
DynamoDB-mock Documentation

Release 0.4.1

Jean-Tiare Le Bigot

November 16, 2012

CONTENTS

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.

1.1 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.

1.2 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 opens another vast range of possibilities :)

1.3 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”.

DOCUMENTATION

2.1 User guide

2.1.1 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 loose 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.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, {})

    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 import connect_boto_patch
        from ddbmock.database.db import dynamodb

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

        # 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, 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/Ludia/dynamodb-mock/src/tip/ddbmock/config.py> for a full list of parameters.

2.1.2 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 maximu 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.

2.1.3 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 multiples 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 requests 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.

2.1.4 Extending DynamoDB-mock

Get the source Luke

```
$ hg clone ssh://hg@bitbucket.org/Ludia/dynamodb-mock
$ pip install nose nosetests coverage mock webtests 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

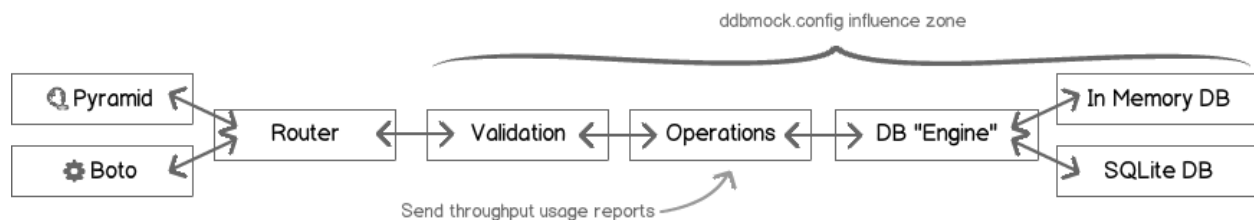


Figure 2.1: Global request flow

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

- 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'
        :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 :)

2.1.5 Change log - Migration guide.

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 backage 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 :)

2.2 Indices and tables

- *genindex*
- *modindex*
- *search*

CONTRIBUTE

Want to contribute, report a bug or 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>