

---

# **DynamoDB-mock Documentation**

***Release 1.0.0***

**Jean-Tiare Le Bigot**

**Sep 18, 2017**



---

## Contents

---

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	What is ddbmock <i>not</i> useful for ?	1
1.2	What is ddbmock useful for ?	1
1.3	History	2
<b>2</b>	<b>Documentation</b>	<b>3</b>
2.1	User guide	3
2.1.1	Getting started with DynamoDB-Mock	3
2.1.2	Current Status	6
2.1.3	Planifications with ddbmock	9
2.1.4	Extending DynamoDB-mock	10
2.1.5	Change log - Migration guide.	13
2.2	Database API	18
2.2.1	DynamoDB class	18
2.2.2	Item class	20
2.2.3	ItemSize class	22
2.2.4	Key class	23
2.2.5	Table class	24
2.3	Indices and tables	28
<b>3</b>	<b>Contribute</b>	<b>29</b>



# CHAPTER 1

---

## Overview

---

**DynamoDB** is a minimalistic NoSQL engine provided by Amazon as a part of their AWS product.

**DynamoDB** allows you to store documents composed of unicode, number or binary data as well as sets. Each table must define a `hash_key` and may define a `range_key`. All other fields are optional.

**DynamoDB** is really awesome but is terribly sloooooow with management tasks. This makes it completely unusable in test environments.

**ddbmock** brings a nice, tiny, in-memory or sqlite implementation of DynamoDB along with much better and detailed error messages. Among its niceties, it features a double entry point:

- regular network based entry-point with 1:1 correspondance with stock DynamoDB
- **embedded entry-point** with seamless boto integration 1, ideal to avoid spinning yet another server.

**ddbmock** is **not** intended for production use. It **will lose** your data. you've been warned! I currently recommend the "boto extension" mode for unit-tests and the "server" mode for functional tests.

## What is ddbmock *not* useful for ?

Do *not* use it in production or as a cheap DynamoDB replacement. I'll never stress it enough.

All the focus was on simplicity/hackability and simulation quality. Nothing else.

## What is ddbmock useful for ?

- FAST and RELIABLE unit testing
- FAST and RELIABLE functional testing
- experiment with DynamoDB API.
- RELIABLE throughput planification

- RELIABLE disk space planification
- almost any DynamoDB simulation !

ddbmock can also persist your data in SQLITE. This open another vast range of possibilities :)

## History

- v1.0.0 (\*): full documentation and bugfixes
- v0.4.1: schema persistence + thread safety, bugfixes
- v0.4.0: sqlite backend + throughput statistics + refactoring, more documentation, more tests
- v0.3.2: batchWriteItem support + pass boto integration tests
- v0.3.1: accuracy in item/table sizes + full test coverage
- v0.3.0: first public release. Full table lifecycle + most items operations

(?) indicates a future release. These are only ideas or “nice to have”.

## User guide

### Getting started with DynamoDB-Mock

**DynamoDB** is a minimalistic NoSQL engine provided by Amazon as a part of their AWS product.

**DynamoDB** allows you to store documents composed of unicode, number or binary data as well as sets. Each table must define a `hash_key` and may define a `range_key`. All other fields are optional.

**DynamoDB** is really awesome but is terribly sloooooow with management tasks. This makes it completely unusable in test environments.

**ddbmock** brings a nice, tiny, in-memory or sqlite implementation of DynamoDB along with much better and detailed error messages. Among its niceties, it features a double entry point:

- regular network based entry-point with 1:1 correspondance with stock DynamoDB
- **embedded entry-point** with seamless boto integration 1, ideal to avoid spinning yet another server.

**ddbmock** is **not** intended for production use. It **will lose** your data. you've been warned! I currently recommend the "boto extension" mode for unit-tests and the "server" mode for functional tests.

### Installation

```
$ pip install ddbmock
```

### Example usage

#### Run as Regular client-server

Ideal for test environment. For stage and production I highly recommend using DynamoDB servers. **ddbmock** comes with no warranty and **will lose your data(tm)**.

Launch the server

```
$ pserve development.ini # launch the server on 0.0.0.0:6543
```

Start the client

```
import boto
from ddbmock import connect_boto_network

# Use the provided helper to connect your *own* endpoint
db = connect_boto_network()

# Done ! just use it wherever in your project as usual.
db.list_tables() # get list of tables (empty at this stage)
```

Note: if you do not want to import ddbmock only for the helper, here is a reference implementation:

```
def connect_boto_network(host='localhost', port=6543):
    import boto
    from boto.regioninfo import RegionInfo
    endpoint = '{}:{}'.format(host, port)
    region = RegionInfo(name='ddbmock', endpoint=endpoint)
    return boto.connect_dynamodb(region=region, port=port, is_secure=False)
```

### Run as a standalone library

Ideal for unit testing or small scale automated functional tests. Nice to play around with boto DynamoDB API too :)

```
import boto
from ddbmock import connect_boto_patch

# Wire-up boto and ddbmock together
db = connect_boto_patch()

# Done ! just use it wherever in your project as usual.
db.list_tables() # get list of tables (empty at this stage)
```

Note, to clean patches made in `boto.dynamodb.layer1`, you can call `clean_boto_patch()` from the same module.

### Using ddbmock for tests

Most tests share the same structure:

1. Set the things up
2. Test and validate
3. Clean everything up and start again

If you use ddbmock as a standalone library (which I recommend for this purpose), feel free to access any of the public methods in the `database` and `table` to perform direct checks

Here is a template taken from `GetItem` functional test using Boto.



```

# -*- coding: utf-8 -*-

import unittest
import boto

TABLE_NAME = 'Table-HR'
TABLE_RT = 45
TABLE_WT = 123
TABLE_HK_NAME = u'hash_key'
TABLE_HK_TYPE = u'N'
TABLE_RK_NAME = u'range_key'
TABLE_RK_TYPE = u'S'

HK_VALUE = u'123'
RK_VALUE = u'Decode this data if you are a coder'

ITEM = {
    TABLE_HK_NAME: {TABLE_HK_TYPE: HK_VALUE},
    TABLE_RK_NAME: {TABLE_RK_TYPE: RK_VALUE},
    u'relevant_data': {u'B': u'THVkaWEgaXMgdGhlIGJlc3QgY29tcGFueSBldmVyIQ=='},
}

class TestGetItem(unittest.TestCase):
    def setUp(self):
        from ddbmock import connect_boto_patch
        from ddbmock.database.db import dynamodb
        from ddbmock.database.table import Table
        from ddbmock.database.key import PrimaryKey

        # Do a full database wipe
        dynamodb.hard_reset()

        # Instanciate the keys
        hash_key = PrimaryKey(TABLE_HK_NAME, TABLE_HK_TYPE)
        range_key = PrimaryKey(TABLE_RK_NAME, TABLE_RK_TYPE)

        # Create a test table and register it in ``self`` so that you can use it_
        ↪directly
        self.t1 = Table(TABLE_NAME, TABLE_RT, TABLE_WT, hash_key, range_key)

        # Very important: register the table in the DB
        dynamodb.data[TABLE_NAME] = self.t1

        # Unconditionally add some data, for example.
        self.t1.put(ITEM, {})

        # Create the database connection ie: patch boto
        self.db = connect_boto_patch()

    def tearDown(self):
        from ddbmock.database.db import dynamodb
        from ddbmock import clean_boto_patch

        # Do a full database wipe
        dynamodb.hard_reset()

        # Remove the patch from Boto code (if any)

```

```
clean_boto_patch()

def test_get_hr(self):
    from ddbmock.database.db import dynamodb

    # Example test
    expected = {
        u'ConsumedCapacityUnits': 0.5,
        u'Item': ITEM,
    }

    key = {
        u"HashKeyElement": {TABLE_HK_TYPE: HK_VALUE},
        u"RangeKeyElement": {TABLE_RK_TYPE: RK_VALUE},
    }

    # Example check
    self.assertEqual(expected, self.db.layer1.get_item(TABLE_NAME, key))
```

If ddbmock is used as a standalone server, restarting it should do the job, unless SQLite persistence is used.

## Advanced usage

A significant part of ddbmock is now configurable through `ddbmock.config` parameters. This includes the storage backend.

By default, ddbmock has no persistence and stores everything in-memory. Alternatively, you can use the SQLite storage engine but be warned that it will be slower. To switch the backend, you will need to change a configuration variable *before* creating the first table.

```
from ddbmock import config

# switch to sqlite backend
config.STORAGE_ENGINE_NAME = 'sqlite'
# define the database path. defaults to 'dynamo.db'
config.STORAGE_SQLITE_FILE = '/tmp/my_database.sqlite'
```

Please note that ddbmock does not persist table metadata currently. As a consequence, you will need to create the tables at each restart even with the SQLite backend. This is hoped to be improved in future releases.

See <https://bitbucket.org/Ludiah/dynamodb-mock/src/tip/ddbmock/config.py> for a full list of parameters.

## Current Status

This document reflects ddbmock status as of 5/11/2012. It may be outdated.

Some items are marked as “WONTFIX”. These are throttling related. The goal of ddbmock is to help you with tests and planification. It won’t get in your way.

## Methods support

- CreateTable DONE
- DeleteTable DONE
- UpdateTable DONE

- DescribeTable DONE
- GetItem DONE
- PutItem DONE
- DeleteItem DONE
- UpdateItem ALMOST
- BatchGetItem DONE
- BatchWriteItem DONE
- Query DONE
- Scan DONE

All “Bulk” actions will handle the whole batch in a single pass, unless instructed to otherwise through `limit` parameter. Beware that real dynamoDB will most likely split bigger one. If you rely on high level libraries such as Boto, don’t worry about this.

`UpdateItem` has a different behavior when the target item did not exist prior the update operation. In particular, the `ADD` operator will always behave as though the item existed before.

## Comparison Operators

Some comparison might not work as expected on binary data as it is performed on the base64 representation instead of the binary one. Please report a bug if this is a problem for you, or, even better, open a pull request :)

All operators exists as lower case functions in `ddbmock.database.comparison`. This list can easily be extended to add new/custom operators.

## Common to Query and Scan

- EQ DONE
- LE DONE
- LT DONE
- GE DONE
- GT DONE
- BEGINS\_WITH DONE
- BETWEEN DONE

## Specific to Scan

- NULL DONE
- NOT\_NULL DONE
- CONTAINS DONE
- NOT\_CONTAINS DONE
- IN DONE

---

**Note:** `IN` operator is the only that can not be imported directly as it overlaps with builtin `in` keyword. If you need it, either import it with `getattr` on the module or as `in_test` which, anyway, is its internal name.

---

### Return value specifications

- `NONE` `DONE`
- `ALL_OLD` `DONE`
- `ALL_NEW` `DONE`
- `UPDATED_OLD` `DONE`
- `UPDATED_NEW` `DONE`

---

**Note:** Only `UpdateItem` recognize them all. Others does only the 2 first

---

### Rates and size limitations

#### Request rate

- Throttle read operations when provisioned throughput exceeded. `WONTFIX`
- Throttle write operations when provisioned throughput exceeded. `WONTFIX`
- Throughput usage logging for planification purpose. `DONE`
- Maximum throughput is 10,000. `DONE`
- Minimum throughput is 1. `DONE`
- Report accurate throughput. `DONE`

#### Request size

- Limit response size to 1MB. `TODO`
- Limit request size to 1MB. `TODO`
- Limit `BatchGetItem` to 100 per request. `TODO`
- Limit `BatchWriteItem` to 25 per request. `TODO`

### Table managment

- No more than 256 tables. `DONE`
- No more than 10 `CREATING` tables. `WONTFIX`
- No more than 10 `DELETING` tables. `WONTFIX`
- No more than 10 `UPDATING` tables. `WONTFIX`
- No more than 1 Throughput decrease/calendar day. `DONE`

- No more than \*2 Throughput increase/update. DONE

## Types and items Limitations

- Table names can only be between 3 and 255 bytes long. DONE
- Table names can only contains a-z, A-Z, 0-9, '\_', '-', and '.'. DONE
- No more than 64kB/Item including fieldname but not indexing overhead. DONE
- Primary key names can only be between 1 and 255 bytes long. DONE
- Attribute value can *not* be Null. DONE
- `hash_key` value maximum 2048 bytes. DONE
- `range_key` value maximum 1024 bytes. DONE
- Numbers max 38 digits precision; between  $10^{-128}$  and  $10^{+126}$ . DONE

## Table description

- item count. DONE
- data size. DONE
- date: creation. DONE
- date: last throughput increase. DONE
- date: last throughput decrease. DONE

Dates are represented as float timestamps using scientific notation by DynamoDB but we only send them as plain number, not caring about the representation. Most parsers won't spot any difference anyway.

## Planifications with ddbmock

DynamoDB-Mock has two main goals. Speeding up tests and helping you plan your real DynamoDB usage. This includes both the throughput and the disk usage.

### Getting disk usage

To get per table disk usage, feedback, one can issue a call to `DescribeTable` method. the informations returned are accurate in the sense of DynamoDB but beware, these are also by far *below* the real usage in ddbmock as there are absolutely no optimisations done on our side.

### Getting Throughput usage

To get per table throughput usage you can rely on the dedicated logger `utils.tp_logger`. By default, `min`, `max` and `average` throughput are logged every 5 minutes and at the end of the program via an atexit handler.

Note that the handler is hooked to `NullHandler` handler by default so that there should not be any noise in the console.

To get statistics more often, you can change `config.STAT_TP_AGGREG` value **before** issuing any requests to ddbmock. `__init__` may be a good place to do so.

For example, if you want to get statistics to the console every 15 seconds, you can use a code like this :

```
from ddbmock import config
from ddbmock.utils import tp_logger
import logging

config.STAT_TP_AGGREG = 15                # every 15 sec
tp_logger.addHandler(logging.StreamHandler()) # to console
```

Depending on how your application scales, it may be interesting to run a representative scenario with multiple users and see how the throughput proportions. This will be a very valuable information when going live.

### General logging

Logger `utils.req_logger` traces request body, response and errors if applicable. Each log entry starts with `request_id=...`. This allows you to keep track of each individual request even in a highly concurrent environment.

By default, all is logged to `NullHandler` and you should at least hook a `logging.StreamHandler` to have a console output.

## Extending DynamoDB-mock

### Get the source Luke

```
$ hg clone ssh://hg@bitbucket.org/Ludia/dynamodb-mock
$ pip install nose nosetests coverage mock webtest boto
$ python setup.py develop
$ nosetests # --no-skip to run boto integration tests too
```

### Folder structure

```
DynamoDB-Mock
+-- ddbmock
|   +-- database      => request engine
|   |   `-- storage  => storage backend
|   +-- operations    => each DynamoDB operation has a route here
|   +-- router         => entry-points logic
|   `-- validators    => request syntax validation middleware
+-- docs
|   `-- pages
`-- tests
    +-- unit           => mainly details and corner cases tests
    `-- functional
        +-- boto       => main/extensive tests
        `-- pyramid    => just make sure that all methods are supported
```

### Request flow: the big picture

Just a couple of comments here:

- The `router` relies on introspection to find the validators (if any)
- The `router` relies on introspection to find the routes

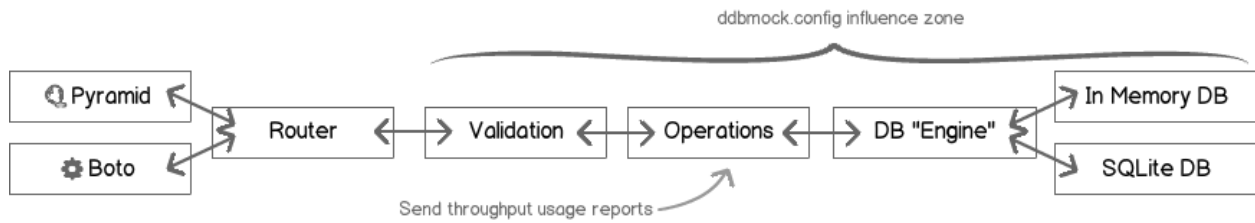


Fig. 2.1: Global request flow

- The database engine relies on introspection to find the configured storage backend
- There is a “catch all” in the router that maps to DynamoDB internal server error

### Adding a custom action

As long as the method follows DynamoDB request structure, it is mostly a matter of adding a file to `ddbmock/routes` with the following conventions:

- `file_name`: “underscore” version of the camel case method name.
- `function_name`: `file_name`.
- `argument`: parsed post payload.
- `return`: response dict.

Example: adding a `HelloWorld` method:

```
# -*- coding: utf-8 -*-
# module: ddbmock.routes.hello_world.py

def hello_world(post):
    return {
        'Hello': 'World'
    }
```

If the post of your method contains `TableName`, you may auto-load the corresponding table this way:

```
# -*- coding: utf-8 -*-
# module: ddbmock.routes.hello_world.py

from ddbmock.utils import load_table

@load_table
def hello_world(post):
    return {
        'Hello': 'World'
    }
```

### Adding a validator

Let’s say you want to let your new `HelloWorld` greet someone in particular, you will want to add an argument to the request.

Example: simplest way to add support for an argument:

```
# -*- coding: utf-8 -*-
# module: ddbmock.routes.hello_world.py

def hello_world(post):
    return {
        'Hello': 'World (and "{you}" too!)'.format(you=post['Name'])
    }
```

Wanna test it?

```
>>> curl -d '{"Name": "chuck"}' -H 'x-amz-target: DynamoDB_custom.HelloWorld' \
↪localhost:6543
{'Hello': 'World (and "chuck" too!)'}
```

But this is most likely to crash the server if ‘Name’ is not in Post. This is where Voluptuous comes.

In ddbmock, all you need to do to enable automatic validations is to add a file with the underscore name in ddbmock . validators. It must contain a post member with the rules.

Example: HelloWorld validator for HelloWorld method:

```
# -*- coding: utf-8 -*-
# module: ddbmock.validators.hello_world.py

post = {
    u'Name': unicode,
}
```

Done !

## Adding a storage backend

Storage backends lives in ‘ddbmock/database/storage’. There are currently two of them built-in. Basic “in-memory” (default) and “sqlite” to add persistence.

As for the methods, storage backends follow conventions to keep the code lean

- they must be in ddbmock . database . storage module
- they must implement Store class following this outline

```
# -*- coding: utf-8 -*-

# in case you need to load configuration constants
from ddbmock import config

# the name can *not* be changed.
class Store(object):
    def __init__(self, name):
        """ Initialize the in-memory store
        :param name: Table name.
        """
        # TODO

    def truncate(self):
        """Perform a full table cleanup. Might be a good idea in tests :)"""
        # TODO
```



```

def __getitem__(self, (hash_key, range_key)):
    """Get item at ``hash_key``, ``range_key`` or the dict at ``hash_key`` if
    ``range_key`` is None.

    :param key: (``hash_key``, ``range_key``) Tuple. If ``range_key`` is None,
    ↪all keys under ``hash_key`` are returned
    :return: Item or item dict

    :raise: KeyError
    """
    # TODO

def __setitem__(self, (hash_key, range_key), item):
    """Set the item at (``hash_key``, ``range_key``). Both keys must be
    defined and valid. By convention, ``range_key`` may be ``False`` to
    indicate a ``hash_key`` only key.

    :param key: (``hash_key``, ``range_key``) Tuple.
    :param item: the actual ``Item`` data structure to store
    """
    # TODO

def __delitem__(self, (hash_key, range_key)):
    """Delete item at key (``hash_key``, ``range_key``)

    :raises: KeyError if not found
    """
    # TODO

def __iter__(self):
    """ Iterate all over the table, abstracting the ``hash_key`` and
    ``range_key`` complexity. Mostly used for ``Scan`` implementation.
    """
    # TODO

```

As an example, I recommend to study “memory.py” implementation. It is pretty straight-forward and well commented. You get the whole package for only 63 lines :)

## Change log - Migration guide.

### ddbmock 1.0.2

This section documents all user visible changes included between ddbmock version 1.0.0 and version 1.0.2.

#### Fixes

- Fixed issues #10, #11, #13 and #15. Thanks to Lance Linder, Michael Hart and James O’Beirne for the pull requests.

### ddbmock 1.0.0

This section documents all user visible changes included between ddbmock versions 0.4.1 and versions 1.0.0

### Additions

- Add documentation for `Table` internal API
- Add documentation for `DynamoDB (database)` internal API
- Add documentation for `Key` internal API
- Add documentation for `Item` and `ItemSize` internal API

### Changes

- Add a `truncate` method to the tables

### ddbmock 0.4.1 aka 1.0.0 RC

This section documents all user visible changes included between ddbmock versions 0.4.0 and versions 0.4.1

This iteration was mostly focused on polishing and brings last missing bits.

### Additions

- Add support for `ExclusiveStartKey`, `LastEvaluatedKey` and `Limit` for `Scan`

### Changes

- Wrap all write operations in a table scope lock: each individual operation should be atomic
- Addressed Thread safety issues
- Add option to disable status update timers (#8)
- Fix `BETWEEN` bug (#7)

### ddbmock 0.4.0

This section documents all user visible changes included between ddbmock versions 0.3.2 and versions 0.4.0

This iteration was focused on modularity and planification.

### Additions

- `consistent_read` parameter to `Query`
- central `config.py` file with all constraints
- timer for table status changes
- full `Query` support
- throughput statistics to help plan real usage
- pre-instantiate `DynamoDB` as `dynamodb`
- **datastore API**

- bundle memory store
  - bundle sqlite store
  - add config param to switch
- `clean_boto_patch` to restore original `boto.dynamodb` behavior
- allow `ConsistentRead` on a per-table basis for `BatchGetItem`

## Removal

- legacy `connect_boto` and `connect_ddbmock`
- `dynamodb_api_validate` decorator. It is now called automatically
- `wrap_exceptions` decorator. It is now integrated to the router
- minimum throughput change of 10 %

## Changes

- global refactoring
- rename `routes` module to `operations` for consistency with DynamoDB
- Move from `Voluptuous` to `Onctuous` for validations, less code
- fix server startup with `pserver` (bad package name)
- fix server crash on validation exception (bad serialization)
- accurate throughput for all Read operations
- accurate throughput for all Write operations
- move ‘views’ to ‘routes’
- remove all pyramid code from ‘views’/‘routes’
- pyramid and boto entry points now shares most of the router
- `UpdateItem` failed to save keys properly
- integrate boto dynamodb integration tests to test suite (disabled unless ‘–no-skip’)
- do not require (real) user credentials in boto patch version (#5)

## Upgrade

- rename `connect_boto` to `connect_boto_patch`
- rename `connect_ddbmock` to `connect_boto_network`
- rename all `DynamoDB()` to ```dynamodb`
- replace `...import DynamoDB` by `... import dynamodb`

### ddbmock 0.3.2

This section documents all user visible changes included between ddbmock versions 0.3.1 and versions 0.3.2  
This iteration was focused on passing boto integration tests.

#### Additions

- preliminary batchWriteItem support

#### Changes

- fix number validation
- fix: item where created by defaultdict magic when looking for bogus item.
- return no Item field when not found, but empty when filtered
- [botopatch] handle DynamoDBConditionalCheckFailedError error

### ddbmock 0.3.1

This section documents all user visible changes included between ddbmock versions 0.3.0 and versions 0.3.1  
This iteration was focused on accuracy

#### Additions

- 100% tests coverage
- add basic tests for pyramid entry-point (#1)
- add plenty of unit and functional tests. Coverage is 100%
- add support for all ALL\_OLD ALL\_NEW UPDATED\_OLD UPDATED\_NEW in UpdateItem
- add accurate field size calculation
- add accurate item size calculation
- add accurate table size calculation
- add MAX\_TABLES check at table creation

#### Changes

- accurate table statuses
- fix pyramid entry-point
- fix list validations. Len limitation was not working
- attempt to store empty field/set raise ValidationError (#4)
- accurate exception detection and reporting in UpdateTable
- accurate hash\_key and range\_key size validation

- accurate number limitations (max 38 digits precision; between  $10^{-128}$  and  $10^{+126}$ )
- rename `connect_boto` to `connect_boto_patch` + compat layer
- rename `connect_ddbmock` to `connect_boto_network` + compat layer
- block `PutItem/UpdateItem` when bigger than `MAX_ITEM_SIZE`

## Upgrade

Nothing mandatory as this is a minor release but, I recommend that you:

- rename `connect_boto` to `connect_boto_patch`
- rename `connect_ddbmock` to `connect_boto_network`

## ddbmock 0.3.0

Initial ddbmock release. This is *alpha quality* software. Some import features such as “Exclusive Start Key”, “Reverse” and “Limit” as well as `BatchWriteItem` have not been written (yet).

## Additions

- entry-point WEB (network mode)
- entry-point Boto (standalone mode)
- support for `CreateTable` method
- support for `DeleteTable` method
- support for `UpdateTable` method
- support for `DescribeTable` method
- support for `GetItem` method
- support for `PutItem` method
- support for `DeleteItem` method
- support for `UpdateItem` method (small approximations)
- support for `BatchGetItem` method (initial)
- support for `Query` method (initial)
- support for `Scan` method (initial)
- all comparison operators
- aggressive input validation

## Known bugs - limitations

- no support for `BatchWriteItem`
- no support for “Exclusive Start Key”, “Reverse” and “Limit” in `Query` and `Scan`
- no support for “UnprocessedKeys” in `BatchGetItem`

- Web entry-point is untested, fill bugs if necessary :)

## Database API

Describe internal database structures. Should be extremely useful for tests.

### DynamoDB class

**class** `ddbmock.database.db.DynamoDB`

Main database, behaves as a singleton in the sense that all instances share the same data.

If underlying store supports it, all tables schema are persisted at creation time to a special table `~*schema*` which is an invalid DynamoDB table name so no collisions are to be expected.

### Constructors

#### `__init__`

`DynamoDB.__init__()`

When first instanciated, `__init__` checks the underlying store for potentially persisted tables. If any, it reloads there schema to make them available to the application.

In all other cases, `__init__` simply loads the shared state.

### Batch data manipulations

#### `get_batch`

`DynamoDB.get_batch(batch)`

Batch processor. Dispatches call to appropriate `ddbmock.database.table.Table` methods. This is the only low\_level API that directly pushes throughput usage.

**Parameters** `batch` – raw DynamoDB request batch.

**Returns** dict compatible with DynamoDB API

**Raises** `ddbmock.errors.ValidationException` if a `range_key` was provided while table has none.

**Raises** `ddbmock.errors.ResourceNotFoundException` if a table does not exist.

#### `write_batch`

`DynamoDB.write_batch(batch)`

Batch processor. Dispatches call to appropriate `ddbmock.database.table.Table` methods. This is the only low\_level API that directly pushes throughput usage.

**Parameters** `batch` – raw DynamoDB request batch.

**Returns** dict compatible with DynamoDB API

**Raises** `ddbmock.errors.ValidationException` if a `range_key` was provided while table has none.

**Raises** `ddbmock.errors.ResourceNotFoundException` if a table does not exist.

## Database management

### list\_tables

`DynamoDB.list_tables()`  
Get a list of all table names.

### create\_table

`DynamoDB.create_table(name, data)`  
Create a `ddbmock.database.table.Table` named 'name' using parameters provided in data if it does not yet exist.

#### Parameters

- **name** – Valid table name. No further checks are performed.
- **data** – raw DynamoDB request data.

**Returns** A reference to the newly created `ddbmock.database.table.Table`

**Raises** `ddbmock.errors.ResourceInUseException` if the table already exists.

**Raises** `ddbmock.errors.LimitExceededException` if more than `ddbmock.config.MAX_TABLES` already exist.

### delete\_table

`DynamoDB.delete_table(name)`  
Triggers internal “realistic” table deletion. This implies changing the status to `DELETING`. Once `:py:const:ddbmock.config.DELAY_DELETING` has expired `:py:meth:_internal_delete_table` is called and the table is de-referenced from `:py:attr:data`.

Since `:py:attr:data` only holds a reference, the table object might still exist at that moment and possibly still handle pending requests. This also allows to safely return a handle to the table object.

**Parameters** **name** – Valid table name.

**Returns** A reference to `ddbmock.database.table.Table` named name

### get\_table

`DynamoDB.get_table(name)`  
Get a handle to `ddbmock.database.table.Table` 'name' if it exists.

**Parameters** **name** – Name of the table to load.

**Returns** `ddbmock.database.table.Table` with name 'name'

**Raises** `ddbmock.errors.ResourceNotFoundException` if the table does not exist.

### hard\_reset

DynamoDB.**hard\_reset** ()

Reset and drop all tables. If any data was persisted, it will be completely lost after a call to this method. I do use in `tearDown` of all ddbmock tests to avoid any side effect.

### Item class

**class** ddbmock.database.item.**Item** (*dico*={})

Internal Item representation. The Item is stored in its raw DynamoDB request form and no parsing is involved unless specifically needed.

It adds a couple of handful helpers to the dict class such as DynamoDB actions, condition validations and specific size computation.

### Constructors

#### `__init__`

Item.**\_\_init\_\_** (*dico*={})

Load a raw DynamoDb Item and enhance it with ou helpers. Also set the cached *ItemSize* to None to mark it as not computed. This avoids unnecessary computations on temporary Items.

**Parameters** *dico* – Raw DynamoDB request Item

### Item manipulations

#### filter

Item.**filter** (*fields*)

Return a dict containing only the keys specified in *fields*. If *fields* evaluates to False (None, empty, ...), the original dict is returned untouched.

Internal *ItemSize* of the filtered Item is set to original Item size as you pay for the data you operated on, not for what was actually sent over the wires.

**Parameters** *fields* – array of name of keys to keep

**Returns** filtered Item

### apply\_actions

Item.**apply\_actions** (*actions*)

Apply actions to the current item. Mostly used by `UpdateItem`. This also resets the cached item size.

**Warning:** There is a corner case in `ADD` action. It will always behave as though the item already existed before that is to say, if the target field is a non existing set, it will always start a new one with this single value in it. In real DynamoDB, if Item was new, it should fail.

**Parameters** *action* – Raw DynamoDB request actions specification



**Raises** `ddbmock.errors.ValidationException` whenever attempting an illegal action

## `assert_match_expected`

`Item.assert_match_expected(expected)`

Make sure this Items matches the expected values. This may be used by any signe item write operation such as `DeleteItem`, `UpdateItem` and `PutItem`.

**Parameters** `expected` – Raw DynamoDB request expected values

**Raises** `ddbmock.errors.ConditionalCheckFailedException` if any of the expected values is not valid

## `match`

`Item.match(conditions)`

Check if the current item matches conditions. Return False if a field is not found, or does not match. If condition is None, it is considered to match.

Condition name are assumed to be valid as Onctuous is in charge of input validation. Expect crashes otherwise :)

### **Parameters**

- **fieldname** – Valid field name
- **condition** – Raw DynamoDB request condition of the form `{"OPERATOR": FIELDDEFINITION}`

**Returns** True on success or False on first failure

## `field_match`

`Item.field_match(fieldname, condition)`

Check if a field matches a condition. Return False when field not found, or do not match. If condition is None, it is considered to match.

Condition name are assumed to be valid as Onctuous is in charge of input validation. Expect crashes otherwise :)

### **Parameters**

- **fieldname** – Valid field name
- **condition** – Raw DynamoDB request condition of the form `{"OPERATOR": FIELDDEFINITION}`

**Returns** True on success

## `read_key`

`Item.read_key(key, name=None, max_size=0)`

Provided key, read field value at name or key.name if not specified.

### **Parameters**

- **key** – Key or PrimaryKey to read

- **name** – override name field of key
- **max\_size** – if specified, check that the item is below a threshold

**Returns** field value at key

**Raises** `ddbmock.errors.ValidationError` if field does not exist, type does not match or is above `max_size`

### get\_field\_size

`Item.get_field_size(fieldname)`

Compute field size in bytes.

**Parameters** **fieldname** – Valid field name

**Returns** Size of the field in bytes or 0 if the field was not found. Remember that empty fields are represented as missing values in DynamoDB.

### get\_size

`Item.get_size()`

Compute Item size as DynamoDB would. This is especially useful for enforcing the 64kb per item limit as well as the capacityUnit cost.

---

**Note:** the result is cached for efficiency. If you ever happen to directly edit values for any reason, do not forget to invalidate the cache: `self.size=None`

---

**Returns** *ItemSize* DynamoDB item size in bytes

### \_\_sub\_\_

`Item.__sub__(other)`

Utility function to compute a ‘diff’ of 2 Items. All fields of `self` (left operand) identical to those of `other` (right operand) are discarded. The other fields from `self` are kept. This proves to be extremely useful to support `ALL_NEW` and `ALL_OLD` return specification of `UpdateItem` in a clean and readable manner.

**Parameters** **other** – Item to be used as filter

**Returns** dict with fields of `self` not in or different from `other`

## ItemSize class

`class ddbmock.database.item.ItemSize`

Utility class to represent an *Item* size as bytes or capacity units

## ItemSize manipulations

## `__add__`

`ItemSize.__add__(value)`

Transparently allow addition of `ItemSize` values. This is useful for all batch requests as `Scan`, `Query`, `BatchWriteItem` and `BatchReadItem`

**Parameters** `value` – foreign int compatible value to add

**Returns** new `ItemSize` value

**Raises** `TypeError` if value is not int compatible

## `as_units`

`ItemSize.as_units()`

Get item size in terms of capacity units. This does *not* include the index overhead. Units can *not* be below 1 ie: a `DeleteItem` on a non existing item is *not* free

**Returns** number of capacity unit consumed by any operation on this `ItemSize`

## `with_indexing_overhead`

`ItemSize.with_indexing_overhead()`

Take the indexing overhead into account. this is especially usefull to compute the table disk size as DynamoDB would but it's not included in the capacity unit calculation.

**Returns** `ItemSize + ddbmock.config.INDEX_OVERHEAD`

## Key class

`class ddbmock.database.key.Key(name, typename)`

Abstraction layer over DynamoDB Keys in `ddbmock.database.item.Item`

## Constructors

### `__init__`

`Key.__init__(name, typename)`

High level Python constructor

**Parameters**

- **name** – Valid key name. No further checks are performed.
- **typename** – Valid key typename. No further checks are performed.

### `from_dict`

`classmethod Key.from_dict(data)`

Alternate constructor which deciphers raw DynamoDB request data before ultimately calling regular `__init__` method.

See `__init__()` for more insight.

**Parameters** `data` – raw DynamoDB request data.

**Returns** fully initialized `Key` instance

## Key manipulations

### read

`Key.read(key)`

Parse a key as specified by DynamoDB API and return its value as long as its typename matches typename

**Parameters** `key` – Raw DynamoDB request key.

**Returns** the value of the key

**Raises** `ddbmock.errors.ValidationException` if field types does not match

### to\_dict

`Key.to_dict()`

Return the key as a Python dict.

**Returns** Serialized version of the key definition metadata compatible with DynamoDB API syntax.

## PrimaryKey

`class ddbmock.database.key.PrimaryKey(name, typename)`

Special marker to provide distinction between regular Keys and PrimaryKey

## Table class

`class ddbmock.database.table.Table(name, rt, wt, hash_key, range_key, status='CREATING')`

Table abstraction. Actual `ddbmock.database.item.Item` are stored in store.

## Constructors

### `__init__`

`Table.__init__(name, rt, wt, hash_key, range_key, status='CREATING')`

Create a new Table. When manually creating a table, make sure you registered it in `ddbmock.database.db.DynamoDB` with a something like `dynamodb.data[name] = Table(name, "...")`.

Even though there are `DELAY_CREATING` seconds before the status is updated to `ACTIVE`, the table is immediately available. This is a slight difference with real DynamoDB to ease unit and functional tests.

#### Parameters

- **name** – Valid table name. No further checks are performed.
- **rt** – Provisioned read throughput.
- **wt** – Provisioned write throughput.
- **hash\_key** – `ddbmock.database.key.Key` instance describe the hash\_key

- **hash\_key** – `ddbmock.database.key.Key` instance describe the `range_key` or `None` if table has no `range_key`
- **status** – (optional) Valid initial table status. If Table needd to be avaible immediately, use `ACTIVE`, otherwise, leave default value.

---

**Note:** `rt` and `wt` are only used by `DescribeTable` and `UpdateTable`. No throttling is nor will ever be done.

---

## from\_dict

**classmethod** `Table.from_dict(data)`

Alternate constructor which decipheres raw DynamoDB request data before ultimately calling regular `__init__` method.

See `__init__()` for more insight.

**Parameters** `data` – raw DynamoDB request data.

**Returns** fully initialized `Table` instance

## Table manipulations

### truncate

`Table.truncate()`

Remove all Items from this table. This is like a reset. Might be very usefull in unit and functional tests.

### delete

`Table.delete()`

If the table was `ACTIVE`, update its state to `DELETING`. This is not a destructor, only a sate up-dater and the Table instance will still be valid afterward. In all othercases, raise `ddbmock.errors.ResourceInUseException`.

If you want to perform the full table delete cycle, please use `ddbmock.database.db.DynamoDB.delete_table()` instead

**Raises** `ddbmock.errors.ResourceInUseException` is the table was not in `Active` state

### activate

`Table.activate()`

Unconditionnaly set Table status to `ACTIVE`. This method is automatically called by the constructor once `DELAY_CREATING` is over.

### update\_throughput

`Table.update_throughput(rt, wt)`

Update table throughput. Same conditions and limitations as real DynamoDB applies:

- No more than 1 decrease operation per UTC day.
- No more than doubling throughput at once.
- Table must be in `ACTIVE` state.

Table status is then set to `UPDATING` until `DELAY_UPDATING` delay is over. Like real DynamoDB, the Table can still be used during this period

### Parameters

- **rt** – New read throughput
- **wt** – New write throughput

**Raises** `ddbmock.errors.ResourceInUseException` if table was not in `ACTIVE` state

**Raises** `ddbmock.errors.LimitExceededException` if the other above conditions are not met.

## get\_size

`Table.get_size()`

Compute the whole table size using the same rules as the real DynamoDB. Actual memory usage in `ddbmock` will be much higher due to dict and Python overhead.

---

**Note:** Real DynamoDB updates this result every 6 hours or so while this is an “on demand” call.

---

**Returns** cumulated size of all items following DynamoDB size computation.

## to\_dict

`Table.to_dict(verbose=True)`

Serialize this table to DynamoDB compatible format. Every fields are realistic, including the `TableSizeBytes` which relies on `get_size()`

Some DynamoDB requests only send a minimal version of Table metadata. to reproduce this behavior, just set `verbose` to `False`.

**Parameters** **verbose** – Set to `False` to skip table size computation.

**Returns** Serialized version of table metadata compatible with DynamoDB API syntax.

## Items manipulations

### delete\_item

`Table.delete_item(key, expected)`

Delete item at `key` from the database provided that it matches `expected` values.

This operation is atomic and blocks all other pending write operations.

### Parameters

- **key** – Raw DynamoDB request hash and range key dict.
- **expected** – Raw DynamoDB request conditions.

**Returns** deepcopy of `ddbmock.database.item.Item` as it was before deletion.

**Raises** `ddbmock.errors.ConditionalCheckFailedException` if conditions are not met.

## update\_item

Table.**update\_item**(*key*, *actions*, *expected*)

Apply actions to item at key provided that it matches expected.

This operation is atomic and blocks all other pending write operations.

### Parameters

- **key** – Raw DynamoDB request hash and range key dict.
- **actions** – Raw DynamoDB request actions.
- **expected** – Raw DynamoDB request conditions.

**Returns** both deepcopies of `ddbmock.database.item.Item` as it was (before, after) the update.

**Raises** `ddbmock.errors.ConditionalCheckFailedException` if conditions are not met.

**Raises** `ddbmock.errors.ValidationException` if actions attempted to modify the key or the resulting Item is bigger than `config.MAX_ITEM_SIZE`

## put

Table.**put**(*item*, *expected*)

Save *item* in the database provided that *expected* matches. Even though DynamoDB `UpdateItem` operation only supports returning `ALL_OLD` or `NONE`, this method returns both old and new values as the throughput, computed in the view, takes the maximum of both size into account.

This operation is atomic and blocks all other pending write operations.

### Parameters

- **item** – Raw DynamoDB request item.
- **expected** – Raw DynamoDB request conditions.

**Returns** both deepcopies of `ddbmock.database.item.Item` as it was (before, after) the update or empty item if not found.

**Raises** `ddbmock.errors.ConditionalCheckFailedException` if conditions are not met.

## get

Table.**get**(*key*, *fields*)

Get fields from `ddbmock.database.item.Item` at key.

### Parameters

- **key** – Raw DynamoDB request key.
- **fields** – Raw DynamoDB request array of field names to return. Empty to return all.

**Returns** reference to `ddbmock.database.item.Item` at key or None when not found

**Raises** `ddbmock.errors.ValidationException` if a `range_key` was provided while table has none.

## query

Table.**query** (*hash\_key, rk\_condition, fields, start, reverse, limit*)

Return fields of all items with provided `hash_key` whose `range_key` matches `rk_condition`.

### Parameters

- **hash\_key** – Raw DynamoDB request `hash_key`.
- **rk\_condition** – Raw DynamoDB request `range_key` condition.
- **fields** – Raw DynamoDB request array of field names to return. Empty to return all.
- **start** – Raw DynamoDB request key of the first item to scan. Empty array to indicate first item.
- **reverse** – Set to `True` to parse the range keys backward.
- **limit** – Maximum number of items to return in this batch. Set to 0 or less for no maximum.

**Returns** `Results(results, cumulated_size, last_key)`

**Raises** `ddbmock.errors.ValidationException` if `start['HashKeyElement']` is not `hash_key`

## scan

Table.**scan** (*scan\_conditions, fields, start, limit*)

Return fields of all items matching `scan_conditions`. No matter the `start` key, `scan` allways starts from teh beginning so that it might be quite slow.

### Parameters

- **scan\_conditions** – Raw DynamoDB request conditions.
- **fields** – Raw DynamoDB request array of field names to return. Empty to return all.
- **start** – Raw DynamoDB request key of the first item to scan. Empty array to indicate first item.
- **limit** – Maximum number of items to return in this batch. Set to 0 or less for no maximum.

**Returns** `Results(results, cumulated_size, last_key, scanned_count)`

## Indices and tables

- `genindex`
- `modindex`
- `search`



## CHAPTER 3

---

### Contribute

---

Want to contribute, report a but of request a feature ? The development goes on BitBucket:

- **Download:** <http://pypi.python.org/pypi/ddbmock>
- **Report bugs:** <https://bitbucket.org/Ludia/dynamodb-mock/issues>
- **Fork the code:** <https://bitbucket.org/Ludia/dynamodb-mock/overview>



## Symbols

`__add__()` (ddbmock.database.item.ItemSize method), 23  
`__init__()` (ddbmock.database.db.DynamoDB method), 18  
`__init__()` (ddbmock.database.item.Item method), 20  
`__init__()` (ddbmock.database.key.Key method), 23  
`__init__()` (ddbmock.database.table.Table method), 24  
`__sub__()` (ddbmock.database.item.Item method), 22

## A

`activate()` (ddbmock.database.table.Table method), 25  
`apply_actions()` (ddbmock.database.item.Item method), 20  
`as_units()` (ddbmock.database.item.ItemSize method), 23  
`assert_match_expected()` (ddbmock.database.item.Item method), 21

## C

`create_table()` (ddbmock.database.db.DynamoDB method), 19

## D

`delete()` (ddbmock.database.table.Table method), 25  
`delete_item()` (ddbmock.database.table.Table method), 26  
`delete_table()` (ddbmock.database.db.DynamoDB method), 19  
DynamoDB (class in ddbmock.database.db), 18

## F

`field_match()` (ddbmock.database.item.Item method), 21  
`filter()` (ddbmock.database.item.Item method), 20  
`from_dict()` (ddbmock.database.key.Key class method), 23  
`from_dict()` (ddbmock.database.table.Table class method), 25

## G

`get()` (ddbmock.database.table.Table method), 27

`get_batch()` (ddbmock.database.db.DynamoDB method), 18  
`get_field_size()` (ddbmock.database.item.Item method), 22  
`get_size()` (ddbmock.database.item.Item method), 22  
`get_size()` (ddbmock.database.table.Table method), 26  
`get_table()` (ddbmock.database.db.DynamoDB method), 19

## H

`hard_reset()` (ddbmock.database.db.DynamoDB method), 20

## I

Item (class in ddbmock.database.item), 20  
ItemSize (class in ddbmock.database.item), 22

## K

Key (class in ddbmock.database.key), 23

## L

`list_tables()` (ddbmock.database.db.DynamoDB method), 19

## M

`match()` (ddbmock.database.item.Item method), 21

## P

PrimaryKey (class in ddbmock.database.key), 24  
`put()` (ddbmock.database.table.Table method), 27

## Q

`query()` (ddbmock.database.table.Table method), 28

## R

`read()` (ddbmock.database.key.Key method), 24  
`read_key()` (ddbmock.database.item.Item method), 21

## S

`scan()` (`ddbmock.database.table.Table` method), [28](#)

## T

`Table` (class in `ddbmock.database.table`), [24](#)

`to_dict()` (`ddbmock.database.key.Key` method), [24](#)

`to_dict()` (`ddbmock.database.table.Table` method), [26](#)

`truncate()` (`ddbmock.database.table.Table` method), [25](#)

## U

`update_item()` (`ddbmock.database.table.Table` method), [27](#)

`update_throughput()` (`ddbmock.database.table.Table` method), [25](#)

## W

`with_indexing_overhead()` (`ddbmock.database.item.ItemSize` method), [23](#)

`write_batch()` (`ddbmock.database.db.DynamoDB` method), [18](#)