PynamoDB Documentation

Release 3.1.0

Jharrod LaFon

Sep 18, 2017

Contents

PynamoDB is a Pythonic interface to Amazon's DynamoDB. By using simple, yet powerful abstractions over the DynamoDB API, PynamoDB allows you to start developing immediately.

CHAPTER 1

Features

- Python 3 support
- Python 2 support
- Support for Unicode, Binary, JSON, Number, Set, and UTC Datetime attributes
- Support for DynamoDB Local
- Support for all of the DynamoDB API
- Support for Global and Local Secondary Indexes
- Batch operations with automatic pagination
- Iterators for working with Query and Scan operations
- Fully tested

CHAPTER 2

Topics

Usage

PynamoDB was written from scratch to be Pythonic, and supports the entire DynamoDB API.

Creating a model

Let's create a simple model to describe users.

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute

class UserModel(Model):
    """
    A DynamoDB User
    """
    class Meta:
        table_name = 'dynamodb-user'
        region = 'us-west-1'
    email = UnicodeAttribute(hash_key=True)
    first_name = UnicodeAttribute()
    last_name = UnicodeAttribute()
```

Models are backed by DynamoDB tables. In this example, the model has a hash key attribute that stores the user's email address. Any attribute can be set as a hash key by including the argument *hash_key=True*. The *region* attribute is not required, and will default to *us-east-1* if not provided.

PynamoDB allows you to create the table:

```
>>> UserModel.create_table(read_capacity_units=1, write_capacity_units=1)
```

Now you can create a user:

```
>>> user = UserModel('test@example.com', first_name='Samuel', last_name='Adams')
dynamodb-user<test@example.com>
```

To write the user to DynamoDB, just call save:

```
>>> user.save()
```

You can see that the table count has changed:

```
>>> UserModel.count()
1
```

Attributes can be accessed and set normally:

```
>>> user.email
'test@example.com'
>>> user.email = 'foo-bar'
>>> user.email
'foo-bar
```

Did another process update the user? We can refresh the user with data from DynamoDB:

```
>>> user.refresh()
```

Ready to delete the user?

```
>>> user.delete()
```

Querying

PynamoDB provides an intuitive abstraction over the DynamoDB Query API. All of the Query API comparison operators are supported.

Suppose you had a table with both a hash key that is the user's last name and a range key that is the user's first name:

```
class UserModel(Model):
        """
        A DynamoDB User
        """
        class Meta:
            table_name = 'dynamodb-user'
        email = UnicodeAttribute()
        first_name = UnicodeAttribute(range_key=True)
        last_name = UnicodeAttribute(hash_key=True)
```

Now, suppose that you want to search the table for users with a last name 'Smith', and first name that begins with the letter 'J':

```
for user in UserModel.query('Smith', UserModel.first_name.startswith('J')):
    print(user.first_name)
```

You can combine query terms:

```
for user in UserModel.query('Smith', UserModel.first_name.startswith('J') | UserModel.
↪email.contains('domain.com')):
    print(user)
```

Counting Items

You can retrieve the count for queries by using the *count* method:

```
print(UserModel.count('Smith', UserModel.first_name.startswith('J'))
```

Counts also work for indexes:

```
print(UserModel.custom_index.count('my_hash_key'))
```

Alternatively, you can retrieve the table item count by calling the *count* method without filters:

```
print(UserModel.count())
```

Note that the first positional argument to *count()* is a *hash_key*. Although this argument can be *None*, filters must not be used when *hash_key* is *None*:

```
# raises a ValueError
print(UserModel.count(UserModel.first_name == 'John'))

# returns count of only the matching users
print(UserModel.count('my_hash_key', UserModel.first_name == 'John'))
```

Batch Operations

PynamoDB provides context managers for batch operations.

Note: DynamoDB limits batch write operations to 25 *PutRequests* and *DeleteRequests* combined. *PynamoDB* automatically groups your writes 25 at a time for you.

Let's create a whole bunch of users:

```
with UserModel.batch_write() as batch:
    for i in range(100):
        batch.save(UserModel('user-{0}@example.com'.format(i), first_name='Samuel',␣
↪last_name='Adams'))
```

Now, suppose you want to retrieve all those users:

```
user_keys = [('user-{0}@example.com'.format(i)) for i in range(100)]
for item in UserModel.batch_get(user_keys):
    print(item)
```

Perhaps you want to delete all these users:

```
with UserModel.batch_write() as batch:
    items = [UserModel('user-{0}@example.com'.format(x)) for x in range(100)]
    for item in items:
        batch.delete(item)
```

Basic Tutorial

PynamoDB is attempt to be a Pythonic interface to DynamoDB that supports all of DynamoDB's powerful features in *both* Python 3, and Python 2. This includes support for unicode and binary attributes.

But why stop there? PynamoDB also supports:

- Sets for Binary, Number, and Unicode attributes
- Automatic pagination for bulk operations
- Global secondary indexes
- Local secondary indexes
- Complex queries

Why PynamoDB?

It all started when I needed to use Global Secondary Indexes, a new and powerful feature of DynamoDB. I quickly realized that my go to library, dynamodb-mapper, didn't support them. In fact, it won't be supporting them anytime soon because dynamodb-mapper relies on another library, boto.dynamodb, which itself won't support them. In fact, boto doesn't support Python 3 either. If you want to know more, I blogged about it.

Installation

```
$ pip install pynamodb
```

Don't have pip? Here are instructions for installing pip..

Getting Started

PynamoDB provides three API levels, a `Connection`, a `TableConnection`, and a `Model`. Each API is built on top of the previous, and adds higher level features. Each API level is fully featured, and can be used directly. Before you begin, you should already have an Amazon Web Services account, and have your AWS credentials configured your boto.

Defining a Model

The most powerful feature of PynamoDB is the `Model` API. You start using it by defining a model class that inherits from `pynamodb.models.Model`. Then, you add attributes to the model that inherit from `pynamodb.attributes.Attribute`. The most common attributes have already been defined for you.

Here is an example, using the same table structure as shown in Amazon's DynamoDB Thread example.

Note: The table that your model represents must exist before you can use it. It can be created in this example by calling *Thread.create_table(...)*. Any other operation on a non existent table will cause a *TableDoesNotExist* exception to be raised.

```
from pynamodb.models import Model
from pynamodb.attributes import (
    UnicodeAttribute, NumberAttribute, UnicodeSetAttribute, UTCDateTimeAttribute
)


class Thread(Model):
    class Meta:
        table_name = 'Thread'

    forum_name = UnicodeAttribute(hash_key=True)
    subject = UnicodeAttribute(range_key=True)
    views = NumberAttribute(default=0)
    replies = NumberAttribute(default=0)
    answered = NumberAttribute(default=0)
    tags = UnicodeSetAttribute()
    last_post_datetime = UTCDateTimeAttribute()
```

All DynamoDB tables have a hash key, and you must specify which attribute is the hash key for each `Model` you define. The `forum_name` attribute in this example is specified as the hash key for this table with the `hash_key` argument; similarly the `subject` attribute is specified as the range key with the `range_key` argument.

Model Settings

The `Meta` class is required with at least the `table_name` class attribute to tell the model which DynamoDB table to use - `Meta` can be used to configure the model in other ways too. You can specify which DynamoDB region to use with the `region`, and the URL endpoint for DynamoDB can be specified using the `host` attribute. You can also specify the table's read and write capacity by adding `read_capacity_units` and `write_capacity_units` attributes.

Here is an example that specifies both the `host` and the `region` to use:

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute


class Thread(Model):
    class Meta:
        table_name = 'Thread'
        # Specifies the region
        region = 'us-west-1'
        # Specifies the hostname
        host = 'http://localhost'
        # Specifies the write capacity
        write_capacity_units = 10
        # Specifies the read capacity
        read_capacity_units = 10
    forum_name = UnicodeAttribute(hash_key=True)
```

Defining Model Attributes

A `Model` has attributes, which are mapped to attributes in DynamoDB. Attributes are responsible for serializing/deserializing values to a format that DynamoDB accepts, optionally specifying whether or not an attribute may be empty using the *null* argument, and optionally specifying a default value with the *default* argument. You can specify a default value for any field, and `default` can even be a function.

Note: DynamoDB will not store empty attributes. By default, an `Attribute` cannot be `None` unless you specify `null=True` in the attribute constructor.

DynamoDB attributes can't be null and set attributes can't be empty. PynamoDB attempts to do the right thing by pruning null attributes when serializing an item to be put into DynamoDB. By default, PynamoDB attributes can't be null either - but you can easily override that by adding `null=True` to the constructor of the attribute. When you make an attribute nullable, PynamoDB will omit that value if the value is `None` when saving to DynamoDB. It is not recommended to give every attribute a value if those values can represent null, as those values representing null take up space - which literally costs you money (DynamoDB pricing is based on reads and writes per second per KB). Instead, treat the absence of a value as equivalent to being null (which is what PynamoDB does). The only exception of course, are hash and range keys which must always have a value.

Here is an example of an attribute with a default value:

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute


class Thread(Model):
    class Meta:
        table_name = 'Thread'
    forum_name = UnicodeAttribute(hash_key=True, default='My Default Value')
```

Here is an example of an attribute with a default *callable* value:

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute

def my_default_value():
    return 'My default value'

class Thread(Model):
    class Meta:
        table_name = 'Thread'
    forum_name = UnicodeAttribute(hash_key=True, default=my_default_value)
```

Here is an example of an attribute that can be empty:

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute

class Thread(Model):
    class Meta:
        table_name = 'Thread'
    forum_name = UnicodeAttribute(hash_key=True)
    my_nullable_attribute = UnicodeAttribute(null=True)
```

By default, PynamoDB assumes that the attribute name used on a Model has the same name in DynamoDB. For example, if you define a *UnicodeAttribute* called 'username' then PynamoDB will use 'username' as the field name for that attribute when interacting with DynamoDB. If you wish to have custom attribute names, they can be overidden. One such use case is the ability to use human readable attribute names in PynamoDB that are stored in DynamoDB using shorter, terse attribute to save space.

Here is an example of customizing an attribute name:

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute

class Thread(Model):
    class Meta:
        table_name = 'Thread'
    forum_name = UnicodeAttribute(hash_key=True)
    # This attribute will be called 'tn' in DynamoDB
    thread_name = UnicodeAttribute(null=True, attr_name='tn')
```

PynamoDB comes with several built in attribute types for convenience, which include the following:

- *UnicodeAttribute*
- *UnicodeSetAttribute*
- *NumberAttribute*
- *NumberSetAttribute*
- *BinaryAttribute*
- *BinarySetAttribute*
- *UTCDateTimeAttribute*
- *BooleanAttribute*
- *JSONAttribute*

All of these built in attributes handle serializing and deserializng themselves, in both Python 2 and Python 3.

Creating the table

If your table doesn't already exist, you will have to create it. This can be done with easily:

```
>>> if not Thread.exists():
        Thread.create_table(read_capacity_units=1, write_capacity_units=1, wait=True)
```

The `wait` argument tells PynamoDB to wait until the table is ready for use before returning.

Deleting a table

Deleting is made quite simple when using a *Model*:

```
>>> Thread.delete_table()
```

Using the Model

Now that you've defined a model (referring to the example above), you can start interacting with your DynamoDB table. You can create a new *Thread* item by calling the *Thread* constructor.

Creating Items

```
>>> thread_item = Thread('forum_name', 'forum_subject')
```

The first two arguments are automatically assigned to the item's hash and range keys. You can specify attributes during construction as well:

```
>>> thread_item = Thread('forum_name', 'forum_subject', replies=10)
```

The item won't be added to your DynamoDB table until you call save:

```
>>> thread_item.save()
```

If you want to retrieve an item that already exists in your table, you can do that with *get*:

```
>>> thread_item = Thread.get('forum_name', 'forum_subject')
```

If the item doesn't exist, *Thread.DoesNotExist* will be raised.

Updating Items

You can update an item with the latest data from your table:

```
>>> thread_item.refresh()
```

Updates to table items are supported too, even atomic updates. Here is an example of atomically updating the view count of an item + updating the value of the last post.

```
>>> thread_item.update(actions=[
        Thread.views.set(Thread.views + 1),
        Thread.last_post_datetime.set(datetime.now()),
    ])
```

Deprecated since version 2.0: `update_item()` is replaced with `update()`

```
>>> thread_item.update_item('views', 1, action='add')
```

Index Queries

DynamoDB supports two types of indexes: global secondary indexes, and local secondary indexes. Indexes can make accessing your data more efficient, and should be used when appropriate. See the documentation for more information.

Index Settings

The `Meta` class is required with at least the `projection` class attribute to specify the projection type. For Global secondary indexes, the `read_capacity_units` and `write_capacity_units` also need to be provided. By default, PynamoDB will use the class attribute name that you provide on the model as the `index_name` used when making requests to the DynamoDB API. You can override the default name by providing the `index_name` class attribute in the `Meta` class of the index.

Global Secondary Indexes

Indexes are defined as classes, just like models. Here is a simple index class:

```
from pynamodb.indexes import GlobalSecondaryIndex, AllProjection
from pynamodb.attributes import NumberAttribute


class ViewIndex(GlobalSecondaryIndex):
    """
    This class represents a global secondary index
    """
    class Meta:
        # index_name is optional, but can be provided to override the default name
        index_name = 'foo-index'
        read_capacity_units = 2
        write_capacity_units = 1
        # All attributes are projected
        projection = AllProjection()

    # This attribute is the hash key for the index
    # Note that this attribute must also exist
    # in the model
    view = NumberAttribute(default=0, hash_key=True)
```

Global indexes require you to specify the read and write capacity, as we have done in this example. Indexes are said to *project* attributes from the main table into the index. As such, there are three styles of projection in DynamoDB, and PynamoDB provides three corresponding projection classes.

- *`AllProjection`*: All attributes are projected.
- *`KeysOnlyProjection`*: Only the index and primary keys are projected.
- *`IncludeProjection(attributes)`*: Only the specified `attributes` are projected.

We still need to attach the index to the model in order for us to use it. You define it as a class attribute on the model, as in this example:

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute


class TestModel(Model):
    """
    A test model that uses a global secondary index
    """
    class Meta:
        table_name = 'TestModel'
    forum = UnicodeAttribute(hash_key=True)
    thread = UnicodeAttribute(range_key=True)
    view_index = ViewIndex()
    view = NumberAttribute(default=0)
```

Local Secondary Indexes

Local secondary indexes are defined just like global ones, but they inherit from `LocalSecondaryIndex` instead:

```
from pynamodb.indexes import LocalSecondaryIndex, AllProjection
from pynamodb.attributes import NumberAttribute
```

```
class ViewIndex(LocalSecondaryIndex):
    """
    This class represents a local secondary index
    """
    class Meta:
        # All attributes are projected
        projection = AllProjection()
    forum = UnicodeAttribute(hash_key=True)
    view = NumberAttribute(range_key=True)
```

You must specify the same hash key on the local secondary index and the model. The range key can be any attribute.

Querying an index

Index queries use the same syntax as model queries. Continuing our example, we can query the `view_index` global secondary index simply by calling `query`:

```
for item in TestModel.view_index.query(1):
    print("Item queried from index: {0}".format(item))
```

This example queries items from the table using the global secondary index, called `view_index`, using a hash key value of 1 for the index. This would return all `TestModel` items that have a `view` attribute of value 1.

Local secondary index queries have a similar syntax. They require a hash key, and can include conditions on the range key of the index. Here is an example that queries the index for values of `view` greater than zero:

```
for item in TestModel.view_index.query('foo', TestModel.view > 0):
    print("Item queried from index: {0}".format(item.view))
```

Batch Operations

Batch operations are supported using context managers, and iterators. The DynamoDB API has limits for each batch operation that it supports, but PynamoDB removes the need implement your own grouping or pagination. Instead, it handles pagination for you automatically.

Note: DynamoDB limits batch write operations to 25 *PutRequests* and *DeleteRequests* combined. *PynamoDB* automatically groups your writes 25 at a time for you.

Suppose that you have defined a *Thread* Model for the examples below.

```
from pynamodb.models import Model
from pynamodb.attributes import (
    UnicodeAttribute, NumberAttribute
)


class Thread(Model):
    class Meta:
        table_name = 'Thread'

    forum_name = UnicodeAttribute(hash_key=True)
    subject = UnicodeAttribute(range_key=True)
    views = NumberAttribute(default=0)
```

Batch Writes

Here is an example using a context manager for a bulk write operation:

```
with Thread.batch_write() as batch:
    items = [TestModel('forum-{0}'.format(x), 'thread-{0}'.format(x)) for x in␣
↪range(1000)]
    for item in items:
        batch.save(item)
```

Batch Gets

Here is an example using an iterator for retrieving items in bulk:

```
item_keys = [('forum-{0}'.format(x), 'thread-{0}'.format(x)) for x in range(1000)]
for item in Thread.batch_get(item_keys):
    print(item)
```

Query Filters

You can query items from your table using a simple syntax:

```
for item in Thread.query('ForumName', Thread.subject.startswith('mygreatprefix')):
    print("Query returned item {0}".format(item))
```

Additionally, you can filter the results before they are returned using condition expressions:

```
for item in Thread.query('ForumName', Thread.subject == 'Subject', Thread.views > 0):
    print("Query returned item {0}".format(item))
```

Query filters use the condition expression syntax (see *Condition Expressions*).

Note: DynamoDB only allows the following conditions on range keys: ==, <, <=, >, >=, *between*, and *startswith*. DynamoDB does not allow multiple conditions using range keys.

Scan Filters

Scan filters have the same syntax as Query filters, but support all condition expressions:

```
>>> for item in Thread.scan(Thread.forum_name.startswith('Prefix') & (Thread.views >␣
↪10)):
        print(item)
```

Limiting results

Both *Scan* and *Query* results can be limited to a maximum number of items using the *limit* argument.

```
for item in Thread.query('ForumName', Thread.subject.startswith('mygreatprefix'),␣
↪limit=5):
    print("Query returned item {0}".format(item))
```

Conditional Operations

Some DynamoDB operations (UpdateItem, PutItem, DeleteItem) support the inclusion of conditions. The user can supply a condition to be evaluated by DynamoDB before the operation is performed. See the official documentation for more details.

Suppose that you have defined a *Thread* Model for the examples below.

```
from pynamodb.models import Model
from pynamodb.attributes import (
    UnicodeAttribute, NumberAttribute
)


class Thread(Model):
    class Meta:
        table_name = 'Thread'

    forum_name = UnicodeAttribute(hash_key=True)
    subject = UnicodeAttribute(range_key=True)
    views = NumberAttribute(default=0)
```

Condition Expressions

PynamoDB supports creating condition expressions from attributes using a mix of built-in operators and method calls. Any value provided will be serialized using the serializer defined for that attribute. See the comparison operator and function reference for more details.

DynamoDB Condition	PynamoDB Syntax	Example
=	==	Thread.forum_name == 'Some Forum'
<>	!=	Thread.forum_name != 'Some Forum'
<	<	Thread.views < 10
<=	<=	Thread.views <= 10
>	>	Thread.views > 10
>=	>=	Thread.views >= 10
BETWEEN	between(*lower* , *upper*)	Thread.views.between(1, 5)
IN	is_in(**values*)	Thread.subject.is_in('Subject', 'Other Subject')
attribute_exists (*path*)	exists()	Thread.forum_name.exists()
attribute_not_exists (*path*)	does_not_exist()	Thread.forum_name.does_not_exist()
attribute_type (*path* , *type*)	is_type()	Thread.forum_name.is_type()
begins_with (*path* , *substr*)	startswith(*prefix*)	Thread.subject.startswith('Example')
contains (*path* , *operand*)	contains(*item*)	Thread.subject.contains('foobar')
size (*path*)	size(*attribute*)	size(Thread.subject) == 10
AND	&	(Thread.views > 1) & (Thread.views < 5)
OR	|	(Thread.views < 1) | (Thread.views > 5)
NOT	~	~Thread.subject.contains('foobar')

If necessary, you can use document paths to access nested list and map attributes:

```
from pynamodb.expressions.condition import size

print(size('foo.bar[0].baz') == 0)
```

Conditional Model.save

This example saves a *Thread* item, only if the item exists.

```
thread_item = Thread('Existing Forum', 'Example Subject')

# DynamoDB will only save the item if forum_name exists
print(thread_item.save(Thread.forum_name.exists())

# You can specify multiple conditions
print(thread_item.save(Thread.forum_name.exists() & Thread.forum_subject.contains(
↪'foobar')))
```

Conditional Model.update

This example will update a *Thread* item, if the *views* attribute is less than 5 *OR* greater than 10:

```
thread_item.update(condition=(Thread.views < 5) | (Thread.views > 10))
```

Conditional Model.delete

This example will delete the item, only if its *views* attribute is equal to 0.

```
print(thread_item.delete(Thread.views == 0))
```

Conditional Operation Failures

You can check for conditional operation failures by inspecting the cause of the raised exception:

```
try:
    thread_item.save(Thread.forum_name.exists())
except PutError as e:
    if isinstance(e.cause, ClientError):
        code = e.cause.response['Error'].get('Code')
        print(code == "ConditionalCheckFailedException")
```

Custom Attributes

Attributes in PynamoDB are classes that are serialized to and from DynamoDB attributes. PynamoDB provides attribute classes for all DynamoDB data types, as defined in the DynamoDB documentation. Higher level attribute types (internally stored as a DynamoDB data types) can be defined with PynamoDB. Two such types are included with PynamoDB for convenience: `JSONAttribute` and `UnicodeDatetimeAttribute`.

Attribute Methods

All `Attribute` classes must define three methods, `serialize`, `deserialize` and `get_value`. The `serialize` method takes a Python value and converts it into a format that can be stored into DynamoDB. The `get_value` method reads the serialized value out of the DynamoDB record. This raw value is then passed to the `deserialize` method. The `deserialize` method then converts it back into its value in Python. Additionally, a class attribute called `attr_type` is required for PynamoDB to know which DynamoDB data type the attribute is stored as. The `get_value` method is provided to help when migrating from one attribute type to another, specifically with the `BooleanAttribute` type. If you're writing your own attribute and the `attr_type` has not changed you can simply use the base `Attribute` implementation of `get_value`.

Writing your own attribute

You can write your own attribute class which defines the necessary methods like this:

```
from pynamodb.attributes import Attribute
from pynamodb.constants import BINARY

class CustomAttribute(Attribute):
    """
    A custom model attribute
    """

    # This tells PynamoDB that the attribute is stored in DynamoDB as a binary
    # attribute
    attr_type = BINARY

    def serialize(value):
        # convert the value to binary and return it

    def deserialize(value):
        # convert the value from binary back into whatever type you require
```

Custom Attribute Example

The example below shows how to write a custom attribute that will pickle a customized class. The attribute itself is stored in DynamoDB as a binary attribute. The `pickle` module is used to serialize and deserialize the attribute. In this example, it is not necessary to define `attr_type` because the `PickleAttribute` class is inheriting from `BinaryAttribute` which has already defined it.

```
import pickle
from pynamodb.attributes import BinaryAttribute, UnicodeAttribute
from pynamodb.models import Model


class Color(object):
    """
    This class is used to demonstrate the PickleAttribute below
    """
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "<Color: {}>".format(self.name)
```

```
class PickleAttribute(BinaryAttribute):
    """
    This class will serializer/deserialize any picklable Python object.
    The value will be stored as a binary attribute in DynamoDB.
    """
    def serialize(self, value):
        """
        The super class takes the binary string returned from pickle.dumps
        and encodes it for storage in DynamoDB
        """
        return super(PickleAttribute, self).serialize(pickle.dumps(value))

    def deserialize(self, value):
        return pickle.loads(super(PickleAttribute, self).deserialize(value))


class CustomAttributeModel(Model):
    """
    A model with a custom attribute
    """
    class Meta:
        host = 'http://localhost:8000'
        table_name = 'custom_attr'
        read_capacity_units = 1
        write_capacity_units = 1

    id = UnicodeAttribute(hash_key=True)
    obj = PickleAttribute()
```

Now we can use our custom attribute to round trip any object that can be pickled.

```
>>>instance = CustomAttributeModel()
>>>instance.obj = Color('red')
>>>instance.id = 'red'
>>>instance.save()

>>>instance = CustomAttributeModel.get('red')
>>>print(instance.obj)
<Color: red>
```

List Attributes

DynamoDB list attributes are simply lists of other attributes. DynamoDB asserts no requirements about the types embedded within the list. Creating an untyped list is done like so:

```
from pynamodb.attributes import ListAttribute, NumberAttribute, UnicodeAttribute

class GroceryList(Model):
    class Meta:
        table_name = 'GroceryListModel'

    store_name = UnicodeAttribute(hash_key=True)
    groceries = ListAttribute()

# Example usage:
```

```
GroceryList(store_name='Haight Street Market',
            groceries=['bread', 1, 'butter', 6, 'milk', 1])
```

PynamoDB can provide type safety if it is required. Currently PynamoDB does not allow type checks on anything other than `MapAttribute` and subclasses of `MapAttribute`. We're working on adding more generic type checking in a future version. When defining your model use the `of=` kwarg and pass in a class. PynamoDB will check that all items in the list are of the type you require.

```
from pynamodb.attributes import ListAttribute, NumberAttribute


class OfficeEmployeeMap(MapAttribute):
    office_employee_id = NumberAttribute()
    person = UnicodeAttribute()


class Office(Model):
    class Meta:
        table_name = 'OfficeModel'
    office_id = NumberAttribute(hash_key=True)
    employees = ListAttribute(of=OfficeEmployeeMap)

# Example usage:

emp1 = OfficeEmployeeMap(
    office_employee_id=123,
    person='justin'
)
emp2 = OfficeEmployeeMap(
    office_employee_id=125,
    person='lita'
)
emp4 = OfficeEmployeeMap(
    office_employee_id=126,
    person='garrett'
)

Office(
    office_id=3,
    employees=[emp1, emp2, emp3]
).save()  # persists

Office(
    office_id=3,
    employees=['justin', 'lita', 'garrett']
).save()  # raises ValueError
```

Map Attributes

DynamoDB map attributes are objects embedded inside of top level models. See the examples here. When implementing your own MapAttribute you can simply extend `MapAttribute` and ignore writing serialization code. These attributes can then be used inside of Model classes just like any other attribute.

```
from pynamodb.attributes import MapAttribute, UnicodeAttribute
```

```
class CarInfoMap(MapAttribute):
    make = UnicodeAttribute(null=False)
    model = UnicodeAttribute(null=True)
```

Use PynamoDB Locally

Several DynamoDB compatible servers have been written for testing and debugging purposes. PynamoDB can be used with any server that provides the same API as DynamoDB.

PynamoDB has been tested with two DynamoDB compatible servers, DynamoDB Local and dynalite.

To use a local server, you need to set the `host` attribute on your `Model`'s `Meta` class to the hostname and port that your server is listening on.

Note: If you are using DynamoDB Local and also use `rate_limited_scan` on your models, you must also set `allow_rate_limited_scan_without_consumed_capacity` to `True` in the settings file (dynalite does not require this step because it implements returning of consumed capacity in responses, which is used by `rate_limited_scan`).

Note: Local implementations of DynamoDB such as DynamoDB Local or dynalite may not be fully featured (and I don't maintain either of those packages), so you may encounter errors or bugs with a local implementation that you would not encounter using DynamoDB.

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute


class Thread(Model):
    class Meta:
        table_name = "Thread"
        host = "http://localhost:8000"
    forum_name = UnicodeAttribute(hash_key=True)
```

Running dynalite

Make sure you have the Node Package Manager installed, instructions here.

Install dynalite:

```
$ npm install -g dynalite
```

Run dynalite:

```
$ dynalite --port 8000
```

That's it, you've got a DynamoDB compatible server running on port 8000.

Running DynamoDB Local

DynamoDB local is a tool provided by Amazon that mocks the DynamoDB API, and uses a local file to store your data. You can use DynamoDB local with PynamoDB for testing, debugging, or offline development. For more information, you can read Amazon's Announcement and Jeff Barr's blog post about it.

- Download the latest version of DynamoDB local here.
- Unpack the contents of the archive into a directory of your choice.

DynamoDB local requires the Java Runtime Environment version 7. Make sure the JRE is installed before continuing.

From the directory where you unpacked DynamoDB local, you can launch it like this:

```
$ java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar
```

Once the server has started, you should see output:

```
$ java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar
2014-03-28 12:09:10.892:INFO:oejs.Server:jetty-8.1.12.v20130726
2014-03-28 12:09:10.943:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.
↪0.0.0:8000
```

Now DynamoDB local is running locally, listening on port 8000 by default.

Table Backups

PynamoDB provides methods for backing up and restoring the items in your table. Items are serialized to and from JSON encoded strings and files. Only serialized item data are stored in a backup, not any table metadata.

Backing up a table

To back up a table, you can simply use the provided *dump* method and write the contents to a file.

```
from pynamodb.models import Model
from pynamodb.attributes import (
    UnicodeAttribute, NumberAttribute
)

class Thread(Model):
    class Meta:
        table_name = 'Thread'

    forum_name = UnicodeAttribute(hash_key=True)
    subject = UnicodeAttribute(range_key=True)
    views = NumberAttribute(default=0)

Thread.dump("thread_backup.json")
```

Alternatively, you can write the contents to a string.

```
content = Thread.dumps()
```

Restoring from a backup

To restore items from a backup file, simply use the provided *load* method.

> **Warning:** Items contained in a backup *will* overwrite any existing items in your table!

```
from pynamodb.models import Model
from pynamodb.attributes import (
    UnicodeAttribute, NumberAttribute
)

class Thread(Model):
    class Meta:
        table_name = 'Thread'

    forum_name = UnicodeAttribute(hash_key=True)
    subject = UnicodeAttribute(range_key=True)
    views = NumberAttribute(default=0)

Thread.load("thread_backup.json")
```

Alternatively, you can also load the contents from a string.

```
Thread.loads(content)
```

Signals

Starting with PynamoDB 3.1.0, there is support for signalling. This support is provided by the blinker library, which is not installed by default. In order to ensure blinker is installed, specify your PynamoDB requirement like so:

```
pynamodb[signals]==<YOUR VERSION NUMBER>
```

Signals allow certain senders to notify subscribers that something happened. PynamoDB currently sends signals before and after every DynamoDB API call.

Note: It is recommended to avoid business logic in signal callbacks, as this can have performance implications. To reinforce this, only the operation name and table name are available in the signal callback.

Subscribing to Signals

PynamoDB fires two signal calls, *pre_dynamodb_send* before the network call and *post_dynamodb_send* after the network call to DynamoDB.

The callback must taking the following arguments:

Arguments	Description
sender	The object that fired that method.
operation_name	The string name of the **'DynamoDB action'_**
table_name	The name of the table the operation is called upon.
req_uuid	A unique identifer so subscribers can correlate the before and after events.

To subscribe to a signal, the user needs to import the signal object and connect your callback, like so.

```
from pynamodb.signals import pre_dynamodb_send, post_dynamodb_send

def record_pre_dynamodb_send(sender, operation_name, table_name, req_uuid):
    pre_recorded.append((operation_name, table_name, req_uuid))

def record_post_dynamodb_send(sender, operation_name, table_name, req_uuid):
    post_recorded.append((operation_name, table_name, req_uuid))

pre_dynamodb_send.connect(record_pre_dynamodb_send)
post_dynamodb_send.connect(record_post_dynamodb_send)
```

PynamoDB Examples

An directory of examples is available with the PynamoDB source on GitHub. The examples are configured to use *http://localhost:8000* as the DynamoDB endpoint. For information on how to run DynamoDB locally, see : *Use PynamoDB Locally*.

Note: You should read the examples before executing them. They are configured to use `http://localhost:8000` by default, so that you can run them without actually consuming DynamoDB resources on AWS, and therefore not costing you any money.

Install PynamoDB

Although you can install & run PynamoDB from GitHub, it's best to use a released version from PyPI:

```
$ pip install pynamodb
```

Getting the examples

You can clone the PynamoDB repository to get the examples:

```
$ git clone https://github.com/pynamodb/PynamoDB.git
```

Running the examples

Go into the examples directory:

```
$ cd pynamodb/examples
```

Configuring the examples

Each example is configured to use `http://localhost:8000` as the DynamoDB endpoint. You'll need to edit an example and either remove the `host` setting (causing PynamoDB to use a default), or specify your own.

Running an example

Each example file can be executed as a script by a Python interpreter:

```
$ python model.py
```

Settings

Settings reference

Here is a complete list of settings which control default PynamoDB behavior.

request_timeout_seconds

Default: `60`

The default timeout for HTTP requests in seconds.

max_retry_attempts

Default: `3`

The number of times to retry certain failed DynamoDB API calls. The most common cases eligible for retries include `ProvisionedThroughputExceededException` and `5xx` errors.

base_backoff_ms

Default: `25`

The base number of milliseconds used for exponential backoff and jitter on retries.

region

Default: `"us-east-1"`

The default AWS region to connect to.

session_cls

Default: `botocore.vendored.requests.Session`

A class which implements the Session interface from requests, used for making API requests to DynamoDB.

allow_rate_limited_scan_without_consumed_capacity

Default: `False`

If `True`, `rate_limited_scan()` will proceed silently (without rate limiting) if the DynamoDB server does not return consumed capacity information in responses.

Overriding settings

Default settings may be overridden by providing a Python module which exports the desired new values. Set the `PYNAMODB_CONFIG` environment variable to an absolute path to this module or write it to `/etc/pynamodb/global_default_settings.py` to have it automatically discovered.

See an example of specifying a custom `session_cls` to configure the connection pool below.

```
from botocore.vendored import requests
from botocore.vendored.requests import adapters

class CustomPynamoSession(requests.Session):
    super(CustomPynamoSession, self).__init__()
    self.mount('http://', adapters.HTTPAdapter(pool_maxsize=100))

session_cls = CustomPynamoSession
```

Low Level API

PynamoDB was designed with high level features in mind, but includes a fully featured low level API. Any operation can be performed with the low level API, and the higher level PynamoDB features were all written on top of it.

Creating a connection

Creating a connection is simple:

```
from pynamodb.connection import Connection

conn = Connection()
```

You can specify a different DynamoDB url:

```
conn = Connection(host='http://alternative-domain/')
```

By default, PynamoDB will connect to the us-east-1 region, but you can specify a different one.

```
conn = Connection(region='us-west-1')
```

Modifying tables

You can easily list tables:

```
>>> conn.list_tables()
{u'TableNames': [u'Thread']}
```

or delete a table:

```
>>> conn.delete_table('Thread')
```

If you want to change the capacity of a table, that can be done as well:

```
>>> conn.update_table('Thread', read_capacity_units=20, write_capacity_units=20)
```

You can create tables as well, although the syntax is verbose. You should really use the model API instead, but here is a low level example to demonstrate the point:

```
kwargs = {
    'write_capacity_units': 1,
    'read_capacity_units': 1
    'attribute_definitions': [
        {
            'attribute_type': 'S',
            'attribute_name': 'key1'
        },
        {
            'attribute_type': 'S',
            'attribute_name': 'key2'
        }
    ],
    'key_schema': [
        {
            'key_type': 'HASH',
            'attribute_name': 'key1'
        },
        {
            'key_type': 'RANGE',
            'attribute_name': 'key2'
        }
    ]
}
conn.create_table('table_name', **kwargs)
```

You can also use *update_table* to change the Provisioned Throughput capacity of Global Secondary Indexes:

```
>>> kwargs = {
    'global_secondary_index_updates': [
        {
            'index_name': 'index_name',
            'read_capacity_units': 10,
            'write_capacity_units': 10
        }
    ]
}
>>> conn.update_table('table_name', **kwargs)
```

Modifying items

The low level API can perform item operationst too, such as getting an item:

```
conn.get_item('table_name', 'hash_key', 'range_key')
```

You can put items as well, specifying the keys and any other attributes:

```
conn.put_item('table_name', 'hash_key', 'range_key', attributes={'key': 'value'})
```

Deleting an item has similar syntax:

```
conn.delete_item('table_name', 'hash_key', 'range_key')
```

AWS Access

PynamoDB uses botocore to interact with the DynamoDB API. Thus, any method of configuration supported by `botocore` works with PynamoDB. For local development the use of environment variables such as *AWS_ACCESS_KEY_ID* and *AWS_SECRET_ACCESS_KEY* is probably preferable. You can of course use IAM users, as recommended by AWS. In addition EC2 roles will work as well and would be recommended when running on EC2.

As for the permissions granted via IAM, many tasks can be carried out by PynamoDB. So you should construct your policies as required, see the DynamoDB docs for more information.

Finally, see the AWS CLI documentation for more details on how to pass credentials to botocore.

Logging

Logging in PynamoDB uses the standard Python logging facilities. PynamoDB is built on top of `botocore` which also uses standard Python logging facilities. Logging is quite verbose, so you may only wish to enable it for debugging purposes.

Here is an example showing how to enable logging for PynamoDB:

```
from __future__ import print_function
import logging
from pynamodb.models import Model
from pynamodb.attributes import (
    UnicodeAttribute, NumberAttribute
)

logging.basicConfig()
log = logging.getLogger("pynamodb")
log.setLevel(logging.DEBUG)
log.propagate = True

class Thread(Model):
    class Meta:
        table_name = 'Thread'

    forum_name = UnicodeAttribute(hash_key=True)
    subject = UnicodeAttribute(range_key=True)
    views = NumberAttribute(default=0)

# Scan
for item in Thread.scan():
    print(item)

# Scan
for item in Thread.rate_limited_scan():
    print(item)
```

Contributing

Pull requests are welcome, forking from the `master` branch. If you are new to GitHub, be sure and check out GitHub's Hello World tutorial.

Make sure that your contribution meets the following requirements: * Be thoroughly tested * Works on all supported versions of Python * Be in the same code style of the existing source code (mostly PEP8)

Testing

The PynamoDB source code is thoroughly tested, which helps ensure that with each change made to it, we aren't breaking someone's code that relies on PynamoDB. It's not easy, and it's not optional. Changes without proper testing won't be accepted.

Please write tests to accompany your changes, and verify that the tests pass using all supported version of Python by using `tox`:

```
$ tox
```

Once you've opened a pull request on GitHub, Travis-ci will run the test suite as well.

Don't forget to add yourself to AUTHORS.rst.

Release Notes

v3.1.0

date 2017-07-07

This is a backwards compatible, minor release.

Note that we now require `botocore>=1.2.0`; this is required to support the `consistent_read` parameter when scanning.

Calling `Model.count()` without a `hash_key` and *with* `filters` will raise a `ValueError`, as it was previously returning incorrect results.

New features in this release:

- Add support for signals via blinker (#278)

Fixes in this release:

- Pass batch parameters down to boto/dynamo (#308)
- Raise a ValueError if count() is invoked with no hash key AND filters (#313)
- Add consistent_read parameter to Model.scan (#311)

Contributors to this release:

- @jmphilli
- @Lordnibbler
- @lita

v3.0.1

date 2017-06-09

This is a major release with breaking changes.

`MapAttribute` now allows pythonic access when recursively defined. If you were not using the `attr_name=` kwarg then you should have no problems upgrading. Previously defined non subclassed `MapAttributes` (raw `MapAttributes`) that were members of a subclassed `MapAttribute` (typed `MapAttributes`) would have to be accessed like a dictionary. Now object access is possible and recommended. See [here](https://github.com/pynamodb/PynamoDB/blob/master/pynamodb/tests/test_attributes.py#L671) for a test example. Access via the `attr_name`, also known as the DynamoDB name, will now throw an `AttributeError`.

`UnicodeSetAttributes` do not json serialize or deserialize anymore. We deprecated the functionality of json serializing as of `1.6.0` but left the deserialization functionality in there so people could migrate away from the old functionality. If you have any `UnicodeSetAttributes` that have not been persisted since version `1.6.0` you will need to migrate your data or manage the json encoding and decoding with a custom attribute in application.

- Performance enhancements for the `UTCDateTimeAttribute` deserialize method. (#277)
- There was a regression with attribute discovery. Fixes attribute discovery for model classes with inheritance (#280)
- Fix to ignore null checks for batch delete (#283)
- Fix for `ListAttribute` and `MapAttribute` serialize (#286)
- Fix for `MapAttribute` pythonic access (#292) This is a breaking change.
- Deprecated the json decode in `UnicodeSetAttribute` (#294) This is a breaking change.
- Raise `TableDoesNotExist` error instead of letting json decoding `ValueErrors` raise (#296)

Contributors to this release:

- @jcbertin
- @johnliu
- @scode
- @rowilla
- @lita
- @garretheel
- @jmphilli

v2.1.6

date 2017-05-10

This is a backwards compatible, minor release.

Fixes in this release:

- Replace Delorean with dateutil (#208)
- Fix a bug with count – consume all pages in paginated response (#256)
- Update mock lib (#262)
- Use pytest instead of nose (#263)
- Documentation changes (#269)
- Fix null deserialization in MapAttributes (#272)

Contributors to this release:

- @funkybob
- @garrettheel
- @lita
- @jmphilli

v2.1.5

date 2017-03-16

This is a backwards compatible, minor release.

Fixes in this release:

- Apply retry to ProvisionedThroughputExceeded (#222)
- rate_limited_scan fix to handle consumed capacity (#235)
- Fix for test when dict ordering differs (#237)

Contributors to this release:

- @anandswaminathan
- @jasonfriedland
- @JohnEmhoff

v2.1.4

date 2017-02-14

This is a minor release, with some changes to *MapAttribute* handling. Previously, when accessing a *MapAttribute* via *item.attr*, the type of the object used during instantiation would determine the return value. *Model(attr={...})* would return a *dict* on access. *Model(attr=MapAttribute(...))* would return an instance of *MapAttribute*. After #223, a *MapAttribute* will always be returned during item access regardless of the type of the object used during instantiation. For convenience, a *dict* version can be accessed using *.as_dict()* on the *MapAttribute*.

New features in this release:

- Support multiple attribute update (#194)
- Rate-limited scan (#205)
- Always create map attributes when setting a dict (#223)

Fixes in this release:

- Remove AttributeDict and require explicit attr names (#220)
- Add distinct DoesNotExist classes per model (#206)
- Ensure defaults are respected for MapAttribute (#221)
- Add docs for GSI throughput changes (#224)

Contributors to this release:

- @anandswaminathan
- @garrettheel
- @ikonst

- @jasonfriedland
- @yedpodtrzitko

v2.0.3

date 2016-11-18

This is a backwards compatible, minor release.

Fixes in this release:

- Allow longs as members of maps + lists in python 2 (#200)
- Allow raw map attributes in subclassed map attributes (#199)

Contributors to this release:

- @jmphilli

v2.0.2

date 2016-11-10

This is a backwards compatible, minor release.

Fixes in this release:

- add BOOL into SHORT_ATTR_TYPES (#190)
- deserialize map attributes correctly (#192)
- prepare request with requests session so session properties are applied (#197)

Contributors to this release:

- @anandswaminathan
- @jmphilli
- @yedpodtrzitko

v2.0.1

date 2016-11-04

This is a backwards compatible, minor release.

Fixes in this release:

- make “unprocessed keys for batch operation” log at info level (#180)
- fix RuntimeWarning during imp_load in custom settings file (#185)
- allow unstructured map attributes (#186)

Contributors to this release:

- @danielhochman
- @jmphilli
- @bedge

v2.0.0

date 2016-11-01

This is a major release, which introduces support for native DynamoDB maps and lists. There are no changes which are expected to break backwards compatibility, but you should test extensively before upgrading in production due to the volume of changes.

New features in this release:

- Add support for native map and list attributes (#175)

Contributors to this release:

- @jmphilli
- @berdim99

v1.6.0

date 2016-10-20

This is a minor release, with some changes to BinaryAttribute handling and new options for configuration.

BooleanAttribute now uses the native API type "B". BooleanAttribute is also compatible with the legacy BooleanAttributes on read. On save, they will be rewritten with the native type. If you wish to avoid this behavior, you can continue to use LegacyBooleanAttribute. LegacyBooleanAttribute is also forward compatible with native boolean attributes to allow for migration.

New features in this release:

- Add support for native boolean attributes (#149)
- Parse legacy and native bool in legacy bool (#158)
- Allow override of settings from global configuration file (#147)

Fixes in this release:

- Serialize UnicodeSetAttributes correctly (#151)
- Make update_item respect attr_name differences (#160)

Contributors to this release:

- @anandswaminathan
- @jmphilli
- @lita

v1.5.3

date 2016-08-08

This is a backwards compatible, minor release.

Fixes in this release:

- Introduce concept of page_size, separate from num items returned limit (#139)

Contributors to this release:

- @anandswaminathan

v1.5.2

date 2016-06-23

This is a backwards compatible, minor release.

Fixes in this release:

- Additional retry logic for HTTP Status Code 5xx, usually attributed to InternalServerError (#135)

Contributors to this release:

- @danielhochman

v1.5.1

date 2016-05-11

This is a backwards compatible, minor release.

Fixes in this release:

- Fix for binary attribute handling of unprocessed items data corruption affecting users of 1.5.0 (#126 fixes #125)

Contributors to this release:

- @danielhochman

v1.5.0

date 2016-05-09

This is a backwards compatible, minor release.

Please consider the fix for limits before upgrading. Correcting for off-by-one when querying is no longer necessary.

Fixes in this release:

- Fix off-by-one error for limits when querying (#123 fixed #95)
- Retry on ConnectionErrors and other types of RequestExceptions (#121 fixes #98)
- More verbose logging when receiving errors e.g. InternalServerError from the DynamoDB API (#115)
- Prevent permanent poisoning of credential cache due to botocore bug (#113 fixes #99)
- Fix for UnprocessedItems serialization error (#114 fixes #103)
- Fix parsing issue with newer version of dateutil and UTCDateTimeAttributes (#110 fixes #109)
- Correctly handle expected value generation for set types (#107 fixes #102)
- Use HTTP proxies configured by botocore (#100 fixes #92)

New features in this release:

- Return the cause of connection exceptions to the caller (#108 documented by #112)
- Configurable session class for custom connection pool size, etc (#91)
- Add attributes_to_get and consistent_read to more of the API (#79)

Contributors to this release:

- @ab

- @danielhochman
- @jlafon
- @joshowen
- @jpinner-lyft
- @mxr
- @nickgravgaard

v1.4.4

date 2015-11-10

This is a backward compatible, minor release.

Changes in this release:

- Support for enabling table streams at table creation time (thanks to @brln)
- Fixed bug where a value was always required for update_item when action was 'delete' (#90)

v1.4.3

date 2015-10-12

This is a backward compatible, minor release. Included are bug fixes and performance improvements.

A huge thank you to all who contributed to this release:

- Daniel Hochman
- Josh Owen
- Keith Mitchell
- Kevin Wilson

Changes in this release:

- Fixed bug where models without a range key weren't handled correctly
- Botocore is now only used for preparing requests (for performance reasons)
- Removed the dependency on OrderedDict
- Fixed bug for zope interface compatibility (#71)
- Fixed bug where the range key was handled incorrectly for integer values

v1.4.2

date 2015-06-26

This is a backward compatible, minor bug fix release.

Bugs fixed in this release:

- Fixed bug where botocore exceptions were not being reraised.

v1.4.1

date 2015-06-26

This is a backward compatible, minor bug fix release.

Bugs fixed in this release:

- Fixed bug where a local variable could be unbound (#67).

v1.4.0

date 2015-06-23

This is a minor release, with backward compatible bug fixes.

Bugs fixed in this release:

- Added support for botocore 1.0.0 (#63)
- Fixed bug where Model.get() could fail in certain cases (#64)
- Fixed bug where JSON strings weren't being encoded properly (#61)

v1.3.7

date 2015-04-06

This is a backward compatible, minor bug fix release.

Bugs fixed in this release:

- Fixed bug where range keys were not included in update_item (#59)
- Fixed documentation bug (#58)

v1.3.6

date 2015-04-06

This is a backward compatible, minor bug fix release.

Bugs fixed in this release:

- Fixed bug where arguments were used incorrectly in update_item (#54)
- Fixed bug where falsy values were used incorrectly in model constructors (#57), thanks @pior
- Fixed bug where the limit argument for scan and query was not always honored.

New features:

- Table counts with optional filters can now be queried using `Model.count(**filters)`

v1.3.5

This is a backward compatible, minor bug fix release.

Bugs fixed in this release.

- Fixed bug where scan did not properly limit results (#45)

- Fixed bug where scan filters were not being preserved (#44)
- Fixed bug where items were mutated as an unexpected side effect (#47)
- Fixed bug where conditional operator wasn't used in scan

v1.3.4

date 2014-10-06

This is a backward compatible, minor bug fix release.

Bugs fixed in this release.

- Fixed bug where attributes could not be used in multiple indexes when creating a table.
- Fixed bug where a dependency on mock was accidentally introduced.

v1.3.3

date 2014-9-18

This is a backward compatible, minor bug fix release, fixing the following issues

- Fixed bug with Python 2.6 compatibility (#28)
- Fixed bug where update_item was incorrectly checking attributes for null (#34)

Other minor improvements

- New API for backing up and restoring tables
- Better support for custom attributes (https://github.com/pynamodb/PynamoDB/commit/0c2ba5894a532ed14b6c14e5059e97dbb653ff12)
- Explicit Travis CI testing of Python 2.6, 2.7, 3.3, 3.4, and PyPy
- Tests added for round tripping unicode values

v1.3.2

date 2014-7-02

- This is a minor bug fix release, fixing a bug where query filters were incorrectly parsed (#26).

v1.3.1

date 2014-05-26

- This is a bug fix release, ensuring that KeyCondition and QueryFilter arguments are constructed correctly (#25).
- Added an example URL shortener to the examples.
- Minor documentation fixes.

v1.3.0

date 2014-05-20

- This is a minor release, with new backward compatible features and bug fixes.
- Fixed bug where NULL and NOT_NULL were not set properly in query and scan operations (#24)
- Support for specifying the index_name as a Index.Meta attribute (#23)
- Support for specifying read and write capacity in Model.Meta (#22)

v1.2.2

date 2014-05-14

- This is a minor bug fix release, resolving #21 (key_schema ordering for create_table).

v1.2.1

date 2014-05-07

- This is a minor bug fix release, resolving #20.

v1.2.0

date 2014-05-06

- Numerous documentation improvements
- Improved support for conditional operations
- Added support for filtering queries on non key attributes (http://aws.amazon.com/blogs/aws/improved-queries-and-updates-for-dynamodb/)
- Fixed issue with JSON loading where escaped characters caused an error (#17)
- Minor bug fixes

v1.1.0

date 2014-04-14

- PynamoDB now requires botocore version 0.42.0 or greater
- Improved documentation
- Minor bug fixes
- New API endpoint for deleting model tables
- Support for expected value conditions in item delete, update, and save
- Support for limit argument to queries
- Support for aliased attribute names

Example of using aliased attribute names:

```
class AliasedModel(Model):
    class Meta:
        table_name = "AliasedModel"
    forum_name = UnicodeAttribute(hash_key=True, attr_name='fn')
    subject = UnicodeAttribute(range_key=True, attr_name='s')
```

v1.0.0

date 2014-03-28

- Major update: New syntax for specifying models that is not backward compatible.

Important: The syntax for models has changed!

The old way:

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute


class Thread(Model):
    table_name = 'Thread'
    forum_name = UnicodeAttribute(hash_key=True)
```

The new way:

```
from pynamodb.models import Model
from pynamodb.attributes import UnicodeAttribute


class Thread(Model):
    class Meta:
        table_name = 'Thread'
    forum_name = UnicodeAttribute(hash_key=True)
```

Other, less important changes:

- Added explicit support for specifying the server hostname in models
- Added documentation for using DynamoDB Local and dynalite
- Made examples runnable with DynamoDB Local and dynalite by default
- Added documentation for the use of `default` and `null` on model attributes
- Improved testing for index queries

v0.1.13

date 2014-03-20

- Bug fix release. Proper handling of update_item attributes for atomic item updates, with tests. Fixes #7.

v0.1.12

date 2014-03-18

- Added a region attribute to model classes, allowing users to specify the AWS region, per model. Fixes #6.

v0.1.11

date 2014-02-26

- New exception behavior: Model.get and Model.refresh will now raise DoesNotExist if the item is not found in the table.
- Correctly deserialize complex key types. Fixes #3
- Correctly construct keys for tables that don't have both a hash key and a range key in batch get operations. Fixes #5
- Better PEP8 Compliance
- More tests
- Removed session and endpoint caching to avoid using stale IAM role credentials

Versioning Scheme

PynamoDB uses Semantic Versioning, where the version number has the format:

`MAJOR.MINOR.PATCH`

- The `MAJOR` version number changes when backward *incompatible* changes are introduced.
- The `MINOR` version number changes when new features are added, but are backward compatible.
- The `PATCH` version number changes when backward compatible bug fixes are added.

CHAPTER 3

API docs

API

High Level API

DynamoDB Models for PynamoDB

class pynamodb.models.**Model**(*hash_key=None*, *range_key=None*, ***attributes*)
Defines a *PynamoDB* Model

This model is backed by a table in DynamoDB. You can create the table by with the `create_table` method.

exception **DoesNotExist**(*msg=None*, *cause=None*)
Raised when an item queried does not exist

classmethod Model.**add_throttle_record**(*records*)
(Experimental) Pulls out the table name and capacity units from *records* and puts it in *self.throttle*

Parameters **records** – A list of usage records

classmethod Model.**batch_get**(*items*, *consistent_read=None*, *attributes_to_get=None*)
BatchGetItem for this model

Parameters **items** – Should be a list of hash keys to retrieve, or a list of tuples if range keys are used.

classmethod Model.**batch_write**(*auto_commit=True*)
Returns a context manager for a batch operation'

Parameters **auto_commit** – Commits writes automatically if *True*

classmethod Model.**count**(*hash_key=None*, *range_key_condition=None*, *filter_condition=None*, *consistent_read=False*, *index_name=None*, *limit=None*, ***filters*)
Provides a filtered count

Parameters

- **hash_key** – The hash key to query. Can be None.

- **range_key_condition** – Condition for range key
- **filter_condition** – Condition used to restrict the query results
- **consistent_read** – If True, a consistent read is performed
- **index_name** – If set, then this index is used
- **filters** – A dictionary of filters to be used in the query. Requires a hash_key to be passed.

classmethod Model.**create_table**(*wait=False*, *read_capacity_units=None*, *write_capacity_units=None*)
Create the table for this model

Parameters

- **wait** – If set, then this call will block until the table is ready for use
- **read_capacity_units** – Sets the read capacity units for this table
- **write_capacity_units** – Sets the write capacity units for this table

Model.**delete**(*condition=None*, *conditional_operator=None*, ***expected_values*)
Deletes this object from dynamodb

classmethod Model.**delete_table**()
Delete the table for this model

classmethod Model.**describe_table**()
Returns the result of a DescribeTable operation on this model's table

classmethod Model.**dump**(*filename*)
Writes the contents of this model's table as JSON to the given filename

classmethod Model.**dumps**()
Returns a JSON representation of this model's table

classmethod Model.**exists**()
Returns True if this table exists, False otherwise

classmethod Model.**from_raw_data**(*data*)
Returns an instance of this class from the raw data

Parameters **data** – A serialized DynamoDB object

classmethod Model.**get**(*hash_key*, *range_key=None*, *consistent_read=False*)
Returns a single object using the provided keys

Parameters

- **hash_key** – The hash key of the desired item
- **range_key** – The range key of the desired item, only used when appropriate.

classmethod Model.**get_throttle**()
Returns the throttle implementation for this Model

classmethod Model.**query**(*hash_key*, *range_key_condition=None*, *filter_condition=None*, *consistent_read=False*, *index_name=None*, *scan_index_forward=None*, *conditional_operator=None*, *limit=None*, *last_evaluated_key=None*, *attributes_to_get=None*, *page_size=None*, ***filters*)
Provides a high level query API

Parameters

- **hash_key** – The hash key to query

- **range_key_condition** – Condition for range key
- **filter_condition** – Condition used to restrict the query results
- **consistent_read** – If True, a consistent read is performed
- **index_name** – If set, then this index is used
- **limit** – Used to limit the number of results returned
- **scan_index_forward** – If set, then used to specify the same parameter to the DynamoDB API. Controls descending or ascending results
- **conditional_operator** –
- **last_evaluated_key** – If set, provides the starting point for query.
- **attributes_to_get** – If set, only returns these elements
- **page_size** – Page size of the query to DynamoDB
- **filters** – A dictionary of filters to be used in the query

classmethod `Model.`**rate_limited_scan**(*filter_condition=None*, *attributes_to_get=None*, *segment=None*, *total_segments=None*, *limit=None*, *conditional_operator=None*, *last_evaluated_key=None*, *page_size=None*, *timeout_seconds=None*, *read_capacity_to_consume_per_second=10*, *allow_rate_limited_scan_without_consumed_capacity=None*, *max_sleep_between_retry=10*, *max_consecutive_exceptions=30*, *consistent_read=None*, ***filters*)

Scans the items in the table at a definite rate. Invokes the low level rate_limited_scan API.

Parameters

- **filter_condition** – Condition used to restrict the scan results
- **attributes_to_get** – A list of attributes to return.
- **segment** – If set, then scans the segment
- **total_segments** – If set, then specifies total segments
- **limit** – Used to limit the number of results returned
- **conditional_operator** –
- **last_evaluated_key** – If set, provides the starting point for scan.
- **page_size** – Page size of the scan to DynamoDB
- **filters** – A list of item filters
- **timeout_seconds** – Timeout value for the rate_limited_scan method, to prevent it from running infinitely
- **read_capacity_to_consume_per_second** – Amount of read capacity to consume every second
- **allow_rate_limited_scan_without_consumed_capacity** – If set, proceeds without rate limiting if the server does not support returning consumed capacity in responses.
- **max_sleep_between_retry** – Max value for sleep in seconds in between scans during throttling/rate limit scenarios

- **max_consecutive_exceptions** – Max number of consecutive provision throughput exceeded exceptions for scan to exit
- **consistent_read** – If True, a consistent read is performed

Model.**refresh**(*consistent_read=False*)
Retrieves this object's data from dynamodb and syncs this local object

Parameters **consistent_read** – If True, then a consistent read is performed.

Model.**save**(*condition=None*, *conditional_operator=None*, ***expected_values*)
Save this object to dynamodb

classmethod Model.**scan**(*filter_condition=None*, *segment=None*, *total_segments=None*, *limit=None*, *conditional_operator=None*, *last_evaluated_key=None*, *page_size=None*, *consistent_read=None*, ***filters*)
Iterates through all items in the table

Parameters

- **filter_condition** – Condition used to restrict the scan results
- **segment** – If set, then scans the segment
- **total_segments** – If set, then specifies total segments
- **limit** – Used to limit the number of results returned
- **conditional_operator** –
- **last_evaluated_key** – If set, provides the starting point for scan.
- **page_size** – Page size of the scan to DynamoDB
- **filters** – A list of item filters
- **consistent_read** – If True, a consistent read is performed

Model.**update**(*attributes=None*, *actions=None*, *condition=None*, *conditional_operator=None*, ***expected_values*)
Updates an item using the UpdateItem operation.

Parameters **attributes** – A dictionary of attributes to update in the following format {

attr_name: {'value': 10, 'action': 'ADD'}, next_attr: {'value': True, 'action': 'PUT'},

}

Model.**update_item**(*attribute*, *value=None*, *action=None*, *condition=None*, *conditional_operator=None*, ***expected_values*)
Updates an item using the UpdateItem operation.

This should be used for updating a single attribute of an item.

Parameters

- **attribute** – The name of the attribute to be updated
- **value** – The new value for the attribute.
- **action** – The action to take if this item already exists. See: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_UpdateItem.html#DDB-UpdateItem-request-AttributeUpdate

PynamoDB attributes

class pynamodb.attributes.**Attribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)

An attribute of a model

deserialize(*value*)
Performs any needed deserialization on the value

serialize(*value*)
This method should return a dynamodb compatible value

class pynamodb.attributes.**BinaryAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)

A binary attribute

deserialize(*value*)
Returns a decoded string from base64

serialize(*value*)
Returns a base64 encoded binary string

class pynamodb.attributes.**BinarySetAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)

A binary set

deserialize(*value*)
Returns a decoded string from base64

serialize(*value*)
Returns a base64 encoded binary string

class pynamodb.attributes.**BooleanAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)

A class for boolean attributes

class pynamodb.attributes.**JSONAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)

A JSON Attribute

Encodes JSON to unicode internally

deserialize(*value*)
Deserializes JSON

serialize(*value*)
Serializes JSON to unicode

class pynamodb.attributes.**LegacyBooleanAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)

A class for legacy boolean attributes

Previous versions of this library serialized bools as numbers. This class allows you to continue to use that functionality.

class pynamodb.attributes.**MapAttribute**(***attributes*)
A Map Attribute

The MapAttribute class can be used to store a JSON document as "raw" name-value pairs, or it can be subclassed and the document fields represented as class attributes using Attribute instances.

To support the ability to subclass MapAttribute and use it as an AttributeContainer, instances of MapAttribute behave differently based both on where they are instantiated and on their type. Because of this complicated behavior, a bit of an introduction is warranted.

Models that contain a MapAttribute define its properties using a class attribute on the model. For example, below we define "MyModel" which contains a MapAttribute "my_map":

class MyModel(Model): my_map = MapAttribute(attr_name="dynamo_name", default={})

When instantiated in this manner (as a class attribute of an AttributeContainer class), the MapAttribute class acts as an instance of the Attribute class. The instance stores data about the attribute (in this example the dynamo name and default value), and acts as a data descriptor, storing any value bound to it on the *attribute_values* dictionary of the containing instance (in this case an instance of MyModel).

Unlike other Attribute types, the value that gets bound to the containing instance is a new instance of MapAttribute, not an instance of the primitive type. For example, a UnicodeAttribute stores strings in the *attribute_values* of the containing instance; a MapAttribute does not store a dict but instead stores a new instance of itself. This difference in behavior is necessary when subclassing MapAttribute in order to access the Attribute data descriptors that represent the document fields.

For example, below we redefine "MyModel" to use a subclass of MapAttribute as "my_map":

class MyMapAttribute(MapAttribute): my_internal_map = MapAttribute()

class MyModel(Model): my_map = MyMapAttribute(attr_name="dynamo_name", default = {})

In order to set the value of my_internal_map on an instance of MyModel we need the bound value for "my_map" to be an instance of MapAttribute so that it acts as a data descriptor:

MyModel().my_map.my_internal_map = {'foo': 'bar'}

That is the attribute access of "my_map" must return a MyMapAttribute instance and not a dict.

When an instance is used in this manner (bound to an instance of an AttributeContainer class), the MapAttribute class acts as an AttributeContainer class itself. The instance does not store data about the attribute, and does not act as a data descriptor. The instance stores name-value pairs in its internal *attribute_values* dictionary.

Thus while MapAttribute multiply inherits from Attribute and AttributeContainer, a MapAttribute instance does not behave as both an Attribute AND an AttributeContainer. Rather an instance of MapAttribute behaves EITHER as an Attribute OR as an AttributeContainer, depending on where it was instantiated.

So, how do we create this dichotomous behavior? Using the AttributeContainerMeta metaclass. All MapAttribute instances are initialized as AttributeContainers only. During construction of AttributeContainer classes (subclasses of MapAttribute and Model), any instances that are class attributes are transformed from AttributeContainers to Attributes (via the *_make_attribute* method call).

deserialize(*values*)
Decode as a dict.

class pynamodb.attributes.**MapAttributeMeta**(*name*, *bases*, *attrs*)
This is only here for backwards compatibility: i.e. so type(MapAttribute) == MapAttributeMeta

class pynamodb.attributes.**NumberAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)
A number attribute

deserialize(*value*)
Decode numbers from JSON

serialize(*value*)
Encode numbers as JSON

class pynamodb.attributes.**NumberSetAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)
A number set attribute

class pynamodb.attributes.**SetMixin**
Adds (de)serialization methods for sets

deserialize(*value*)
Deserializes a set

serialize(*value*)
Serializes a set

Because dynamodb doesn't store empty attributes, empty sets return None

class pynamodb.attributes.**UTCDateTimeAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)
An attribute for storing a UTC Datetime

deserialize(*value*)
Takes a UTC datetime string and returns a datetime object

serialize(*value*)
Takes a datetime object and returns a string

class pynamodb.attributes.**UnicodeAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)
A unicode attribute

serialize(*value*)
Returns a unicode string

class pynamodb.attributes.**UnicodeSetAttribute**(*hash_key=False*, *range_key=False*, *null=None*, *default=None*, *attr_name=None*)
A unicode set

element_serialize(*value*)
This serializes unicode / strings out as unicode strings. It does not touch the value if it is already a unicode str :param value: :return:

PynamoDB Indexes

class pynamodb.indexes.**AllProjection**
An ALL projection

class pynamodb.indexes.**GlobalSecondaryIndex**
A global secondary index

class pynamodb.indexes.**IncludeProjection**(*non_attr_keys=None*)
An INCLUDE projection

class pynamodb.indexes.**Index**
Base class for secondary indexes

classmethod count(*hash_key*, *range_key_condition=None*, *filter_condition=None*, *consistent_read=False*, ***filters*)
Count on an index

classmethod query(*hash_key*, *range_key_condition=None*, *filter_condition=None*, *scan_index_forward=None*, *consistent_read=False*, *limit=None*, *last_evaluated_key=None*, *attributes_to_get=None*, ***filters*)
Queries an index

class pynamodb.indexes.**IndexMeta**(*name*, *bases*, *attrs*)
Index meta class

This class is here to allow for an index *Meta* class that contains the index settings

class pynamodb.indexes.**KeysOnlyProjection**
Keys only projection

class pynamodb.indexes.**LocalSecondaryIndex**
A local secondary index

class pynamodb.indexes.**Projection**
A class for presenting projections

Low Level API

PynamoDB lowest level connection

class pynamodb.connection.**Connection**(*region=None*, *host=None*, *session_cls=None*, *request_timeout_seconds=None*, *max_retry_attempts=None*, *base_backoff_ms=None*)
A higher level abstraction over botocore

batch_get_item(*table_name*, *keys*, *consistent_read=None*, *return_consumed_capacity=None*, *attributes_to_get=None*)
Performs the batch get item operation

batch_write_item(*table_name*, *put_items=None*, *delete_items=None*, *return_consumed_capacity=None*, *return_item_collection_metrics=None*)
Performs the batch_write_item operation

client
Returns a botocore dynamodb client

create_table(*table_name*, *attribute_definitions=None*, *key_schema=None*, *read_capacity_units=None*, *write_capacity_units=None*, *global_secondary_indexes=None*, *local_secondary_indexes=None*, *stream_specification=None*)
Performs the CreateTable operation

delete_item(*table_name*, *hash_key*, *range_key=None*, *condition=None*, *expected=None*, *conditional_operator=None*, *return_values=None*, *return_consumed_capacity=None*, *return_item_collection_metrics=None*)
Performs the DeleteItem operation and returns the result

delete_table(*table_name*)
Performs the DeleteTable operation

describe_table(*table_name*)
Performs the DescribeTable operation

dispatch(*operation_name*, *operation_kwargs*)
Dispatches *operation_name* with arguments *operation_kwargs*

Raises TableDoesNotExist if the specified table does not exist

get_attribute_type(*table_name*, *attribute_name*, *value=None*)
Returns the proper attribute type for a given attribute name :param value: The attribute value an be supplied just in case the type is already included

get_conditional_operator(*operator*)
Returns a dictionary containing the correct conditional operator, validating it first.

get_consumed_capacity_map(*return_consumed_capacity*)
Builds the consumed capacity map that is common to several operations

get_exclusive_start_key_map(*table_name*, *exclusive_start_key*)
Builds the exclusive start key attribute map

get_expected_map(*table_name*, *expected*)
Builds the expected map that is common to several operations

get_identifier_map(*table_name*, *hash_key*, *range_key=None*, *key='Key'*)
Builds the identifier map that is common to several operations

get_item(*table_name*, *hash_key*, *range_key=None*, *consistent_read=False*, *attributes_to_get=None*)
Performs the GetItem operation and returns the result

get_item_attribute_map(*table_name*, *attributes*, *item_key='Item'*, *pythonic_key=True*)
Builds up a dynamodb compatible AttributeValue map

get_item_collection_map(*return_item_collection_metrics*)
Builds the item collection map

get_meta_table(*table_name*, *refresh=False*)
Returns a MetaTable

get_query_filter_map(*table_name*, *query_filters*)
Builds the QueryFilter object needed for the Query operation

get_return_values_map(*return_values*)
Builds the return values map that is common to several operations

list_tables(*exclusive_start_table_name=None*, *limit=None*)
Performs the ListTables operation

parse_attribute(*attribute*, *return_type=False*)
Returns the attribute value, where the attribute can be a raw attribute value, or a dictionary containing the type: {'S': 'String value'}

put_item(*table_name*, *hash_key*, *range_key=None*, *attributes=None*, *condition=None*, *expected=None*, *conditional_operator=None*, *return_values=None*, *return_consumed_capacity=None*, *return_item_collection_metrics=None*)
Performs the PutItem operation and returns the result

query(*table_name*, *hash_key*, *range_key_condition=None*, *filter_condition=None*, *attributes_to_get=None*, *consistent_read=False*, *exclusive_start_key=None*, *index_name=None*, *key_conditions=None*, *query_filters=None*, *conditional_operator=None*, *limit=None*, *return_consumed_capacity=None*, *scan_index_forward=None*, *select=None*)
Performs the Query operation and returns the result

rate_limited_scan(*table_name*, *filter_condition=None*, *attributes_to_get=None*, *page_size=None*, *limit=None*, *conditional_operator=None*, *scan_filter=None*, *exclusive_start_key=None*, *segment=None*, *total_segments=None*, *timeout_seconds=None*, *read_capacity_to_consume_per_second=10*, *allow_rate_limited_scan_without_consumed_capacity=None*, *max_sleep_between_retry=10*, *max_consecutive_exceptions=10*, *consistent_read=None*)
Performs a rate limited scan on the table. The API uses the scan API to fetch items from DynamoDB. The rate_limited_scan uses the 'ConsumedCapacity' value returned from DynamoDB to limit the rate of the scan. 'ProvisionedThroughputExceededException' is also handled and retried.

Parameters

- **table_name** – Name of the table to perform scan on.
- **filter_condition** – Condition used to restrict the scan results
- **attributes_to_get** – A list of attributes to return.

- **page_size** – Page size of the scan to DynamoDB
- **limit** – Used to limit the number of results returned
- **conditional_operator** –
- **scan_filter** – A map indicating the condition that evaluates the scan results
- **exclusive_start_key** – If set, provides the starting point for scan.
- **segment** – If set, then scans the segment
- **total_segments** – If set, then specifies total segments
- **timeout_seconds** – Timeout value for the rate_limited_scan method, to prevent it from running infinitely
- **read_capacity_to_consume_per_second** – Amount of read capacity to consume every second
- **allow_rate_limited_scan_without_consumed_capacity** – If set, proceeds without rate limiting if the server does not support returning consumed capacity in responses.
- **max_sleep_between_retry** – Max value for sleep in seconds in between scans during throttling/rate limit scenarios
- **max_consecutive_exceptions** – Max number of consecutive ProvisionedThroughputExceededException exception for scan to exit
- **consistent_read** – enable consistent read

requests_session
Return a requests session to execute prepared requests using the same pool

scan(*table_name*, *filter_condition=None*, *attributes_to_get=None*, *limit=None*, *conditional_operator=None*, *scan_filter=None*, *return_consumed_capacity=None*, *exclusive_start_key=None*, *segment=None*, *total_segments=None*, *consistent_read=None*)
Performs the scan operation

session
Returns a valid botocore session

update_item(*table_name*, *hash_key*, *range_key=None*, *actions=None*, *attribute_updates=None*, *condition=None*, *expected=None*, *return_consumed_capacity=None*, *conditional_operator=None*, *return_item_collection_metrics=None*, *return_values=None*)
Performs the UpdateItem operation

update_table(*table_name*, *read_capacity_units=None*, *write_capacity_units=None*, *global_secondary_index_updates=None*)
Performs the UpdateTable operation

class pynamodb.connection.**TableConnection**(*table_name*, *region=None*, *host=None*, *session_cls=None*, *request_timeout_seconds=None*, *max_retry_attempts=None*, *base_backoff_ms=None*)
A higher level abstraction over botocore

batch_get_item(*keys*, *consistent_read=None*, *return_consumed_capacity=None*, *attributes_to_get=None*)
Performs the batch get item operation

batch_write_item(*put_items=None*, *delete_items=None*, *return_consumed_capacity=None*, *return_item_collection_metrics=None*)
Performs the batch_write_item operation

create_table(*attribute_definitions=None*, *key_schema=None*, *read_capacity_units=None*, *write_capacity_units=None*, *global_secondary_indexes=None*, *local_secondary_indexes=None*, *stream_specification=None*)
Performs the CreateTable operation and returns the result

delete_item(*hash_key*, *range_key=None*, *condition=None*, *expected=None*, *conditional_operator=None*, *return_values=None*, *return_consumed_capacity=None*, *return_item_collection_metrics=None*)
Performs the DeleteItem operation and returns the result

delete_table()
Performs the DeleteTable operation and returns the result

describe_table()
Performs the DescribeTable operation and returns the result

get_item(*hash_key*, *range_key=None*, *consistent_read=False*, *attributes_to_get=None*)
Performs the GetItem operation and returns the result

put_item(*hash_key*, *range_key=None*, *attributes=None*, *condition=None*, *expected=None*, *conditional_operator=None*, *return_values=None*, *return_consumed_capacity=None*, *return_item_collection_metrics=None*)
Performs the PutItem operation and returns the result

query(*hash_key*, *range_key_condition=None*, *filter_condition=None*, *attributes_to_get=None*, *consistent_read=False*, *exclusive_start_key=None*, *index_name=None*, *key_conditions=None*, *query_filters=None*, *limit=None*, *return_consumed_capacity=None*, *scan_index_forward=None*, *conditional_operator=None*, *select=None*)
Performs the Query operation and returns the result

rate_limited_scan(*filter_condition=None*, *attributes_to_get=None*, *page_size=None*, *limit=None*, *conditional_operator=None*, *scan_filter=None*, *segment=None*, *total_segments=None*, *exclusive_start_key=None*, *timeout_seconds=None*, *read_capacity_to_consume_per_second=None*, *allow_rate_limited_scan_without_consumed_capacity=None*, *max_sleep_between_retry=None*, *max_consecutive_exceptions=None*, *consistent_read=None*)
Performs the scan operation with rate limited

scan(*filter_condition=None*, *attributes_to_get=None*, *limit=None*, *conditional_operator=None*, *scan_filter=None*, *return_consumed_capacity=None*, *segment=None*, *total_segments=None*, *exclusive_start_key=None*, *consistent_read=None*)
Performs the scan operation

update_item(*hash_key*, *range_key=None*, *actions=None*, *attribute_updates=None*, *condition=None*, *expected=None*, *conditional_operator=None*, *return_consumed_capacity=None*, *return_item_collection_metrics=None*, *return_values=None*)
Performs the UpdateItem operation

update_table(*read_capacity_units=None*, *write_capacity_units=None*, *global_secondary_index_updates=None*)
Performs the UpdateTable operation and returns the result

Exceptions

exception pynamodb.exceptions.**PynamoDBConnectionError**(*msg=None*, *cause=None*)
A base class for connection errors

exception pynamodb.exceptions.**DeleteError**(*msg=None*, *cause=None*)
Raised when an error occurs deleting an item

exception pynamodb.exceptions.**QueryError**(*msg=None*, *cause=None*)
Raised when queries fail

exception pynamodb.exceptions.**ScanError**(*msg=None*, *cause=None*)
Raised when a scan operation fails

exception pynamodb.exceptions.**PutError**(*msg=None*, *cause=None*)
Raised when an item fails to be created

exception pynamodb.exceptions.**UpdateError**(*msg=None*, *cause=None*)
Raised when an item fails to be updated

exception pynamodb.exceptions.**GetError**(*msg=None*, *cause=None*)
Raised when an item fails to be retrieved

exception pynamodb.exceptions.**TableError**(*msg=None*, *cause=None*)
An error involving a dynamodb table operation

exception pynamodb.exceptions.**TableDoesNotExist**(*table_name*)
Raised when an operation is attempted on a table that doesn't exist

exception pynamodb.exceptions.**DoesNotExist**(*msg=None*, *cause=None*)
Raised when an item queried does not exist

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

T

U