 "IDEAL_STATE_MODE":"CUSTOMIZED"
 ,"MAX_PARTITIONS_PER_INSTANCE":"2147483647"
 ,"NUM_PARTITIONS":"2147483647"
 ,"REBALANCE_MODE":"CUSTOMIZED"
 ,"REPLICAS":"2147483647"
 ,"STATE_MODEL_DEF_REF":"BrokerResourceOnlineOfflineStateModel"
 ,"STATE_MODEL_FACTORY_NAME":"DEFAULT"
 }
 ,"listFields":{
 }
 ,"mapFields":{
 "mirrorProfileViewOfflineEvents1_O":{
 "Broker_localhost_8099":"ONLINE"
 }
 }
}
```

Server Info in Helix

The znode `CONFIGS/PARTICIPANT/Server_localhost_8098` looks as below

```
{
 "id":"Server_localhost_8098"
 ,"simpleFields":{
 "HELIX_ENABLED":"true"
 ,"HELIX_HOST":"Server_localhost"
```

(continues on next page)

(continued from previous page)

```
 ,"HELIX_PORT":"8098"
 }
 ,"listFields":{
 "TAG_LIST":["XLNT"]
 }
 ,"mapFields":{
 }
}
```

And the znode `/IDEALSTATES/XLNT (XLNT Data Resource IdealState)` looks as below:

```
{
 "id":"XLNT"
 ,"simpleFields":{
 "IDEAL_STATE_MODE":"CUSTOMIZED"
 ,"INSTANCE_GROUP_TAG":"XLNT"
 ,"MAX_PARTITIONS_PER_INSTANCE":"1"
 ,"NUM_PARTITIONS":"0"
 ,"REBALANCE_MODE":"CUSTOMIZED"
 ,"REPLICAS":"1"
 ,"STATE_MODEL_DEF_REF":"SegmentOnlineOfflineStateModel"
 ,"STATE_MODEL_FACTORY_NAME":"DEFAULT"
 }
 ,"listFields":{}
 ,"mapFields":{ }
}
```

7.1.15 Adding tables to Resources

Once the resource is created, we can create tables and upload segments accordingly.

7.1.16 Add a table to data resource

Sample Curl request

```
curl -i -X PUT -H 'Content-Type: application/json' -d '{"requestType":
↪"addTableToResource","resourceName":"XLNT","tableName":"T1", "resourceType":"OFFLINE
↪", "metadata":{}}' <span class="nolink">[http://CONTROLLER-HOST:PORT/
↪dataresources](http://CONTROLLER-HOST:PORT/dataresources)
```

After the table is added, mapping between Resources and Tables are maintained in Helix Property Store (This is a place holder in Zookeeper provided by Helix to store application specific attributes).

The znode `/PROPERTYSTORE/CONFIGS/RESOURCE/XLNT` like like:

```
{
 "id":"mirrorProfileViewOfflineEvents1_O"
 ,"simpleFields":{
 "brokerTagName":"broker_mirrorProfileViewOfflineEvents1"
 ,"numberOfBrokerInstances":"1"
 ,"numberOfCopies":"1"
 ,"numberOfDataInstances":"1"
 ,"pushFrequency":"daily"
 ,"resourceName":"mirrorProfileViewOfflineEvents1"
```

(continues on next page)

(continued from previous page)

```
 ,"resourceType":"OFFLINE"
 ,"retentionTimeUnit":"DAYS"
 ,"retentionTimeValue":"300"
 ,"segmentAssignmentStrategy":"BalanceNumSegmentAssignmentStrategy"
 ,"timeColumnName":"daysSinceEpoch"
 ,"timeType":"DAYS"
 }
 ,"listFields":{
 "tableName":["T1"]
 }
 ,"mapFields":{
 "metadata":{
 }
 }
}
//This will change slightly when retention properties
//are stored at table scope </pre>
```

The znode `/IDEALSTATES/XLNT (XLNT Data Resource IdealState)`

```
{
 "id":"XLNT_O"
 ,"simpleFields":{
 "IDEAL_STATE_MODE":"CUSTOMIZED"
 ,"INSTANCE_GROUP_TAG":"XLNT_O"
 ,"MAX_PARTITIONS_PER_INSTANCE":"1"
 ,"NUM_PARTITIONS":"3"
 ,"REBALANCE_MODE":"CUSTOMIZED"
 ,"REPLICAS":"1"
 ,"STATE_MODEL_DEF_REF":"SegmentOnlineOfflineStateModel"
 ,"STATE_MODEL_FACTORY_NAME":"DEFAULT"
 }
 ,"listFields":{
 }
 ,"mapFields":{
 "XLNT_T1_daily_2014-08-01_2014-08-01_0":{
 "Server_localhost_8098":"ONLINE"
 }
 ,"XLNT_T1_daily_2014-08-01_2014-08-01_1":{
 "Server_localhost_8098":"ONLINE"
 }
 ,"XLNT_T1_daily_2014-08-01_2014-08-01_2":{
 "Server_localhost_8098":"ONLINE"
 }
 }
}
```

For all other admin operations, take a look at this [wiki](https://iwww.corp.linkedin.com/wiki/cf/display/PRT/Deployment+Tutorials).