PynamoDB Documentation

Release 6.0.2

Jharrod LaFon

Jan 24, 2025

CONTENTS

1 Features **3**

2 Topics **5**

2.1 Usage 5
2.2 Basic Tutorial 8
2.3 Index Queries 13
2.4 Batch Operations 16
2.5 Update Operations 18
2.6 Conditional Operations 21
2.7 Polymorphism 24
2.8 Custom Attributes 25
2.9 Transaction Operations 29
2.10 Optimistic Locking 34
2.11 Rate-Limited Operation 38
2.12 Use PynamoDB Locally 39
2.13 Signals 40
2.14 PynamoDB Examples 41
2.15 Settings 42
2.16 Low Level API 43
2.17 AWS Access 45
2.18 Logging 46
2.19 Contributing 46
2.20 Release Notes 48
2.21 Versioning Scheme 74
2.22 Upgrading 74

3 API docs **79**

3.1 API 79

4 Indices and tables **97**

Python Module Index **99**

Index **101**

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

ONE

FEATURES

- Python 3 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 TWO

TOPICS

2.1 Usage

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

2.1.1 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 in local memory:

```
>>> 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()
```

2.1.2 Changing items

Changing existing items in the database can be done using either *update()* or *save()*. There are important differences between the two.

Use of *save()* looks like this:

```
user = UserModel.get('test@example.com')
user.first_name = 'Robert'
user.save()
```

Use of *update()* (in its simplest form) looks like this:

```
user = UserModel.get('test@example.com')
user.update(
  actions=[
    UserModel.first_name.set('Robert')
  ]
)
```

save() will entirely replace an object (it internally uses PutItem). As a consequence, even if you modify only one attribute prior to calling *save()*, the entire object is re-written. Any modifications done to the same user by other processes will be lost, even if made to other attributes that you did not change. To avoid this, use *update()* to perform more fine grained updates or see the *Conditional Operations* for how to avoid race conditions entirely.

Additionally, PynamoDB ignores attributes it does not know about when reading an object from the database. As a result, if the item in DynamoDB contains attributes not declared in your model, *save()* will cause those attributes to be deleted.

In particular, performing a rolling upgrade of your application after having added an attribute is an example of such a situation. To avoid data loss, either avoid using *save()* or perform a multi-step update with the first step is to upgrade to a version that merely declares the attribute on the model without ever setting it to any value.

2.1.3 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)
```

2.1.4 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'))
```

2.1.5 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)
```

2.2 Basic Tutorial

PynamoDB is an attempt to be a Pythonic interface to DynamoDB that supports all of DynamoDB's powerful features. 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

2.2.1 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.

2.2.2 Installation

```
$ pip install pynamodb
```

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

Alternatively, if you are running Anaconda or miniconda, use:

```
$ conda install -c conda-forge pynamodb
```

2.2.3 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)
```

(continues on next page)

(continued from previous page)

```
    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'
        # Optional: Specify the hostname only if it needs to be changed from the default␣
↪AWS setting
        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 overridden. 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*
- *MapAttribute*

All of these built in attributes handle serializing and deserializing themselves.

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()
```

2.2.4 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()),
    ])
```

Update actions use the update expression syntax (see *Update Expressions*).

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

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

2.3 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.

2.3.1 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.

2.3.2 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)
```

2.3.3 Local Secondary Indexes

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

```python
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)
```

Every local secondary index must meet the following conditions:

- The partition key (hash key) is the same as that of its base table.
- The sort key (range key) consists of exactly one scalar attribute. The range key can be any attribute.
- The sort key (range key) of the base table is projected into the index, where it acts as a non-key attribute.

2.3.4 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`:

```python
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:

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

2.3.5 Pagination and last evaluated key

The query returns a `ResultIterator` object that transparently paginates through results. To stop iterating and allow the caller to continue later on, use the `last_evaluated_key` property of the iterator:

```python
def iterate_over_page(last_evaluated_key = None):
    results = TestModel.view_index.query('foo', TestModel.view > 0,
                                         limit=10,
                                         last_evaluated_key=last_evaluated_key)
    for item in results:
```

(continues on next page)

(continued from previous page)

```
        ...
    return results.last_evaluated_key
```

The `last_evaluated_key` is effectively the key attributes of the last iterated item; the next returned items will be the items following it. For index queries, the returned `last_evaluated_key` will contain both the table's hash/range keys and the indexes hash/range keys. This is due to the fact that DynamoDB indexes have no uniqueness constraint, i.e. the same hash/range pair can map to multiple items. For the example above, the `last_evaluated_key` will look like:

```
{
    "forum": {"S": "..."},
    "thread": {"S": "..."},
    "view": {"N": "..."}
}
```

2.4 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)
```

2.4.1 Batch Writes

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

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

2.4.2 Batch Gets

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

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

2.4.3 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.

2.4.4 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)
```

2.4.5 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))
```

2.5 Update Operations

The UpdateItem DynamoDB operations allows you to create or modify attributes of an item using an update expression. 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 (
    ListAttribute, UnicodeAttribute, UnicodeSetAttribute, NumberAttribute
)


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

    forum_name = UnicodeAttribute(hash_key=True)
    subjects = UnicodeSetAttribute(default=set)
    author = UnicodeAttribute(null=True)
    views = NumberAttribute(default=0)
    notes = ListAttribute(default=list)
```

2.5.1 Update Expressions

PynamoDB supports creating update 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.

DynamoDB Action / Operator	PynamoDB Syntax	Attribute Types	Example
SET	set(*value*)	Any	`Thread.views.set(10)`
REMOVE	remove()	Any	`Thread.notes.remove()`
REMOVE	remove()	Element of List	`Thread.notes[0].remove()`
ADD	add(*number*)	Number	`Thread.views.add(1)`
ADD	add(*set*)	Set	`Thread.subjects.add({'A New Subject', 'Another New Subject'})`
DELETE	delete(*set*)	Set	`Thread.subjects.delete({'An Old Subject'})`

The following expressions and functions can only be used in the context of the above actions:

DynamoDB Action / Operator	PynamoDB Syntax	Attribute Types	Example
attr_or_value_1 + *attr_or_value_2*	*attr_or_value_1* + *attr_or_value_2*	Number	`Thread.views + 5`
attr_or_value_1 - *attr_or_value_2*	*attr_or_value_1* - *attr_or_value_2*	Number	`5 - Thread.views`
list_append(*attr* , *value*)	append(*value*)	List	`Thread.notes.append(['my last note'])`
list_append(*value* , *attr*)	prepend(*value*)	List	`Thread.notes.prepend(['my first note'])`
if_not_exists(*attr*, *value*)	*attr* | *value*	Any	`Thread.forum_name | 'Default Forum Name'`

2.5.2 `set` action

The `set` action is the simplest action as it overwrites any previously stored value:

```
thread.update(actions=[
    Thread.views.set(10),
])
assert thread.views == 10
```

It can reference existing values (from this or other attributes) for arithmetics and concatenation:

```
# Increment views by 5
thread.update(actions=[
    Thread.views.set(Thread.views + 5)
])

# Append 2 notes
thread.update(actions=[
    Thread.notes.set(
        Thread.notes.append([
            'my last note',
            'p.s. no, really, this is my last note',
        ]),
    )
])

# Prepend a note
thread.update(actions=[
    Thread.notes.set(
        Thread.notes.prepend([
            'my first note',
        ]),
    )
])

# Set author to John Doe unless there's already one
thread.update(actions=[
```

(continues on next page)

(continued from previous page)

```
    Thread.author.set(Thread.author | 'John Doe')
])
```

2.5.3 `remove` action

The `remove` action unsets attributes:

```
thread.update(actions=[
    Thread.views.remove(),
])
assert thread.views == 0  # default value
```

It can also be used to remove elements from a list attribute:

```
# Remove the first note
thread.update(actions=[
    Thread.notes[0].remove(),
])
```

2.5.4 `add` action

Applying to (binary, number and string) set attributes, the add action adds elements to the set:

```
# Add the subjects 'A New Subject' and 'Another New Subject'
thread.update(actions=[
    Thread.subjects.add({'A New Subject', 'Another New Subject'})
])
```

Applying to number attributes, the add action increments or decrements the number and is equivalent to a `set` action:

```
# Increment views by 5
thread.update(actions=[
    Thread.views.add(5),
])
# Also increment views by 5
thread.update(actions=[
    Thread.views.set(Thread.views + 5),
])
```

2.5.5 `delete` action

For set attributes, the `delete` action is the opposite of the add action:

```
# Delete the subject 'An Old Subject'
thread.update(actions=[
    Thread.subjects.delete({'An Old Subject'})
])
```

2.6 Conditional Operations

Some DynamoDB operations support the inclusion of conditions. The user can supply a condition to be evaluated by DynamoDB before an item is modified (with save, update and delete) or before an item is included in the result (with query and scan). See the official documentation for more details.

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

```python
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)
    authors = ListAttribute()
    properties = MapAttribute()
```

2.6.1 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	Attribute Types	Example
=	==	Any	`Thread.forum_name == 'Some Forum'`
<>	!=	Any	`Thread.forum_name != 'Some Forum'`
<	<	Binary, Number, String	`Thread.views < 10`
<=	<=	Binary, Number, String	`Thread.views <= 10`
>	>	Binary, Number, String	`Thread.views > 10`
>=	>=	Binary, Number, String	`Thread.views >= 10`
BETWEEN	between(*lower* , *upper*)	Binary, Number, String	`Thread.views.between(1, 5)`
IN	is_in(**values*)	Binary, Number, String	`Thread.subject.is_in('Subject', 'Other Subject')`
attribute_exists (*path*)	exists()	Any	`Thread.forum_name.exists()`
attribute_not_exists (*path*)	does_not_exist()	Any	`Thread.forum_name.does_not_exist()`
attribute_type (*path* , *type*)	is_type()	Any	`Thread.forum_name.is_type()`
begins_with (*path* , *substr*)	startswith(*prefix*)	String	`Thread.subject.startswith('Example')`
contains (*path* , *operand*)	contains(*item*)	Set, String	`Thread.subject.contains('foobar')`
size (*path*)	size(*attribute*)	Binary, List, Map, Set, String	`size(Thread.subject) == 10`
AND	&	Any	`(Thread.views > 1) & (Thread.views < 5)`
OR	|	Any	`(Thread.views < 1) | (Thread.views > 5)`
NOT	~	Any	`~Thread.subject.contains('foobar')`

Conditions expressions using nested list and map attributes can be created with Python's item operator `[]`.

```
# Query for threads where 'properties' map contains key 'emoji'
Thread.query(..., filter_condition=Thread.properties['emoji'].exists())

# Query for threads where the first author's name contains "John"
Thread.authors[0].contains("John")
```

Conditions can be composited using `&` (AND) and `|` (OR) operators. For the `&` (AND) operator, the left-hand side operand can be `None` to allow easier chaining of filter conditions:

```
condition = None

if request.subject:
  condition &= Thread.subject.contains(request.subject)

if request.min_views:
  condition &= Thread.views >= min_views
```

(continues on next page)

(continued from previous page)

```
results = Thread.query(..., filter_condition=condition)
```

2.6.2 Conditioning on keys

When writing to a table (save, update, delete), an `exists()` condition on a key attribute ensures that the item already exists (under the given key) in the table before the operation. For example, a *save* or *update* would update an existing item, but fail if the item does not exist.

Correspondingly, a `does_not_exist()` condition on a key ensures that the item does not exist. For example, a *save* with such a condition ensures that it's not overwriting an existing item.

For models with a range key, conditioning `exists()` on either the hash key or the range key has the same effect. There is no way to condition on _some_ item existing with the given hash key. For example:

```
thread = Thread('DynamoDB', 'Using conditions')

# This will fail if the item ('DynamoDB', 'Using conditions') does not exist,
# even if the item ('DynamoDB', 'Using update expressions') does.
thread.save(condition=Thread.forum_name.exists())

# This will fail if the item ('DynamoDB', 'Using conditions') does not exist,
# even if the item ('S3', 'Using conditions') does.
thread.save(condition=Thread.subject.exists())
```

2.6.3 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.subject.contains('foobar')))
```

2.6.4 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))
```

2.6.5 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))
```

2.6.6 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 e.cause_response_code = "ConditionalCheckFailedException":
        raise ThreadDidNotExistError()
```

2.7 Polymorphism

PynamoDB supports polymorphism through the use of discriminators.

A discriminator is a value that is written to DynamoDB that identifies the python class being stored.

2.7.1 Discriminator Attributes

The discriminator value is stored using a special attribute, the DiscriminatorAttribute. Only a single DiscriminatorAttribute can be defined on a class.

The discriminator value can be assigned to a class as part of the definition:

```
class ParentClass(MapAttribute):
    cls = DiscriminatorAttribute()

class ChildClass(ParentClass, discriminator='child'):
    pass
```

Declaring the discriminator value as part of the class definition will automatically register the class with the discriminator attribute. A class can also be registered manually:

```
class ParentClass(MapAttribute):
    cls = DiscriminatorAttribute()

class ChildClass(ParentClass):
    pass

ParentClass._cls.register_class(ChildClass, 'child')
```

Note: A class may be registered with a discriminator attribute multiple times. Only the first registered value is used during serialization; however, any registered value can be used to deserialize the class. This behavior is intended to facilitate migrations if discriminator values must be changed.

Warning: Discriminator values are written to DynamoDB. Changing the value after items have been saved to the database can result in deserialization failures. In order to read items with an old discriminator value, the old value must be manually registered.

2.7.2 Model Discriminators

Model classes also support polymorphism through the use of discriminators. (Note: currently discriminator attributes cannot be used as the hash or range key of a table.)

```
class ParentModel(Model):
    class Meta:
        table_name = 'polymorphic_table'
    id = UnicodeAttribute(hash_key=True)
    cls = DiscriminatorAttribute()

class FooModel(ParentModel, discriminator='Foo'):
    foo = UnicodeAttribute()

class BarModel(ParentModel, discriminator='Bar'):
    bar = UnicodeAttribute()

BarModel(id='Hello', bar='World!').serialize()
# {'id': {'S': 'Hello'}, 'cls': {'S': 'Bar'}, 'bar': {'S': 'World!'}}
```

Note: Read operations that are performed on a class that has a discriminator value are slightly modified to ensure that only instances of the class are returned. Query and scan operations transparently add a filter condition to ensure that only items with a matching discriminator value are returned. Get and batch get operations will raise a `ValueError` if the returned item(s) are not a subclass of the model being read.

2.8 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 `UTCDateTimeAttribute`.

2.8.1 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`.

2.8.2 Writing your own attribute

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

```python
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(self, value):
        # convert the value to binary and return it

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

2.8.3 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.

```python
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):
        """
```

(continues on next page)

(continued from previous page)

```
        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>
```

2.8.4 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 subclasses of `Attribute`. 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
```

2.8.5 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)
```

As with a model and its top-level attributes, a PynamoDB MapAttribute will ignore sub-attributes it does not know about during deserialization. As a result, if the item in DynamoDB contains sub-attributes not declared as properties of the corresponding MapAttribute, save() will cause those sub-attributes to be deleted.

`DynamicMapAttribute` is a subclass of `MapAttribute` which allows you to mix and match defined attributes and undefined attributes.

```
from pynamodb.attributes import DynamicMapAttribute, UnicodeAttribute

class CarInfo(DynamicMapAttribute):
    make = UnicodeAttribute(null=False)
    model = UnicodeAttribute(null=True)

car = CarInfo(make='Make-A', model='Model-A', year=1975)
other_car = CarInfo(make='Make-A', model='Model-A', year=1975, seats=3)
```

2.9 Transaction Operations

Transact operations are similar to Batch operations, with the key differences being that the writes support the inclusion of condition checks, and they all must fail or succeed together.

Transaction operations are supported using context managers. Keep in mind that DynamoDB imposes limits on the number of items that a single transaction can contain.

Suppose you have defined a BankStatement model, like in the example below.

```
from pynamodb.models import Model
from pynamodb.attributes import BooleanAttribute, NumberAttribute, UnicodeAttribute

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

    user_id = UnicodeAttribute(hash_key=True)
    account_balance = NumberAttribute(default=0)
    is_active = BooleanAttribute()
```

2.9.1 Transact Writes

A *TransactWrite* can be initialized with the following parameters:

- `connection` (required) - the *Connection* used to make the request (see *Low Level API*)
- `client_request_token` - an idempotency key for the request (see ClientRequestToken in the DynamoDB API reference)
- `return_consumed_capacity` - determines the level of detail about provisioned throughput consumption that is returned in the response (see ReturnConsumedCapacity in the DynamoDB API reference)
- `return_item_collection_metrics` - determines whether item collection metrics are returned (see ReturnItemCollectionMetrics in the DynamoDB API reference)

Here's an example of using a context manager for a *TransactWrite* operation:

```
from pynamodb.connection import Connection
from pynamodb.transactions import TransactWrite

# Two existing bank statements in the following states
user1_statement = BankStatement('user1', account_balance=2000, is_active=True)
user2_statement = BankStatement('user2', account_balance=0, is_active=True)

user1_statement.save()
user2_statement.save()

connection = Connection()

with TransactWrite(connection=connection, client_request_token='super-unique-key') as␣
↪transaction:
    # attempting to transfer funds from user1's account to user2's
    transfer_amount = 1000
    transaction.update(
        BankStatement(user_id='user1'),
        actions=[BankStatement.account_balance.add(transfer_amount * -1)],
        condition=(
            (BankStatement.account_balance >= transfer_amount) &
            (BankStatement.is_active == True)
        )
    )
    transaction.update(
        BankStatement(user_id='user2'),
        actions=[BankStatement.account_balance.add(transfer_amount)],
        condition=(BankStatement.is_active == True)
    )

user1_statement.refresh()
user2_statement.refresh()

assert user1_statement.account_balance == 1000
assert user2_statement.account_balance == 1000
```

Now, say you make another attempt to debit one of the accounts when they don't have enough money in the bank:

```
from pynamodb.exceptions import TransactWriteError

assert user1_statement.account_balance == 1000
assert user2_statement.account_balance == 1000

try:
    with TransactWrite(connection=connection, client_request_token='another-super-unique-
↪key') as transaction:
        # attempting to transfer funds from user1's account to user2's
        transfer_amount = 2000
        transaction.update(
            BankStatement(user_id='user1'),
            actions=[BankStatement.account_balance.add(transfer_amount * -1)],
            condition=(
                (BankStatement.account_balance >= transfer_amount) &
                (BankStatement.is_active == True)
            ),
            return_values=ALL_OLD
        )
        transaction.update(
            BankStatement(user_id='user2'),
            actions=[BankStatement.account_balance.add(transfer_amount)],
            condition=(BankStatement.is_active == True)
        )
except TransactWriteError as e:
    # Because the condition check on the account balance failed,
    # the entire transaction should be cancelled
    assert e.cause_response_code == 'TransactionCanceledException'
    # the first 'update' was a reason for the cancellation
    assert e.cancellation_reasons[0].code == 'ConditionalCheckFailed'
    # when return_values=ALL_OLD, the old values can be accessed from the raw_item␣
↪property
    assert BankStatement.from_dynamodb_dict(e.cancellation_reasons[0].raw_item) == user1_
↪statement
    # the second 'update' wasn't a reason, but was cancelled too
    assert e.cancellation_reasons[1] is None

    user1_statement.refresh()
    user2_statement.refresh()
    # and both models should be unchanged
    assert user1_statement.account_balance == 1000
    assert user2_statement.account_balance == 1000
```

Condition Check

The `ConditionCheck` operation is used on a *TransactWrite* to check if the current state of a record you aren't modifying within the overall transaction fits some criteria that, if it fails, would cause the entire transaction to fail. The `condition` argument is of type *Conditional Operations*.

- `model_cls` (required)
- `hash_key` (required)
- `range_key` (optional)
- `condition` (required) - of type `Condition` (see *Conditional Operations*)

```
with TransactWrite(connection=connection) as transaction:
    transaction.condition_check(BankStatement, 'user1', condition=(BankStatement.is_
↪active == True))
```

Delete

The `Delete` operation functions similarly to `Model.delete`.

- `model` (required)
- `condition` (optional) - of type `Condition` (see *Conditional Operations*)

```
statement = BankStatement.get('user1')

with TransactWrite(connection=connection) as transaction:
    transaction.delete(statement, condition=(~BankStatement.is_active))
```

Save

The `Put` operation functions similarly to `Model.save`.

- `model` (required)
- `condition` (optional) - of type `Condition` (see *Conditional Operations*)
- `return_values` (optional) - the values that should be returned if the condition fails ((see Put ReturnValuesOnConditionCheckFailure in the DynamoDB API reference)

```
statement = BankStatement(user_id='user3', account_balance=20, is_active=True)

with TransactWrite(connection=connection) as transaction:
    transaction.save(statement, condition=(BankStatement.user_id.does_not_exist()))
```

Update

The Update operation functions similarly to Model.update.

- model (required)
- actions (required) - a list of type Action (see *Update Expressions*)
- condition (optional) - of type Condition (see *Conditional Operations*)
- return_values (optional) - the values that should be returned if the condition fails (see Update ReturnValuesOnConditionCheckFailure in the DynamoDB API reference)

```
user1_statement = BankStatement('user1')
with TransactWrite(connection=connection) as transaction:
    transaction.update(
        user1_statement,
        actions=[BankStatement.account_balance.set(0), BankStatement.is_active.
↪set(False)]
        condition=(BankStatement.user_id.exists())
    )
```

2.9.2 Transact Gets

```
with TransactGet(connection=connection) as transaction:
    """ attempting to get records of users' bank statements """
    user1_statement_future = transaction.get(BankStatement, 'user1')
    user2_statement_future = transaction.get(BankStatement, 'user2')

user1_statement: BankStatement = user1_statement_future.get()
user2_statement: BankStatement = user2_statement_future.get()
```

The *TransactGet* operation currently only supports the Get method, which only takes the following parameters:

- model_cls (required)
- hash_key (required)
- range_key (optional)

The .get returns a class of type _ModelFuture that acts as a placeholder for the record until the transaction completes.

To retrieve the resolved model, you say *model_future.get()*. Any attempt to access this model before the transaction is complete will result in a *InvalidStateError*.

2.9.3 Error Types

You can expect some new error types with transactions, such as:

- *TransactWriteError* - thrown when a *TransactWrite* request returns a bad response (see the TransactWriteItems Errors section in the DynamoDB API reference).
- *TransactGetError* - thrown when a *TransactGet* request returns a bad response (see the TransactGetItems Errors section in the DynamoDB API reference).
- *InvalidStateError* - thrown when an attempt is made to access data on a _ModelFuture before the *TransactGet* request is completed.

2.10 Optimistic Locking

Optimistic Locking is a strategy for ensuring that your database writes are not overwritten by the writes of others. With optimistic locking, each item has an attribute that acts as a version number. If you retrieve an item from a table, the application records the version number of that item. You can update the item, but only if the version number on the server side has not changed. If there is a version mismatch, it means that someone else has modified the item before you did. The update attempt fails, because you have a stale version of the item. If this happens, you simply try again by retrieving the item and then trying to update it. Optimistic locking prevents you from accidentally overwriting changes that were made by others. It also prevents others from accidentally overwriting your changes.

Warning:

- Optimistic locking will not work properly if you use DynamoDB global tables as they use last-write-wins for concurrent updates.

See also: DynamoDBMapper Documentation on Optimistic Locking.

2.10.1 Version Attribute

To enable optimistic locking for a table, add a `VersionAttribute` to your model definition. The presence of this attribute will change the model's behaviors:

- *`save()`* and *`update()`* would increment the version attribute every time the model is persisted. This allows concurrent updates not to overwrite each other, at the expense of the latter update failing.
- *`save()`*, *`update()`* and *`delete()`* would fail if they are the "latter update" (by adding to the update's *conditions*). This behavior is optional since sometimes a more granular approach can be desired (see *Conditioning on the version*).

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

    def __eq__(self, other):
        return isinstance(other, OfficeEmployeeMap) and self.person == other.person


class Office(Model):
    class Meta:
        read_capacity_units = 1
        write_capacity_units = 1
        table_name = 'Office'
        host = "http://localhost:8000"
    office_id = UnicodeAttribute(hash_key=True)
    employees = ListAttribute(of=OfficeEmployeeMap)
    name = UnicodeAttribute()
    version = VersionAttribute()
```

The attribute is underpinned by an integer which is initialized with 1 when an item is saved for the first time and is incremented by 1 with each subsequent write operation.

```
justin = OfficeEmployeeMap(office_employee_id=str(uuid4()), person='justin')
garrett = OfficeEmployeeMap(office_employee_id=str(uuid4()), person='garrett')
office = Office(office_id=str(uuid4()), name="office", employees=[justin, garrett])
office.save()
assert office.version == 1

# Get a second local copy of Office
office_out_of_date = Office.get(office.office_id)

# Add another employee and persist the change.
office.employees.append(OfficeEmployeeMap(office_employee_id=str(uuid4()), person='lita
↪'))
office.save()
# On subsequent save or update operations the version is also incremented locally to␣
↪match the persisted value so
# there's no need to refresh between operations when reusing the local copy.
assert office.version == 2
assert office_out_of_date.version == 1
```

The version checking is implemented using DynamoDB conditional write constraints, asserting that no value exists for the version attribute on the initial save and that the persisted value matches the local value on subsequent writes.

2.10.2 Model.{update, save, delete}

These operations will fail if the local object is out-of-date.

```
@contextmanager
def assert_condition_check_fails():
    try:
        yield
    except (PutError, UpdateError, DeleteError) as e:
        assert isinstance(e.cause, ClientError)
        assert e.cause_response_code == "ConditionalCheckFailedException"
    except TransactWriteError as e:
        assert isinstance(e.cause, ClientError)
        assert e.cause_response_code == "TransactionCanceledException"
        assert any(r.code == "ConditionalCheckFailed" for r in e.cancellation_reasons)
    else:
        raise AssertionError("The version attribute conditional check should have failed.
↪")


with assert_condition_check_fails():
    office_out_of_date.update(actions=[Office.name.set('new office name')])

office_out_of_date.employees.remove(garrett)
with assert_condition_check_fails():
    office_out_of_date.save()

# After refreshing the local copy our write operations succeed.
office_out_of_date.refresh()
office_out_of_date.employees.remove(garrett)
```

(continues on next page)

(continued from previous page)

```
office_out_of_date.save()
assert office_out_of_date.version == 3

with assert_condition_check_fails():
    office.delete()
```

2.10.3 Conditioning on the version

To have *save()*, *update()* or *delete()* execute even if the item was changed by someone else, pass the `add_version_condition=False` parameter. In this mode, updates would perform unconditionally but would still increment the version: in other words, you could make other updates fail, but your update will succeed.

Done indiscriminately, this would be unsafe, but can be useful in certain scenarios:

1. For `save`, this is almost always unsafe and undesirable.
2. For `update`, use it when updating attributes for which a "last write wins" approach is acceptable, or if you're otherwise conditioning the update in a way that is more domain-specific.
3. For `delete`, use it to delete the item regardless of its contents.

For example, if your `save` operation experiences frequent "ConditionalCheckFailedException" failures, rewrite your code to call `update` with individual attributes while passing `add_version_condition=False`. By disabling the version condition, you could no longer rely on the checks you've done prior to the modification (due to what is known as the "time-of-check to time-of-use" problem). Therefore, consider adding domain-specific conditions to ensure the item in the table is in the expected state prior to the update.

For example, let's consider a hotel room-booking service with the conventional constraint that only one person can book a room at a time. We can switch from a `save` to an `update` by specifying the individual attributes and rewriting the *if* statement as a condition:

```
- if room.booked_by:
-   raise Exception("Room is already booked")
- room.booked_by = user_id
- room.save()
+ room.update(
+   actions=[Room.booked_by.set(user_id)],
+   condition=Room.booked_by.does_not_exist(),
+   add_version_condition=False,
+ )
```

2.10.4 Transactions

Transactions are supported.

Successful

```
connection = Connection(host='http://localhost:8000')

office2 = Office(office_id=str(uuid4()), name="second office", employees=[justin])
office2.save()
assert office2.version == 1
office3 = Office(office_id=str(uuid4()), name="third office", employees=[garrett])
office3.save()
assert office3.version == 1

with TransactWrite(connection=connection) as transaction:
    transaction.condition_check(Office, office.office_id, condition=(Office.name.
↪exists()))
    transaction.delete(office2)
    transaction.save(Office(office_id=str(uuid4()), name="new office", employees=[justin,
↪ garrett]))
    transaction.update(
        office3,
        actions=[
            Office.name.set('birdistheword'),
        ]
    )

try:
    office2.refresh()
except DoesNotExist:
    pass
else:
    raise AssertionError(
        'Office with office_id="{}" should have been deleted in the transaction.'
        .format(office2.office_id)
    )

assert office.version == 2
assert office3.version == 2
```

Failed

```
with assert_condition_check_fails(), TransactWrite(connection=connection) as transaction:
    transaction.save(Office(office.office_id, name='newer name', employees=[]))

with assert_condition_check_fails(), TransactWrite(connection=connection) as transaction:
    transaction.update(
        Office(office.office_id, name='newer name', employees=[]),
        actions=[Office.name.set('Newer Office Name')]
    )

with assert_condition_check_fails(), TransactWrite(connection=connection) as transaction:
    transaction.delete(Office(office.office_id, name='newer name', employees=[]))
```

2.10.5 Batch Operations

Unsupported as they do not support conditional writes.

2.11 Rate-Limited Operation

Scan, *Query* and *Count* operations can be rate-limited based on the consumed capacities returned from DynamoDB. Simply specify the *rate_limit* argument when calling these methods. Rate limited batch writes are not currently supported, but if you would like to see it in a future version, please add a feature request for it in Issues.

Note: Rate-limiting is only meant to slow operations down to conform to capacity limitations. Rate-limiting can not be used to speed operations up. Specifying a higher rate-limit that exceeds the possible writing speed allowed by the environment, will not have any effect.

2.11.1 Example Usage

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

```python
from pynamodb.models import Model
from pynamodb.attributes import (
    UnicodeAttribute
)


class User(Model):
    class Meta:
        table_name = 'Users'

    id = UnicodeAttribute(hash_key=True)
    name = UnicodeAttribute(range_key=True)
```

Here is an example using *rate-limit* in while scanning the *User* model

```python
# Using only 5 RCU per second
for user in User.scan(rate_limit=5):
    print("User id: {}, name: {}".format(user.id, user.name))
```

2.11.2 Query

You can use *rate-limit* when querying items from your table:

```python
# Using only 15 RCU per second
for user in User.query('id1', User.name.startswith('re'), rate_limit = 15):
    print("Query returned user {0}".format(user))
```

2.11.3 Count

You can use *rate-limit* when counting items in your table:

```
# Using only 15 RCU per second
count = User.count(rate_limit=15)
print("Count : {}".format(count))
```

2.12 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: 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)
```

2.12.1 Running dynalite

Make sure you have the Node Package Manager installed (see npm instructions).

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.

2.12.2 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.
- 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.

2.13 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.

2.13.1 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 identifier 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.

```python
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)
```

2.14 PynamoDB Examples

A 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.

2.14.1 Install PynamoDB

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

```
$ pip install pynamodb
```

2.14.2 Getting the examples

You can clone the PynamoDB repository to get the examples:

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

2.14.3 Running the examples

Go into the examples directory:

```
$ cd pynamodb/examples
```

2.14.4 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.

2.14.5 Running an example

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

```
$ python model.py
```

2.15 Settings

2.15.1 Settings reference

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

connect_timeout_seconds

Default: `15`

The time in seconds till a `ConnectTimeoutError` is thrown when attempting to make a connection.

read_timeout_seconds

Default: `30`

The time in seconds till a `ReadTimeoutError` is thrown when attempting to read from a connection.

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.

region

Default: `"us-east-1"`

The default AWS region to connect to.

max_pool_connections

Default: `10`

The maximum number of connections to keep in a connection pool.

extra_headers

Default: `None`

A dictionary of headers that should be added to every request. This is only useful when interfacing with DynamoDB through a proxy, where headers are stripped by the proxy before forwarding along. Failure to strip these headers before sending to AWS will result in an `InvalidSignatureException` due to request signing.

host

Default: automatically constructed by boto to account for region

The URL endpoint for DynamoDB. This can be used to use a local implementation of DynamoDB such as DynamoDB Local or dynalite.

2.15.2 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.

2.16 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.

2.16.1 Creating a connection

Creating a connection is simple:

```python
from pynamodb.connection import Connection

conn = Connection()
```

You can specify a different DynamoDB url:

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

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

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

2.16.2 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
        }
    ]
```

(continues on next page)

(continued from previous page)

```
}
>>> conn.update_table('table_name', **kwargs)
```

2.16.3 Modifying items

The low level API can perform item operations 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')
```

2.17 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.

If for some reason you can't use conventional AWS configuration methods, you can set the credentials in the Model Meta class:

```
from pynamodb.models import Model

class MyModel(Model):
    class Meta:
        aws_access_key_id = 'my_access_key_id'
        aws_secret_access_key = 'my_secret_access_key'
        aws_session_token = 'my_session_token' # Optional, only for temporary␣
↪credentials like those received when assuming a role
```

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

2.18 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:

```
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)
```

2.19 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.

2.19.1 Environment Setup

You'll need a python3 installation and a virtualenv. There are many ways to manage virtualenvs, but a minimal example is shown below.

```
$ virtualenv -p python3 venv && source venv/bin/activate
$ pip install -e .[signals] -r requirements-dev.txt
```

A java runtime is required to run the integration tests. After installing java, download and untar the mock dynamodb server like so:

```
$ wget --quiet http://dynamodb-local.s3-website-us-west-2.amazonaws.com/dynamodb_local_
↪latest.tar.gz -O /tmp/dynamodb_local_latest.tar.gz
$ tar -xzf /tmp/dynamodb_local_latest.tar.gz -C /tmp
```

Note that you may want to place files somewhere other than `/tmp`.

2.19.2 Running Tests

After installing requirements in environment setup and ensuring your venv is activated, unit tests are run with:

```
$ pytest tests/ -k "not ddblocal"
```

There are also a set of integration tests that require a local dynamodb server to be mocked.

```
$ java -Djava.library.path=/tmp/DynamoDBLocal_lib -jar /tmp/DynamoDBLocal.jar -inMemory -
↪port 8000
$ pytest tests/      # in another window
```

2.19.3 Backwards Compatibility

Particular care should be paid to backwards compatibility when making any change in PynamoDB, especially with attributes and serialization/deserialization. Consider data written with an older version of the library and whether it can still be read after upgrading.

Where possible, write logic to continue supporting older data for at least one major version to simplify the upgrade path. Where that's not possible, create a new version of the attribute with a different name and mark the old one as deprecated.

Outside of data compatibility, follow the usual semver rules for API changes and limit breaking changes to a major release.

2.19.4 The Scope of the Library

The purpose of this library is to provide a Pythonic ODM layer on top of DynamoDB to be used in server applications' runtime, i.e. to enable their various application logic and features. While striving for the library to be useful, we're also trying to "do one thing well". For this reason:

- Database administration tasks are out of scope, and while PynamoDB has functions for operations like CreateTable, CreateIndex and DeleteTable, it's because they are useful for interacting with dynamodb-local and moto's DynamoDB backend from within test code.

  For this reason, features such as enabling PITR backups, restoring from such backups, updating indices, etc. are intentionally absent. For getting started and operating on a small scale, AWS Console and the AWS Command Line Interface (awscli) can be used. For larger scale, infrastructure provisioning by dedicated tools (such as CloudFormation or Terraform) would be vastly preferable over anything PynamoDB could offer.

  Per security best practices, we recommend running your application's runtime with an IAM role having the least privileges necessary for it to function (which likely excludes any database administration operations).

- While the library aims to empower application developers, it steers away from high-level features which are not specific to DynamoDB. For example, a custom attribute which serializes UUIDs as strings is doubtlessly something many applications have had a need for, but as long as it doesn't exercise any core DynamoDB functionality (e.g. in the case of a UUID attribute, there isn't a dedicated DynamoDB data type or API feature for storing UUIDs), we would recommend relegating such functionality to auxiliary libraries. One such library is pynamodb-attributes.

2.19.5 Pull Requests

Pull requests should:

1. Specify an accurate title and detailed description of the change
2. Include thorough testing. Unit tests at a minimum, sometimes integration tests
3. Add test coverage for new code (CI will verify the delta)
4. Add type annotations to any code modified
5. Write documentation for new features
6. Maintain the existing code style (mostly PEP8) and patterns

2.19.6 Changelog

Any non-trivial change should be documented in the release notes. Please include sufficient detail in the PR description, which will be used by maintainers to populate the release notes.

2.19.7 Documentation

Docs are built using sphinx and the latest are available on readthedocs. A release of the *latest* tag (tracking master) happens automatically on merge via a GitHub webhook.

2.20 Release Notes

2.20.1 v6.0.2

Fixes:

- Fix a warning about *datetime.utcfromtimestamp* deprecation (#1261)

2.20.2 v6.0.1

Features:

- For failed transaction, return the underlying item in `cancellation_reasons[...].raw_item` (#1226). This only applies when passing `return_values=ALL_OLD`.

Fixes:

- Fixing (#1242) regression to the `extra_headers` feature. These headers are intended for proxies that strip them, so they should be excluded from the AWS signature.

2.20.3 v6.0.0

This is a major release and contains breaking changes. Please read the notes below carefully.

Breaking changes:

- *BinaryAttribute* and *BinarySetAttribute* have undergone breaking changes:
  - The attributes' internal encoding has changed. To prevent this change going unnoticed, a new required `legacy_encoding` parameter was added: see upgrading_binary for details. If your codebase uses *BinaryAttribute* or *BinarySetAttribute*, go over the attribute declarations and mark them accordingly.
  - When using binary attributes, the return value of *serialize()* will no longer be JSON-serializable since it will contain `bytes` objects. Use *to_dynamodb_dict()* and *to_simple_dict()* for JSON-serializable mappings. for a safe JSON-serializable representation.
- Python 3.6 is no longer supported.
- PynamoDB no longer has a default AWS region (used to be us-east-1) (#1003). If needed, update your models' *Meta* or set the *AWS_DEFAULT_REGION* environment variable.
- *Model*'s JSON serialization helpers were changed:
  - `to_json` was renamed to *to_simple_dict()* (#1126). Additionally, *to_dynamodb_dict()* and *from_dynamodb_dict()* were added for round-trip JSON serialization.
  - `pynamodb.util.attribute_value_to_json` was removed (#1126)
- *Attribute*'s `default` parameter must be either an immutable value (of one of the built-in immutable types) or a callable. This prevents a common class of errors caused by unintentionally mutating the default value. A simple workaround is to pass an initializer (e.g. change `default={}` to `default=dict`) or wrap in a lambda (e.g. change `default={'foo': 'bar'}` to `default=lambda: {'foo': 'bar'}`).
- *count()*, *query()*, and *scan()* are now instance methods.
- `OperationSettings` has been removed.

Major changes:

- We are now compatible with opentelemetry botocore instrumentation.
- We've reduced our usage of botocore private APIs (#1079). On multiple occasions, new versions of botocore broke PynamoDB, and this change lessens the likelihood of that happening in the future by reducing (albeit not eliminating) our reliance on private botocore APIs.

Minor changes:

- *save()*, *update()*, `delete_item()`, and *delete()* now accept a `add_version_condition` parameter. See *Conditioning on the version* for more details.
- *batch_get()*, has guard rails defending against items without a hash_key and range_key.
- `set()`, can remove attribute by assigning an empty value in the update expression.

2.20.4 v5.5.1

- Fix compatibility with botocore 1.33.2 (#1205)

2.20.5 v5.5.0

- `save()`, `update()`, `delete_item()`, and `delete()` now accept a `add_version_condition` parameter. See *Conditioning on the version* for more details.

2.20.6 v5.4.1

- Use model's AWS credentials in threads (#1164)

  A model can specify custom AWS credentials in the `Meta` class (in lieu of "global" AWS credentials from the environment). Previously those model-specific credentials were not used from within new threads.

Contributors to this release:

- @atsuoishimoto

2.20.7 v5.4.0

- Expose transaction cancellation reasons in `cancellation_reasons()` and `cancellation_reasons()` (#1144).

2.20.8 v5.3.2

- Prevent `typing_tests` from being installed into site-packages (#1118)

Contributors to this release:

- @musicinmybrain

2.20.9 v5.3.1

- Fixed issue introduced in 5.3.0: using `TableConnection` directly (not through a model) raised the following exception:

```
pynamodb.exceptions.TableError: Meta-table for '(table-name)' not initialized
```

- Fix typing on `TransactGet` (backport of #1057)

2.20.10 v5.3.0

- No longer call `DescribeTable` API before first operation

  Before this change, we would call `DescribeTable` before the first operation on a given table in order to discover its schema. This slowed down bootstrap (particularly important for lambdas), complicated testing and could potentially cause inconsistent behavior since queries were serialized using the table's (key) schema but deserialized using the model's schema.

  With this change, both queries and models now use the model's schema.

2.20.11 v5.2.3

- Update for botocore 1.28 private API change (#1087) which caused the following exception:

```
TypeError: Cannot mix str and non-str arguments
```

2.20.12 v5.2.2

- Update for botocore 1.28 private API change (#1083) which caused the following exception:

```
TypeError: _convert_to_request_dict() missing 1 required positional argument:
↪'endpoint_url'
```

2.20.13 v5.2.1

- Fix issue from 5.2.0 with attempting to set GSI provisioned throughput on PAY_PER_REQUEST billing mode (#1018)

2.20.14 v5.2.0

- The `IndexMeta` class has been removed. Now `type(Index) == type` (#998)
- JSON serialization support (`Model.to_json` and `Model.from_json`) has been added (#857)
- Improved type annotations for expressions and transactions (#951, #991)
- Always use Model attribute definitions in create table schema (#996)

2.20.15 v5.1.0

date
  2021-06-29

- Introduce `DynamicMapAttribute` to enable partially defining attributes on a `MapAttribute` (#868)
- Quality of life improvements: Type annotations, better comment, more resilient test (#934, #936, #948)
- Fix type annotation of `is_in` conditional expression (#947)
- Null errors should include full attribute path (#915)
- Fix for serializing and deserializing dates prior to year 1000 (#949)

2.20.16 v5.0.3

date
2021-02-14

This version has an unintentional breaking change:

- Propagate `Model.serialize`'s `null_check` parameter to nested MapAttributes (#908)

  Previously null errors (persisting `None` into an attribute defined as `null=False`) were ignored for attributes in map attributes that were nested in maps or lists. After upgrade, these will resulted in an *`AttributeNullError`* being raised.

2.20.17 v5.0.2

date
2021-02-11

- Do not serialize all attributes for updates and deletes (#905)

2.20.18 v5.0.1

date
2021-02-10

- Fix type errors when deriving from a MapAttribute and another type (#904)

2.20.19 v5.0.0

date
2021-01-26

This is major release and contains breaking changes. Please read the notes below carefully.

Breaking changes:

- Python 2 is no longer supported. Python 3.6 or greater is now required.
- *`UnicodeAttribute`* and *`BinaryAttribute`* now support empty values (#830)

  In previous versions, assigning an empty value to would be akin to assigning `None`: if the attribute was defined with `null=True` then it would be omitted, otherwise an error would be raised.

  As of May 2020, DynamoDB supports empty values for String and Binary attributes. This release of PynamoDB starts treating empty values like any other values. If existing code unintentionally assigns empty values to StringAttribute or BinaryAttribute, this may be a breaking change: for example, the code may rely on the fact that in previous versions empty strings would be "read back" as `None` values when reloaded from the database.
- *`UTCDateTimeAttribute`* now strictly requires the date string format `'%Y-%m-%dT%H:%M:%S.%f%z'` to ensure proper ordering. PynamoDB has always written values with this format but previously would accept reading other formats. Items written using other formats must be rewritten before upgrading.
- Table backup functionality (`Model.dump[s]` and `Model.load[s]`) has been removed.
- `Model.query` no longer converts unsupported range key conditions into filter conditions.
- Internal attribute type constants are replaced with their "short" DynamoDB version (#827)

- Remove `ListAttribute.remove_indexes` (added in v4.3.2) and document usage of remove for list elements (#838)
- Remove `pynamodb.connection.util.pythonic` (#753) and (#865)
- Remove `ModelContextManager` class (#861)

Features:

- **Polymorphism**

  This release introduces *Polymorphism* support via `DiscriminatorAttribute`. Discriminator values are written to DynamoDB and used during deserialization to instantiate the desired class.

- **Model Serialization**

  The `Model` class now includes public methods for serializing and deserializing its attributes. `Model.serialize` and `Model.deserialize` convert the model to/from a dictionary of DynamoDB attribute values.

Other changes in this release:

- Typed list attributes can now support any Attribute subclass (#833)
- Most API operation methods now accept a `settings` argument to customize settings of individual operations. This currently allow adding or overriding HTTP headers. (#887)
- Add the attribute name to error messages when deserialization fails (#815)
- Add the table name to error messages for transactional operations (#835)

Contributors to this release:

- @jpinner
- @ikonst
- @rchilaka-amzn
- @jonathantan

2.20.20 v4.4.1

- Fix compatibility with botocore 1.33.2 (#1235)

2.20.21 v4.4.0

- Update for botocore 1.28 private API change (#1130) which caused the following exception:

```
TypeError: _convert_to_request_dict() missing 1 required positional argument:
↪'endpoint_url'
```

2.20.22 v4.3.3

- Add type stubs for indexing into a `ListAttribute` for forming conditional expressions (#774)

```
class MyModel(Model):
  ...
  my_list = ListAttribute()

MyModel.query(..., condition=MyModel.my_list[0] == 42)
```

- Fix a warning about `collections.abc` deprecation (#782)

2.20.23 v4.3.2

- Fix discrepancy between runtime and type-checker's perspective of `Index` and derived types (#769)
- Add `ListAttribute.remove_indexes` action for removing specific indexes from a `ListAttribute` (#754)
- Type stub fixes:
  - Add missing parameters of `Model.scan` (#750)
  - Change `Model.get`'s `hash_key` parameter to be typed `Any` (#756)
- Prevent integration tests from being packaged (#758)
- Various documentation fixes (#762, #765, #766)

Contributors to this release:

- @mxr
- @sodre
- @biniow
- @MartinAltmayer
- @dotpmrcunha
- @meawoppl

2.20.24 v4.3.1

- Fix Index.query and Index.scan typing regressions introduced in 4.2.0, which were causing false errors in type checkers

2.20.25 v4.3.0

- Implement exponential backoff for batch writes (#728)
- Avoid passing 'PROVISIONED' BillingMode for compatibility with some AWS AZs (#721)
- On Python >= 3.3, use importlib instead of deprecated imp (#723)
- Update in-memory object correctly on `REMOVE` update expressions (#741)

Contributors to this release:

- @hallie

- @bit-bot-bit
- @edholland
- @reginalin
- @MichelML
- @timgates42
- @sunaoka
- @conjmurph

2.20.26 v4.2.0

date
2019-10-17

This is a backwards compatible, minor release.

- Add `attributes_to_get` parameter to `Model.scan` (#431)
- Disable botocore parameter validation for performance (#711)

Contributors to this release:

- @ButtaKnife

2.20.27 v4.1.0

date
2019-10-17

This is a backwards compatible, minor release.

- In the Model's Meta, you may now provide an AWS session token, which is mostly useful for assumed roles (#700):

```
sts_client = boto3.client("sts")
role_object = sts_client.assume_role(RoleArn=role_arn, RoleSessionName="role_name", ↪DurationSeconds=BOTO3_CLIENT_DURATION)
role_credentials = role_object["Credentials"]

class MyModel(Model):
  class Meta:
    table_name = "table_name"
    aws_access_key_id = role_credentials["AccessKeyId"]
    aws_secret_access_key = role_credentials["SecretAccessKey"]
    aws_session_token = role_credentials["SessionToken"]

  hash = UnicodeAttribute(hash_key=True)
  range = UnicodeAttribute(range_key=True)
```

- Fix warning about *inspect.getargspec* (#701)
- Fix provisioning GSIs when using pay-per-request billing (#690)
- Suppress Python 3 exception chaining when "re-raising" botocore errors as PynamoDB model exceptions (#705)

Contributors to this release:

- @asottile
- @julienduchesne

2.20.28 v4.0.0

date
2019-04-10

This is major release and contains breaking changes. Please read the notes below carefully.

Requests Removal

Given that `botocore` has moved to using `urllib3` directly for making HTTP requests, we'll be doing the same (via `botocore`). This means the following:

- The `session_cls` option is no longer supported.
- The `request_timeout_seconds` parameter is no longer supported. `connect_timeout_seconds` and `read_timeout_seconds` are available instead.
  - Note that the timeouts for connection and read are now `15` and `30` seconds respectively. This represents a change from the previous `60` second combined `requests` timeout.
- *Wrapped* exceptions (i.e `exc.cause`) that were from `requests.exceptions` will now be comparable ones from `botocore.exceptions` instead.

Key attribute types must match table

The previous release would call *DescribeTable* to discover table metadata and would use the key types as defined in the DynamoDB table. This could obscure type mismatches e.g. where a table's hash key is a number (*N*) in DynamoDB, but defined in PynamoDB as a *UnicodeAttribute*.

With this release, we're always using the PynamoDB model's definition of all attributes including the key attributes.

Deprecation of old APIs

Support for Legacy Conditional Parameters has been removed. See a complete list of affected `Model` methods below:

Method	Changes
`update_i`	removed in favor of `update`
`rate_lim`	removed in favor of `scan` and `ResultIterator`
`delete`	`conditional_operator` and `**expected_values` kwargs removed. Use `condition` instead.
`update`	`attributes`, `conditional_operator` and `**expected_values` kwargs removed. Use `actions` and `condition` instead.
`save`	`conditional_operator` and `**expected_values` kwargs removed. Use `condition` instead.
`count`	`**filters` kwargs removed. Use `range_key_condition`/`filter_condition` instead.
`query`	`conditional_operator` and `**filters` kwargs removed. Use `range_key_condition`/`filter_condition` instead.
`scan`	• `conditional_operator` and `**filters` kwargs removed. Use `filter_condition` instead. • `allow_rate_limited_scan_without_consumed_capacity` was removed

When upgrading, pay special attention to use of `**filters` and `**expected_values`, as you'll need to check for arbitrary names that correspond to attribute names. Also keep an eye out for kwargs like `user_id__eq=5` or `email__null=True`, which are no longer supported. A type check can help you catch cases like these.

New features in this release:

- Support for transactions (`TransactGet` and `TransactWrite`) (#618)
- Support for versioned optimistic locking (#664)

Other changes in this release:

- Python 2.6 is no longer supported. 4.x.x will be the last major release to support Python 2.7 given the upcoming EOL.
- Added the `max_pool_connection` and `extra_headers` settings to replace common use cases for `session_cls`
- Added support for moto through implementing the botocore "before-send" hook.
- Performance improvements to `UTCDateTimeAttribute` deserialization. (#610)
- The `MapAttributeMeta` class has been removed. Now `type(MapAttribute) == AttributeContainerMeta`.
- Removed `LegacyBooleanAttribute` and the read-compatibility for it in `BooleanAttribute`.
- *None* can now be used to bootstrap condition chaining (#653)
- Allow specifying timedeltas in expressions involving TTLAttributes (#665)

2.20.29 v3.4.1

date
2019-06-28

This is a backwards compatible, minor release.

Changes in this release:

- Fix type stubs to include new methods and parameters introduced with time-to-live support

2.20.30 v3.4.0

date
2019-06-13

This is a backwards compatible, minor release.

Changes in this release:

- Adds a TTLAttribute that specifies when items expire (#259)
- Enables time-to-live on a DynamoDB table if the corresponding model has a TTLAttribute
- Adds a default_for_new parameter for Attribute which is a default that applies to new items only

Contributors to this release:

- @irhkang
- @ikonst

2.20.31 v3.3.3

date
2019-01-15

This is a backwards compatible, minor release.

Fixes in this release:

- Legacy boolean attribute migration fix. (#538)
- Correctly package type stubs. (#585)

Contributors to this release:

- @vo-va

2.20.32 v3.3.2

date
2019-01-03

This is a backwards compatible, minor release.

Changes in this release:

- Built-in support for mypy type stubs, superseding those in python/typeshed. (#537)

2.20.33 v3.3.1

date
2018-08-30

This is a backwards compatible, minor bug fix release.

Fixes in this release:

- Clearer error message on missing consumed capacity during rate-limited scan. (#506)
- Python 3 compatibility in PageIterator. (#535)
- Proxy configuration changes in botocore>=1.11.0. (#531)

Contributors to this release:

- @ikonst
- @zetaben
- @ningirsu

2.20.34 v3.3.0

date
2018-05-09

This is a backwards compatible, major bug fix release.

New features in this release:

- Support scan operations on secondary indexes. (#141, #392)
- Support projections in model get function. (#337, #403)
- Handle values from keys when batch get returns unprocessed keys. (#252, #376)
- Externalizes AWS Credentials. (#426)
- Add migration support for LegacyBooleanAttribute. (#404, #405)
- Rate limited Page Iterator. (#481)

Fixes in this release:

- Thread-safe client creation in botocore. (#153, #393)
- Use attr.get_value(value) when deserialize. (#450)
- Skip null attributes post serialization for maps. (#455)
- Fix deserialization bug in BinaryAttribute and BinarySetAttribute. (#459, #480)
- Allow MapAttribute instances to be used as the RHS in expressions. (#488)
- Return the correct last_evaluated_key for limited queries/scans. (#406, #410)
- Fix exclusive_start_key getting lost in PageIterator. (#421)
- Add python 3.5 for Travis ci builds. (#437)

Contributors to this release:

- @jpinner-lyft
- @scode
- @behos
- @jmphilli
- @drewisme
- @nicysneiros
- @jcomo
- @kevgliss
- @asottile
- @harleyk
- @betamoo

2.20.35 v3.2.1

date
2017-10-25

This is a backwards compatible, minor bug fix release.

Removed features in this release:

- Remove experimental Throttle api. (#378)

Fixes in this release:

- Handle attributes that cannot be retrieved by getattr. Fixes #104 (#385)
- Model.refresh() should reset all model attribuets. Fixes #166 (#388)
- Model.loads() should deserialize using custom attribute names. Fixes #168 (#387)
- Deserialize hash key during table loads. Fixes #143 (#386)
- Support pagination in high-level api query and scan methods. Fixes #50, #118, #207, and #248 (#379)
- Don't serialize null nested attributed. Fixes #240 and #309 (#375)
- Legacy update item subset removal using DELETE operator. Fixes #132 (#374)

Contributors to this release:

- @jpinner-lyft

2.20.36 v3.2.0

date
2017-10-13

This is a backwards compatible, minor release.

This release updates PynamoDB to interact with Dynamo via the current version of Dynamo's API. Condition and update expressions can now be created from attributes and used in model operations. Legacy filter and attribute update keyword arguments have been deprecated. Using these arguments will cause a warning to be logged.

New features in this release:

- Add support for current version of DynamoDB API
- Improved `MapAttribute` item assignment and access.

Contributors to this release:

- @jpinner-lyft

2.20.37 v3.2.0rc2

date
2017-10-09

This is a backwards compatible, release candidate.

This release candidate allows dereferencing raw `MapAttributes` in condition expressions. It also improves `MapAttribute` assignment and access.

Contributors to this release:

- @jpinner-lyft

2.20.38 v3.2.0rc1

date
2017-09-22

This is a backwards compatible, release candidate.

This release candidate updates PynamoDB to interact with Dynamo via the current version of Dynamo's API. It deprecates some internal methods that were used to interact with Dynamo that are no longer relevant. If your project was calling those low level methods a warning will be logged.

New features in this release:

- Add support for current version of DynamoDB API

Contributors to this release:

- @jpinner-lyft

2.20.39 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

2.20.40 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. 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

2.20.41 v2.2.0

date
2017-10-25

This is a backwards compatible, minor release.

The purpose of this release is to prepare users to upgrade to v3.0.1+ (see issue #377 for details).

Pull request #294 removes the backwards compatible deserialization of UnicodeSetAttributes introduced in #151.

This release introduces a migration function on the Model class to help re-serialize any data that was written with v1.5.4 and below.

Temporary feature in this release:

- Model.fix_unicode_set_attributes() migration helper
- Model.needs_unicode_set_fix() migration helper

2.20.42 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

2.20.43 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

2.20.44 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

2.20.45 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

2.20.46 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

2.20.47 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

2.20.48 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

2.20.49 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

2.20.50 v1.5.4

date
2017-10-25

This is a backwards compatible, minor bug fix release.

The purpose of this release is to prepare users to upgrade to v1.6.0+ (see issue #377 for details).

Pull request #151 introduces a backwards incompatible change to how UnicodeSetAttributes are serialized. While the commit attempts to provide compatibility by deserializing values written with v1.5.3 and below, it prevents users from upgrading because it starts writing non JSON-encoded values to dynamo.

Anyone using UnicodeSetAttribute must first deploy this version.

Fixes in this release:

- Backport UnicodeSetAttribute deserialization code from #151

2.20.51 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

2.20.52 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

2.20.53 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

2.20.54 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

2.20.55 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)

2.20.56 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

2.20.57 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.

2.20.58 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).

2.20.59 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)

2.20.60 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)

2.20.61 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)`

2.20.62 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

2.20.63 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.

2.20.64 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

2.20.65 v1.3.2

date
2014-7-02

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

2.20.66 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.

2.20.67 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)

2.20.68 v1.2.2

date
2014-05-14

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

2.20.69 v1.2.1

date
2014-05-07

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

2.20.70 v1.2.0

date
2014-05-06

- Numerous documentation improvements
- Improved support for conditional operations
- Added support for filtering queries on non key attributes (https://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

2.20.71 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')
```

2.20.72 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:

```python
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:

```python
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

2.20.73 v0.1.13

date
2014-03-20

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

2.20.74 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.

2.20.75 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

2.21 Versioning Scheme

PynamoDB conforms to PEP 440. Generally, 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.

2.22 Upgrading

This file complements the *release notes*, focusing on helping safe upgrades of the library in production scenarios.

2.22.1 PynamoDB 5.x to 6.x

BinaryAttribute is no longer double base64-encoded

See upgrading_binary for details.

2.22.2 PynamoDB 4.x to 5.x

Null checks enforced where they weren't previously

Previously null errors (persisting `None` into an attribute defined as `null=False`) were ignored inside **nested** map attributes, e.g.

```
from pynamodb.models import Model
from pynamodb.attributes import ListAttribute, MapAttribute, UnicodeAttribute

class Employee(MapAttribute):
```

(continues on next page)

(continued from previous page)

```
  name = UnicodeAttribute(null=False)

class Team(Model):
  employees = ListAttribute(of=Employee)


team = Team()
team.employees = [Employee(name=None)]
team.save()  # this will raise now
```

Now these will resulted in an *AttributeNullError* being raised.

This was an unintentional breaking change introduced in 5.0.3.

Empty values are now meaningful

UnicodeAttribute and *BinaryAttribute* now support empty values (#830)

In previous versions, assigning an empty value to would be akin to assigning `None`: if the attribute was defined with `null=True` then it would be omitted, otherwise an error would be raised. DynamoDB added support empty values for String and Binary attributes. This release of PynamoDB starts treating empty values like any other values. If existing code unintentionally assigns empty values to StringAttribute or BinaryAttribute, this may be a breaking change: for example, the code may rely on the fact that in previous versions empty strings would be "read back" as `None` values when reloaded from the database.

No longer parsing date-time strings leniently

UTCDateTimeAttribute now strictly requires the date string format `'%Y-%m-%dT%H:%M:%S.%f%z'` to ensure proper ordering. PynamoDB has always written values with this format but previously would accept reading other formats. Items written using other formats must be rewritten before upgrading.

Removed functionality

The following changes are breaking but are less likely to go unnoticed:

- Python 2 is no longer supported. Python 3.6 or greater is now required.
- Table backup functionality (`Model.dump[s]` and `Model.load[s]`) has been removed.
- `Model.query` no longer converts unsupported range key conditions into filter conditions.
- Internal attribute type constants are replaced with their "short" DynamoDB version (#827)
- Remove `ListAttribute.remove_indexes` (added in v4.3.2) and document usage of remove for list elements (#838)
- Remove `pynamodb.connection.util.pythonic` (#753) and (#865)
- Remove `ModelContextManager` class (#861)

2.22.3 PynamoDB 3.x to 4.x

Requests Removal

Given that `botocore` has moved to using `urllib3` directly for making HTTP requests, we'll be doing the same (via `botocore`). This means the following:

- The `session_cls` option is no longer supported.
- The `request_timeout_seconds` parameter is no longer supported. `connect_timeout_seconds` and `read_timeout_seconds` are available instead.
  - Note that the timeouts for connection and read are now `15` and `30` seconds respectively. This represents a change from the previous `60` second combined `requests` timeout.
- *Wrapped* exceptions (i.e `exc.cause`) that were from `requests.exceptions` will now be comparable ones from `botocore.exceptions` instead.

Key attribute types must match table

The previous release would call *DescribeTable* to discover table metadata and would use the key types as defined in the DynamoDB table. This could obscure type mismatches e.g. where a table's hash key is a number (*N*) in DynamoDB, but defined in PynamoDB as a *UnicodeAttribute*.

With this release, we're always using the PynamoDB model's definition of all attributes including the key attributes.

Deprecation of old APIs

Support for Legacy Conditional Parameters has been removed. See a complete list of affected `Model` methods below:

Method	Changes
`update_i`	removed in favor of `update`
`rate_lim`	removed in favor of `scan` and `ResultIterator`
`delete`	`conditional_operator` and `**expected_values` kwargs removed. Use `condition` instead.
`update`	`attributes`, `conditional_operator` and `**expected_values` kwargs removed. Use `actions` and `condition` instead.
`save`	`conditional_operator` and `**expected_values` kwargs removed. Use `condition` instead.
`count`	`**filters` kwargs removed. Use `range_key_condition`/`filter_condition` instead.
`query`	`conditional_operator` and `**filters` kwargs removed. Use `range_key_condition`/`filter_condition` instead.
`scan`	• `conditional_operator` and `**filters` kwargs removed. Use `filter_condition` instead. • `allow_rate_limited_scan_without_consumed_capacity` was removed

When upgrading, pay special attention to use of `**filters` and `**expected_values`, as you'll need to check for arbitrary names that correspond to attribute names. Also keep an eye out for kwargs like `user_id__eq=5` or `email__null=True`, which are no longer supported. A type check can help you catch cases like these.

2.22.4 PynamoDB 2.x to 3.x

Changes to UnicodeSetAttribute

See upgrading_unicodeset for details.

CHAPTER

THREE

API DOCS

3.1 API

3.1.1 High Level API

DynamoDB Models for PynamoDB

class pynamodb.models.**BatchWrite**(*model: Type[_T]*, *auto_commit: bool = True*)

A class for batch writes

commit() → None

Writes all of the changes that are pending

delete(*del_item: _T*) → None

This adds *del_item* to the list of pending operations to be performed.

If the list currently contains 25 items, which is the DynamoDB imposed limit on a BatchWriteItem call, one of two things will happen. If auto_commit is True, a BatchWriteItem operation will be sent with the already pending operations after which put_item is appended to the (now empty) list. If auto_commit is False, ValueError is raised to indicate additional items cannot be accepted due to the DynamoDB imposed limit.

Parameters
del_item – Should be an instance of a *Model* to be deleted

save(*put_item: _T*) → None

This adds *put_item* to the list of pending operations to be performed.

If the list currently contains 25 items, which is the DynamoDB imposed limit on a BatchWriteItem call, one of two things will happen. If auto_commit is True, a BatchWriteItem operation will be sent with the already pending writes after which put_item is appended to the (now empty) list. If auto_commit is False, ValueError is raised to indicate additional items cannot be accepted due to the DynamoDB imposed limit.

Parameters
put_item – Should be an instance of a *Model* to be written

class pynamodb.models.**MetaModel**(*name*, *bases*, *namespace*, *discriminator=None*)

Model meta class

class pynamodb.models.**MetaProtocol**(**args*, ***kwargs*)

class pynamodb.models.**Model**(*hash_key: Any | None = None*, *range_key: Any | None = None*, *_user_instantiated: bool = True*, ***attributes: Any*)

Defines a *PynamoDB* Model

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

exception DoesNotExist(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when an item queried does not exist

classmethod batch_get(*items: Iterable[Any | Iterable[Any]]*, *consistent_read: bool | None = None*, *attributes_to_get: Sequence[str] | None = None*) → Iterator[_T]

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 batch_write(*auto_commit: bool = True*) → *BatchWrite*[_T]

Returns a BatchWrite context manager for a batch operation.

Parameters

auto_commit – If true, the context manager will commit writes incrementally as items are written to as necessary to honor item count limits in the DynamoDB API (see BatchWrite). Regardless of the value passed here, changes automatically commit on context exit (whether successful or not).

classmethod count(*hash_key: Any | None = None*, *range_key_condition: Condition | None = None*, *filter_condition: Condition | None = None*, *consistent_read: bool = False*, *index_name: str | None = None*, *limit: int | None = None*, *rate_limit: float | None = None*) → int

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
- **rate_limit** – If set then consumed capacity will be limited to this amount per second

classmethod create_table(*wait: bool = False*, *read_capacity_units: int | None = None*, *write_capacity_units: int | None = None*, *billing_mode: str | None = None*, *ignore_update_ttl_errors: bool = False*) → Any

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
- **billing_mode** – Sets the billing mode 'PROVISIONED' (default) or 'PAY_PER_REQUEST' for this table

delete(*condition: Condition | None = None*, ***, *add_version_condition: bool = True*) → Any

Deletes this object from DynamoDB.

Parameters

add_version_condition – For models which have a *VersionAttribute*, specifies whether the item should only be deleted if its current version matches the expected one. Set to *False* for a 'delete anyway' strategy.

Raises

`pynamodb.exceptions.DeleteError` – If the record can not be deleted

classmethod delete_table() → Any

Delete the table for this model

classmethod describe_table() → Any

Returns the result of a DescribeTable operation on this model's table

deserialize(*attribute_values: Dict[str, Dict[str, Any]]*) → None

Deserializes a model from botocore's DynamoDB client.

classmethod exists() → bool

Returns True if this table exists, False otherwise

classmethod from_raw_data(*data: Dict[str, Any]*) → _T

Returns an instance of this class from the raw data

Parameters

data – A serialized DynamoDB object

classmethod get(*hash_key: Any*, *range_key: Any | None = None*, *consistent_read: bool = False*, *attributes_to_get: Sequence[str] | None = None*) → _T

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.
- **consistent_read**
- **attributes_to_get**

Raises

`ModelInstance.DoesNotExist` – if the object to be updated does not exist

classmethod query(*hash_key: Any*, *range_key_condition: Condition | None = None*, *filter_condition: Condition | None = None*, *consistent_read: bool = False*, *index_name: str | None = None*, *scan_index_forward: bool | None = None*, *limit: int | None = None*, *last_evaluated_key: Dict[str, Dict[str, Any]] | None = None*, *attributes_to_get: Iterable[str] | None = None*, *page_size: int | None = None*, *rate_limit: float | None = None*) → *ResultIterator*[_T]

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
- **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
- **rate_limit** – If set then consumed capacity will be limited to this amount per second

refresh(*consistent_read: bool = False*) → None

Retrieves this object's data from dynamodb and syncs this local object

Parameters

- **consistent_read** – If True, then a consistent read is performed.
- **settings** – per-operation settings

Raises

ModelInstance.DoesNotExist – if the object to be updated does not exist

save(*condition: Condition | None = None*, ***, *add_version_condition: bool = True*) → Dict[str, Any]

Save this object to dynamodb

classmethod scan(*filter_condition: Condition | None = None*, *segment: int | None = None*, *total_segments: int | None = None*, *limit: int | None = None*, *last_evaluated_key: Dict[str, Dict[str, Any]] | None = None*, *page_size: int | None = None*, *consistent_read: bool | None = None*, *index_name: str | None = None*, *rate_limit: float | None = None*, *attributes_to_get: Sequence[str] | None = None*) → *ResultIterator*[_T]

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
- **last_evaluated_key** – If set, provides the starting point for scan.
- **page_size** – Page size of the scan to DynamoDB
- **consistent_read** – If True, a consistent read is performed
- **index_name** – If set, then this index is used
- **rate_limit** – If set then consumed capacity will be limited to this amount per second
- **attributes_to_get** – If set, specifies the properties to include in the projection expression

serialize(*null_check: bool = True*) → Dict[str, Dict[str, Any]]

Serializes a model for botocore's DynamoDB client.

Warning: BINARY and BINARY_SET attributes (whether top-level or nested) serialization would contain `bytes` objects which are not JSON-serializable by the `json` module.

Use *to_dynamodb_dict()* and *to_simple_dict()* for JSON-serializable mappings.

update(*actions: List[Action]*, *condition: Condition | None = None*, ***, *add_version_condition: bool = True*) → Any

Updates an item using the UpdateItem operation.

Parameters

- **`actions`** – a list of Action updates to apply
- **`condition`** – an optional Condition on which to update
- **`settings`** – per-operation settings
- **`add_version_condition`** – For models which have a `VersionAttribute`, specifies whether only to update if the version matches the model that is currently loaded. Set to *False* for a 'last write wins' strategy. Regardless, the version will always be incremented to prevent "rollbacks" by concurrent `save()` calls.

Raises

- **`ModelInstance.DoesNotExist`** – if the object to be updated does not exist
- **`pynamodb.exceptions.UpdateError`** – if the *condition* is not met

`classmethod` **`update_ttl`**(*ignore_update_ttl_errors: bool*) → None

Attempt to update the TTL on the table. Certain implementations (eg: dynalite) do not support updating TTLs and will fail.

PynamoDB attributes

`class pynamodb.attributes.`**`Attribute`**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

An attribute of a model or an index.

Parameters

- **`hash_key`** – If *True*, this attribute is a model's or an index's hash key (partition key).
- **`range_key`** – If *True*, this attribute is a model's or an index's range key (sort key).
- **`null`** – If *True*, a *None* value would be considered valid and would result in the attribute not being set in the underlying DynamoDB item. If *False* (default), an exception will be raised when the attribute is persisted with a *None* value.

  Note: This is different from `pynamodb.attributes.NullAttribute`, which manifests in a *NULL*-typed DynamoDB attribute value.

- **`default`** – A default value that will be assigned in new models (when they are initialized) and existing models (when they are loaded).

  Note: Starting with PynamoDB 6.0, the default must be either an immutable value (of one of the built-in immutable types) or a callable. This prevents a common class of errors caused by unintentionally mutating the default value. A simple workaround is to pass an initializer (e.g. change `default={}` to `default=dict`) or wrap in a lambda (e.g. change `default={'foo': 'bar'}` to `default=lambda: {'foo': 'bar'}`).

- **`default_for_new`** – Like *default*, but used only for new models. Use this to assign a default for new models that you don't want to apply to existing models when they are loaded and then re-saved.

Note: Starting with PynamoDB 6.0, the default must be either an immutable value (of one of the built-in immutable types) or a callable.

- **attr_name** – The name that is used for the attribute in the underlying DynamoDB item; use this to assign a "pythonic" name that is different from the persisted name, i.e.

```
number_of_threads = NumberAttribute(attr_name='thread_count')
```

deserialize(*value: Any*) → Any

Deserializes a value from botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

serialize(*value: Any*) → Any

Serializes a value for botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

class pynamodb.attributes.**AttributeContainer**(*_user_instantiated: bool = True*, ***attributes: Attribute*)

Base class for models and maps.

from_dynamodb_dict(*d: Dict[str, Dict[str, Any]]*) → None

Sets attributes from a mapping previously produced by `to_dynamodb_dict()`.

from_simple_dict(*d: Dict[str, Any]*) → None

Sets attributes from a mapping previously produced by `to_simple_dict()`.

classmethod **get_attributes**() → Dict[str, *Attribute*]

Returns the attributes of this class as a mapping from *python_attr_name* => *attribute*.

to_dynamodb_dict() → Dict[str, Dict[str, Any]]

Returns the contents of this instance as a JSON-serializable mapping, where each attribute is represented as a mapping with the attribute type as the key and the attribute value as the value, e.g.

```
{
    "id": {
        "N": "12345"
    },
    "name": {
        "S": "Alice"
    },
}
```

This matches the structure of the "DynamoDB" JSON mapping in the AWS Console.

to_simple_dict(**, force: bool = False*) → Dict[str, Any]

Returns the contents of this instance as a simple JSON-serializable mapping.

```
{
    "id": 12345,
    "name": "Alice",
}
```

This matches the structure of the "normal" JSON mapping in the AWS Console.

Note: This representation is limited: by default, it cannot represent binary or set attributes, as their encoded form is indistinguishable from a string or list attribute respectively (and therefore ambiguous).

Parameters
`force` – If `True`, force the conversion even if the model contains Binary or Set attributes If `False`, a `ValueError` will be raised if such attributes are set.

`class pynamodb.attributes.`**`BinaryAttribute`**`(`**args: Any*, *legacy_encoding: bool*, ***kwargs: Any*`)`

An attribute containing a binary data object (`bytes`).

Parameters
`legacy_encoding` – If `True`, inefficient legacy encoding will be used to maintain compatibility with PynamoDB 5 and lower. Set to `False` for new tables and models, and always set to `False` within *`MapAttribute`*.

For more details, see upgrading_binary.

`deserialize``(`*value*`)`

Deserializes a value from botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

`serialize``(`*value*`)`

Serializes a value for botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

`class pynamodb.attributes.`**`BinarySetAttribute`**`(`**args: Any*, *legacy_encoding: bool*, ***kwargs: Any*`)`

An attribute containing a set of binary data objects (`bytes`).

Parameters
`legacy_encoding` – If `True`, inefficient legacy encoding will be used to maintain compatibility with PynamoDB 5 and lower. Set to `False` for new tables and models, and always set to `False` within *`MapAttribute`*.

For more details, see upgrading_binary.

`deserialize``(`*value*`)`

Returns a set of decoded byte strings from base64 encoded values.

`serialize``(`*value*`)`

Returns a list of base64 encoded binary strings. Encodes empty sets as "None".

`class pynamodb.attributes.`**`BooleanAttribute`**`(`*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*`)`

A class for boolean attributes

`deserialize``(`*value*`)`

Deserializes a value from botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

serialize(*value*)

Serializes a value for botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

class pynamodb.attributes.**DiscriminatorAttribute**(*attr_name: str | None = None*)

deserialize(*value*)

Returns the class corresponding to the given discriminator value.

serialize(*value*)

Returns the discriminator value corresponding to the given class.

class pynamodb.attributes.**DynamicMapAttribute**(***attributes*)

A map attribute that supports declaring attributes (like an AttributeContainer) but will also store any other values that are set on it (like a raw MapAttribute).

```
>>> class MyDynamicMapAttribute(DynamicMapAttribute):
>>>     a_date_time = UTCDateTimeAttribute()  # raw map attributes cannot serialize/
↪deserialize datetime values
>>>
>>> dynamic_map = MyDynamicMapAttribute()
>>> dynamic_map.a_date_time = datetime.utcnow()
>>> dynamic_map.a_number = 5
>>> dynamic_map.serialize()  # {'a_date_time': {'S': 'xxx'}, 'a_number': {'N': '5'}}
```

deserialize(*values*)

Decode as a dict.

serialize(*values*, ***, *null_check: bool = True*)

Serializes a value for botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

class pynamodb.attributes.**JSONAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

A JSON Attribute

Encodes JSON to unicode internally

deserialize(*value*)

Deserializes JSON

serialize(*value*) → str | None

Serializes JSON to unicode

class pynamodb.attributes.**ListAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: Any | Callable[[...], Any] | None = None*, *attr_name: str | None = None*, *of: Type[_T] | None = None*)

deserialize(*values*)

Decode from list of AttributeValue types.

`serialize`(*values*, *, *null_check: bool = True*)

Encode the given list of objects into a list of AttributeValue types.

`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=dict)

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? 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.

serialize(*values*, ***, *null_check: bool = True*)

Serializes a value for botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

class pynamodb.attributes.**NullAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

deserialize(*value*)

Deserializes a value from botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

serialize(*value*)

Serializes a value for botocore's DynamoDB client.

For a list of DynamoDB attribute types and their matching botocore Python types, see DynamoDB.Client.get_item API reference.

class pynamodb.attributes.**NumberAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

A number attribute

deserialize(*value*)

Decode numbers from JSON

serialize(*value*)

Encode numbers as JSON

class pynamodb.attributes.**NumberSetAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

A number set attribute

deserialize(*value*)

Returns a set from a JSON list of numbers.

serialize(*value*)

Encodes a set of numbers as a JSON list. Encodes empty sets as "None".

class pynamodb.attributes.**TTLAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

A time-to-live attribute that signifies when the item expires and can be automatically deleted. It can be assigned with a timezone-aware datetime value (for absolute expiry time) or a timedelta value (for expiry relative to the current time), but always reads as a UTC datetime value.

deserialize(*value*)

Deserializes a timestamp (Unix time) as a UTC datetime.

serialize(*value*)

Serializes a datetime as a timestamp (Unix time).

class pynamodb.attributes.**UTCDateTimeAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = 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: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

A unicode attribute

class pynamodb.attributes.**UnicodeSetAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

A unicode set

deserialize(*value*)

Returns a set from a list of strings.

serialize(*value*)

Returns a list of strings. Encodes empty sets as "None".

class pynamodb.attributes.**VersionAttribute**(*hash_key: bool = False*, *range_key: bool = False*, *null: bool | None = None*, *default: _T | Callable[[...], _T] | None = None*, *default_for_new: Any | Callable[[...], _T] | None = None*, *attr_name: str | None = None*)

A number attribute that implements *optimistic locking*.

deserialize(*value*)

Decode numbers from JSON and cast to int.

serialize(*value*)

Cast value to int then encode as JSON

PynamoDB Indexes

class pynamodb.indexes.**AllProjection**

An ALL projection

class pynamodb.indexes.**GlobalSecondaryIndex**

A global secondary index

class pynamodb.indexes.**IncludeProjection**(*non_attr_keys: List[str] | None = None*)

An INCLUDE projection

`class pynamodb.indexes.`**`Index`**

Base class for secondary indexes

`count`(*hash_key: Any*, *range_key_condition: Condition | None = None*, *filter_condition: Condition | None = None*, *consistent_read: bool = False*, *limit: int | None = None*, *rate_limit: float | None = None*) → int

Count on an index

`query`(*hash_key: Any*, *range_key_condition: Condition | None = None*, *filter_condition: Condition | None = None*, *consistent_read: bool = False*, *scan_index_forward: bool | None = None*, *limit: int | None = None*, *last_evaluated_key: Dict[str, Dict[str, Any]] | None = None*, *attributes_to_get: List[str] | None = None*, *page_size: int | None = None*, *rate_limit: float | None = None*) → *ResultIterator*[_M]

Queries an index

`scan`(*filter_condition: Condition | None = None*, *segment: int | None = None*, *total_segments: int | None = None*, *limit: int | None = None*, *last_evaluated_key: Dict[str, Dict[str, Any]] | None = None*, *page_size: int | None = None*, *consistent_read: bool | None = None*, *rate_limit: float | None = None*, *attributes_to_get: List[str] | None = None*) → *ResultIterator*[_M]

Scans an index

`class pynamodb.indexes.`**`KeysOnlyProjection`**

Keys only projection

`class pynamodb.indexes.`**`LocalSecondaryIndex`**

A local secondary index

`class pynamodb.indexes.`**`Projection`**

A class for presenting projections

`class pynamodb.transactions.`**`TransactGet`**(**args: Any*, ***kwargs: Any*)

`get`(*model_cls: Type[_M]*, *hash_key: Any*, *range_key: Any | None = None*) → _ModelFuture[_M]

Adds the operation arguments for an item to list of models to get returns a _ModelFuture object as a placeholder

Parameters

- **`model_cls`**
- **`hash_key`**
- **`range_key`**

Returns

`class pynamodb.transactions.`**`TransactWrite`**(*client_request_token: str | None = None*, *return_item_collection_metrics: str | None = None*, ***kwargs: Any*)

`class pynamodb.transactions.`**`Transaction`**(*connection: Connection*, *return_consumed_capacity: str | None = None*)

Base class for a type of transaction operation

`class pynamodb.pagination.`**`PageIterator`**(*operation: Callable*, *args: Any*, *kwargs: Dict[str, Any]*, *rate_limit: float | None = None*)

PageIterator handles Query and Scan result pagination.

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.Pagination.html https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Scan.html#Scan.Pagination `class pynamodb.pagination.`**`RateLimiter`**`(`*rate_limit: float*, *time_module: Any | None = None*`)`

RateLimiter limits operations to a pre-set rate of units/seconds

Example:

Initialize a RateLimiter with the desired rate
rate_limiter = RateLimiter(rate_limit)

Now, every time before calling an operation, call acquire()
rate_limiter.acquire()

And after an operation, update the number of units consumed
rate_limiter.consume(units)

`acquire``()` → None

Sleeps the appropriate amount of time to follow the rate limit restriction

Returns
None

`consume``(`*units: int*`)` → None

Records the amount of units consumed.

Parameters
`units` – Number of units consumed

Returns
None

`property rate_limit: float`

A limit of units per seconds

`class pynamodb.pagination.`**`ResultIterator`**`(`*operation: Callable*, *args: Any*, *kwargs: Dict[str, Any]*, *map_fn: Callable | None = None*, *limit: int | None = None*, *rate_limit: float | None = None*`)`

ResultIterator handles Query and Scan item pagination.

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.Pagination.html https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Scan.html#Scan.Pagination

3.1.2 Low Level API

PynamoDB lowest level connection

`class pynamodb.connection.`**`Connection`**`(`*region: str | None = None*, *host: str | None = None*, *read_timeout_seconds: float | None = None*, *connect_timeout_seconds: float | None = None*, *max_retry_attempts: int | None = None*, *max_pool_connections: int | None = None*, *extra_headers: Mapping[str, str] | None = None*, *aws_access_key_id: str | None = None*, *aws_secret_access_key: str | None = None*, *aws_session_token: str | None = None*`)`

A higher level abstraction over botocore

`add_meta_table``(`*meta_table: MetaTable*`)` → None

Adds information about the table's schema.

`batch_get_item``(`*table_name: str*, *keys: Sequence[str]*, *consistent_read: bool | None = None*, *return_consumed_capacity: str | None = None*, *attributes_to_get: Any | None = None*`)` → Dict

Performs the batch get item operation

`batch_write_item`(*table_name: str*, *put_items: Any | None = None*, *delete_items: Any | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*) → Dict

Performs the batch_write_item operation

`property client: BotocoreBaseClientPrivate`

Returns a botocore dynamodb client

`create_table`(*table_name: str*, *attribute_definitions: Any | None = None*, *key_schema: Any | None = None*, *read_capacity_units: int | None = None*, *write_capacity_units: int | None = None*, *global_secondary_indexes: Any | None = None*, *local_secondary_indexes: Any | None = None*, *stream_specification: Dict | None = None*, *billing_mode: str = 'PROVISIONED'*, *tags: Dict[str, str] | None = None*) → Dict

Performs the CreateTable operation

`delete_item`(*table_name: str*, *hash_key: str*, *range_key: str | None = None*, *condition: Condition | None = None*, *return_values: str | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*) → Dict

Performs the DeleteItem operation and returns the result

`delete_table`(*table_name: str*) → Dict

Performs the DeleteTable operation

`describe_table`(*table_name: str*) → Dict

Performs the DescribeTable operation

`dispatch`(*operation_name: str*, *operation_kwargs: Dict*) → Dict

Dispatches *operation_name* with arguments *operation_kwargs*

Raises TableDoesNotExist if the specified table does not exist

`get_attribute_type`(*table_name: str*, *attribute_name: str*, *value: Any | None = None*) → str

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_consumed_capacity_map`(*return_consumed_capacity: str*) → Dict

Builds the consumed capacity map that is common to several operations

`get_exclusive_start_key_map`(*table_name: str*, *exclusive_start_key: str*) → Dict

Builds the exclusive start key attribute map

`get_identifier_map`(*table_name: str*, *hash_key: str*, *range_key: str | None = None*, *key: str = 'Key'*) → Dict

Builds the identifier map that is common to several operations

`get_item`(*table_name: str*, *hash_key: str*, *range_key: str | None = None*, *consistent_read: bool = False*, *attributes_to_get: Any | None = None*) → Dict

Performs the GetItem operation and returns the result

`get_item_attribute_map`(*table_name: str*, *attributes: Any*, *item_key: str = 'Item'*, *pythonic_key: bool = True*) → Dict

Builds up a dynamodb compatible AttributeValue map `get_item_collection_map`(*return_item_collection_metrics: str*) → Dict

Builds the item collection map

`get_meta_table`(*table_name: str*) → MetaTable

Returns information about the table's schema.

`get_return_values_map`(*return_values: str*) → Dict

Builds the return values map that is common to several operations

`get_return_values_on_condition_failure_map`(*return_values_on_condition_failure: str*) → Dict

Builds the return values map that is common to several operations

`list_tables`(*exclusive_start_table_name: str | None = None*, *limit: int | None = None*) → Dict

Performs the ListTables operation

`parse_attribute`(*attribute: Any*, *return_type: bool = False*) → Any

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: str*, *hash_key: str*, *range_key: str | None = None*, *attributes: Any | None = None*, *condition: Condition | None = None*, *return_values: str | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*) → Dict

Performs the PutItem operation and returns the result

`query`(*table_name: str*, *hash_key: str*, *range_key_condition: Condition | None = None*, *filter_condition: Any | None = None*, *attributes_to_get: Any | None = None*, *consistent_read: bool = False*, *exclusive_start_key: Any | None = None*, *index_name: str | None = None*, *limit: int | None = None*, *return_consumed_capacity: str | None = None*, *scan_index_forward: bool | None = None*, *select: str | None = None*) → Dict

Performs the Query operation and returns the result

`scan`(*table_name: str*, *filter_condition: Any | None = None*, *attributes_to_get: Any | None = None*, *limit: int | None = None*, *return_consumed_capacity: str | None = None*, *exclusive_start_key: str | None = None*, *segment: int | None = None*, *total_segments: int | None = None*, *consistent_read: bool | None = None*, *index_name: str | None = None*) → Dict

Performs the scan operation

`property session: Session`

Returns a valid botocore session

`transact_get_items`(*get_items: Sequence[Dict]*, *return_consumed_capacity: str | None = None*) → Dict

Performs the TransactGet operation and returns the result

`transact_write_items`(*condition_check_items: Sequence[Dict]*, *delete_items: Sequence[Dict]*, *put_items: Sequence[Dict]*, *update_items: Sequence[Dict]*, *client_request_token: str | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*) → Dict

Performs the TransactWrite operation and returns the result

`update_item`(*table_name: str*, *hash_key: str*, *range_key: str | None = None*, *actions: Sequence[Action] | None = None*, *condition: Condition | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*, *return_values: str | None = None*) → Dict

Performs the UpdateItem operation

update_table(*table_name: str*, *read_capacity_units: int | None = None*, *write_capacity_units: int | None = None*, *global_secondary_index_updates: Any | None = None*) → Dict

Performs the UpdateTable operation

update_time_to_live(*table_name: str*, *ttl_attribute_name: str*) → Dict

Performs the UpdateTimeToLive operation

class pynamodb.connection.**TableConnection**(*table_name: str*, *region: str | None = None*, *host: str | None = None*, *connect_timeout_seconds: float | None = None*, *read_timeout_seconds: float | None = None*, *max_retry_attempts: int | None = None*, *max_pool_connections: int | None = None*, *extra_headers: Mapping[str, str] | None = None*, *aws_access_key_id: str | None = None*, *aws_secret_access_key: str | None = None*, *aws_session_token: str | None = None*, ***, *meta_table: MetaTable | None = None*)

A higher level abstraction over botocore

batch_get_item(*keys: Sequence[str]*, *consistent_read: bool | None = None*, *return_consumed_capacity: str | None = None*, *attributes_to_get: Any | None = None*) → Dict

Performs the batch get item operation

batch_write_item(*put_items: Any | None = None*, *delete_items: Any | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*) → Dict

Performs the batch_write_item operation

create_table(*attribute_definitions: Any | None = None*, *key_schema: Any | None = None*, *read_capacity_units: int | None = None*, *write_capacity_units: int | None = None*, *global_secondary_indexes: Any | None = None*, *local_secondary_indexes: Any | None = None*, *stream_specification: Dict | None = None*, *billing_mode: str = 'PROVISIONED'*, *tags: Dict[str, str] | None = None*) → Dict

Performs the CreateTable operation and returns the result

delete_item(*hash_key: str*, *range_key: str | None = None*, *condition: Condition | None = None*, *return_values: str | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*) → Dict

Performs the DeleteItem operation and returns the result

delete_table() → Dict

Performs the DeleteTable operation and returns the result

describe_table() → Dict

Performs the DescribeTable operation and returns the result

get_item(*hash_key: str*, *range_key: str | None = None*, *consistent_read: bool = False*, *attributes_to_get: Any | None = None*) → Dict

Performs the GetItem operation and returns the result

get_meta_table() → MetaTable

Returns a MetaTable

put_item(*hash_key: str*, *range_key: str | None = None*, *attributes: Any | None = None*, *condition: Condition | None = None*, *return_values: str | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*) → Dict

Performs the PutItem operation and returns the result

query(*hash_key: str*, *range_key_condition: Condition | None = None*, *filter_condition: Any | None = None*, *attributes_to_get: Any | None = None*, *consistent_read: bool = False*, *exclusive_start_key: Any | None = None*, *index_name: str | None = None*, *limit: int | None = None*, *return_consumed_capacity: str | None = None*, *scan_index_forward: bool | None = None*, *select: str | None = None*) → Dict

Performs the Query operation and returns the result

scan(*filter_condition: Any | None = None*, *attributes_to_get: Any | None = None*, *limit: int | None = None*, *return_consumed_capacity: str | None = None*, *segment: int | None = None*, *total_segments: int | None = None*, *exclusive_start_key: str | None = None*, *consistent_read: bool | None = None*, *index_name: str | None = None*) → Dict

Performs the scan operation

update_item(*hash_key: str*, *range_key: str | None = None*, *actions: Sequence[Action] | None = None*, *condition: Condition | None = None*, *return_consumed_capacity: str | None = None*, *return_item_collection_metrics: str | None = None*, *return_values: str | None = None*) → Dict

Performs the UpdateItem operation

update_table(*read_capacity_units: int | None = None*, *write_capacity_units: int | None = None*, *global_secondary_index_updates: Any | None = None*) → Dict

Performs the UpdateTable operation and returns the result

update_time_to_live(*ttl_attr_name: str*) → Dict

Performs the UpdateTimeToLive operation and returns the result

3.1.3 Exceptions

exception `pynamodb.exceptions.`**PynamoDBException**(*msg: str | None = None*, *cause: Exception | None = None*)

Base class for all PynamoDB exceptions.

property cause_response_code: str | None

The DynamoDB response code such as:

- `ConditionalCheckFailedException`
- `ProvisionedThroughputExceededException`
- `TransactionCanceledException`

Inspect this value to determine the cause of the error and handle it.

property cause_response_message: str | None

The human-readable description of the error returned by DynamoDB.

exception `pynamodb.exceptions.`**PynamoDBConnectionError**(*msg: str | None = None*, *cause: Exception | None = None*)

A base class for connection errors

exception `pynamodb.exceptions.`**DeleteError**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when an error occurs deleting an item

exception `pynamodb.exceptions.`**QueryError**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when queries fail

exception `pynamodb.exceptions.`**ScanError**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when a scan operation fails

exception `pynamodb.exceptions.`**`PutError`**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when an item fails to be created

exception `pynamodb.exceptions.`**`UpdateError`**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when an item fails to be updated

exception `pynamodb.exceptions.`**`GetError`**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when an item fails to be retrieved

exception `pynamodb.exceptions.`**`TableError`**(*msg: str | None = None*, *cause: Exception | None = None*)

An error involving a dynamodb table operation

exception `pynamodb.exceptions.`**`TableDoesNotExist`**(*table_name: str*)

Raised when an operation is attempted on a table that doesn't exist

exception `pynamodb.exceptions.`**`DoesNotExist`**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when an item queried does not exist

exception `pynamodb.exceptions.`**`TransactWriteError`**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when a *`TransactWrite`* operation fails.

property cancellation_reasons: List[*CancellationReason* | None]

When *`cause_response_code`* is `TransactionCanceledException`, this property lists cancellation reasons in the same order as the transaction items (one-to-one). Items which were not part of the reason for cancellation would have `None` as the value.

For a list of possible cancellation reasons and their semantics, see TransactWriteItems in the AWS documentation.

exception `pynamodb.exceptions.`**`TransactGetError`**(*msg: str | None = None*, *cause: Exception | None = None*)

Raised when a *`TransactGet`* operation fails.

property cancellation_reasons: List[*CancellationReason* | None]

When *`cause_response_code`* is `TransactionCanceledException`, this property lists cancellation reasons in the same order as the transaction items (one-to-one). Items which were not part of the reason for cancellation would have `None` as the value.

For a list of possible cancellation reasons and their semantics, see TransactGetItems in the AWS documentation.

exception `pynamodb.exceptions.`**`InvalidStateError`**(*msg: str | None = None*, *cause: Exception | None = None*)

Raises when the internal state of an operation context is invalid.

exception `pynamodb.exceptions.`**`AttributeDeserializationError`**(*attr_name: str*, *attr_type: str*)

Raised when attribute type is invalid during deserialization.

exception `pynamodb.exceptions.`**`AttributeNullError`**(*attr_name: str*)

Raised when an attribute which is not nullable (`null=False`) is unset during serialization.

class `pynamodb.exceptions.`**`CancellationReason`**(*code: str*, *message: str | None = None*, *raw_item: Dict[str, Dict[str, Any]] | None = None*)

A reason for a transaction cancellation.

For a list of possible cancellation reasons and their semantics, see TransactGetItems and TransactWriteItems in the AWS documentation.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pynamodb.attributes, 83
pynamodb.connection, 91
pynamodb.indexes, 89
pynamodb.models, 79
pynamodb.pagination, 90
pynamodb.transactions, 90

INDEX

A

acquire() (*pynamodb.pagination.RateLimiter method*), 91
add_meta_table() (*pynamodb.connection.Connection method*), 91
AllProjection (*class in pynamodb.indexes*), 89
Attribute (*class in pynamodb.attributes*), 83
AttributeContainer (*class in pynamodb.attributes*), 84
AttributeDeserializationError, 96
AttributeNullError, 96

B

batch_get() (*pynamodb.models.Model class method*), 80
batch_get_item() (*pynamodb.connection.Connection method*), 91
batch_get_item() (*pynamodb.connection.TableConnection method*), 94
batch_write() (*pynamodb.models.Model class method*), 80
batch_write_item() (*pynamodb.connection.Connection method*), 92
batch_write_item() (*pynamodb.connection.TableConnection method*), 94
BatchWrite (*class in pynamodb.models*), 79
BinaryAttribute (*class in pynamodb.attributes*), 85
BinarySetAttribute (*class in pynamodb.attributes*), 85
BooleanAttribute (*class in pynamodb.attributes*), 85

C

cancellation_reasons (*pynamodb.exceptions.TransactGetError property*), 96
cancellation_reasons (*pynamodb.exceptions.TransactWriteError property*), 96
CancellationReason (*class in pynamodb.exceptions*), 96
cause_response_code (*pynamodb.exceptions.PynamoDBException property*), 95
cause_response_message (*pynamodb.exceptions.PynamoDBException property*), 95
client (*pynamodb.connection.Connection property*), 92
commit() (*pynamodb.models.BatchWrite method*), 79
Connection (*class in pynamodb.connection*), 91
consume() (*pynamodb.pagination.RateLimiter method*), 91
count() (*pynamodb.indexes.Index method*), 90
count() (*pynamodb.models.Model class method*), 80
create_table() (*pynamodb.connection.Connection method*), 92
create_table() (*pynamodb.connection.TableConnection method*), 94
create_table() (*pynamodb.models.Model class method*), 80

D

delete() (*pynamodb.models.BatchWrite method*), 79
delete() (*pynamodb.models.Model method*), 80
delete_item() (*pynamodb.connection.Connection method*), 92
delete_item() (*pynamodb.connection.TableConnection method*), 94
delete_table() (*pynamodb.connection.Connection method*), 92
delete_table() (*pynamodb.connection.TableConnection method*), 94
delete_table() (*pynamodb.models.Model class method*), 81
DeleteError, 95
describe_table() (*pynamodb.connection.Connection method*), 92
describe_table() (*pynamodb.connection.TableConnection method*), 94
describe_table() (*pynamodb.models.Model class*

method), 81
deserialize() (*pynamodb.attributes.Attribute method*), 84
deserialize() (*pynamodb.attributes.BinaryAttribute method*), 85
deserialize() (*pynamodb.attributes.BinarySetAttribute method*), 85
deserialize() (*pynamodb.attributes.BooleanAttribute method*), 85
deserialize() (*pynamodb.attributes.DiscriminatorAttribute method*), 86
deserialize() (*pynamodb.attributes.DynamicMapAttribute method*), 86
deserialize() (*pynamodb.attributes.JSONAttribute method*), 86
deserialize() (*pynamodb.attributes.ListAttribute method*), 86
deserialize() (*pynamodb.attributes.MapAttribute method*), 87
deserialize() (*pynamodb.attributes.NullAttribute method*), 88
deserialize() (*pynamodb.attributes.NumberAttribute method*), 88
deserialize() (*pynamodb.attributes.NumberSetAttribute method*), 88
deserialize() (*pynamodb.attributes.TTLAttribute method*), 88
deserialize() (*pynamodb.attributes.UnicodeSetAttribute method*), 89
deserialize() (*pynamodb.attributes.UTCDateTimeAttribute method*), 89
deserialize() (*pynamodb.attributes.VersionAttribute method*), 89
deserialize() (*pynamodb.models.Model method*), 81
DiscriminatorAttribute (*class in pynamodb.attributes*), 86
dispatch() (*pynamodb.connection.Connection method*), 92
DoesNotExist, 96
DynamicMapAttribute (*class in pynamodb.attributes*), 86

E

exists() (*pynamodb.models.Model class method*), 81

F

from_dynamodb_dict() (*pynamodb.attributes.AttributeContainer method*), 84
from_raw_data() (*pynamodb.models.Model class method*), 81
from_simple_dict() (*pynamodb.attributes.AttributeContainer method*), 84

G

get() (*pynamodb.models.Model class method*), 81
get() (*pynamodb.transactions.TransactGet method*), 90
get_attribute_type() (*pynamodb.connection.Connection method*), 92
get_attributes() (*pynamodb.attributes.AttributeContainer class method*), 84
get_consumed_capacity_map() (*pynamodb.connection.Connection method*), 92
get_exclusive_start_key_map() (*pynamodb.connection.Connection method*), 92
get_identifier_map() (*pynamodb.connection.Connection method*), 92
get_item() (*pynamodb.connection.Connection method*), 92
get_item() (*pynamodb.connection.TableConnection method*), 94
get_item_attribute_map() (*pynamodb.connection.Connection method*), 92
get_item_collection_map() (*pynamodb.connection.Connection method*), 92
get_meta_table() (*pynamodb.connection.Connection method*), 93
get_meta_table() (*pynamodb.connection.TableConnection method*), 94
get_return_values_map() (*pynamodb.connection.Connection method*), 93
get_return_values_on_condition_failure_map() (*pynamodb.connection.Connection method*), 93
GetError, 96
GlobalSecondaryIndex (*class in pynamodb.indexes*), 89

I

IncludeProjection (*class in pynamodb.indexes*), 89
Index (*class in pynamodb.indexes*), 89
InvalidStateError, 96

J

JSONAttribute (*class in pynamodb.attributes*), 86

K

KeysOnlyProjection (*class in pynamodb.indexes*), 90

L

list_tables() (*pynamodb.connection.Connection method*), 93
ListAttribute (*class in pynamodb.attributes*), 86
LocalSecondaryIndex (*class in pynamodb.indexes*), 90

M

MapAttribute (*class in pynamodb.attributes*), 87
MetaModel (*class in pynamodb.models*), 79
MetaProtocol (*class in pynamodb.models*), 79
Model (*class in pynamodb.models*), 79
Model.DoesNotExist, 80
module
    pynamodb.attributes, 83
    pynamodb.connection, 91
    pynamodb.indexes, 89
    pynamodb.models, 79
    pynamodb.pagination, 90
    pynamodb.transactions, 90

N

NullAttribute (*class in pynamodb.attributes*), 88
NumberAttribute (*class in pynamodb.attributes*), 88
NumberSetAttribute (*class in pynamodb.attributes*), 88

P

PageIterator (*class in pynamodb.pagination*), 90
parse_attribute() (*pynamodb.connection.Connection method*), 93
Projection (*class in pynamodb.indexes*), 90
put_item() (*pynamodb.connection.Connection method*), 93
put_item() (*pynamodb.connection.TableConnection method*), 94
PutError, 95
pynamodb.attributes
    module, 83
pynamodb.connection
    module, 91
pynamodb.indexes
    module, 89
pynamodb.models
    module, 79
pynamodb.pagination
    module, 90
pynamodb.transactions
    module, 90
PynamoDBConnectionError, 95
PynamoDBException, 95

Q

query() (*pynamodb.connection.Connection method*), 93
query() (*pynamodb.connection.TableConnection method*), 94
query() (*pynamodb.indexes.Index method*), 90
query() (*pynamodb.models.Model class method*), 81
QueryError, 95

R

rate_limit (*pynamodb.pagination.RateLimiter property*), 91
RateLimiter (*class in pynamodb.pagination*), 90
refresh() (*pynamodb.models.Model method*), 82
ResultIterator (*class in pynamodb.pagination*), 91

S

save() (*pynamodb.models.BatchWrite method*), 79
save() (*pynamodb.models.Model method*), 82
scan() (*pynamodb.connection.Connection method*), 93
scan() (*pynamodb.connection.TableConnection method*), 95
scan() (*pynamodb.indexes.Index method*), 90
scan() (*pynamodb.models.Model class method*), 82
ScanError, 95
serialize() (*pynamodb.attributes.Attribute method*), 84
serialize() (*pynamodb.attributes.BinaryAttribute method*), 85
serialize() (*pynamodb.attributes.BinarySetAttribute method*), 85
serialize() (*pynamodb.attributes.BooleanAttribute method*), 86
serialize() (*pynamodb.attributes.DiscriminatorAttribute method*), 86
serialize() (*pynamodb.attributes.DynamicMapAttribute method*), 86
serialize() (*pynamodb.attributes.JSONAttribute method*), 86
serialize() (*pynamodb.attributes.ListAttribute method*), 86
serialize() (*pynamodb.attributes.MapAttribute method*), 87
serialize() (*pynamodb.attributes.NullAttribute method*), 88
serialize() (*pynamodb.attributes.NumberAttribute method*), 88
serialize() (*pynamodb.attributes.NumberSetAttribute method*), 88
serialize() (*pynamodb.attributes.TTLAttribute method*), 88
serialize() (*pynamodb.attributes.UnicodeSetAttribute method*), 89
serialize() (*pynamodb.attributes.UTCDateTimeAttribute method*), 89
serialize() (*pynamodb.attributes.VersionAttribute method*), 89

serialize() (*pynamodb.models.Model method*), 82
session (*pynamodb.connection.Connection property*), 93

T

TableConnection (*class in pynamodb.connection*), 94
TableDoesNotExist, 96
TableError, 96
to_dynamodb_dict() (*pynamodb.attributes.AttributeContainer method*), 84
to_simple_dict() (*pynamodb.attributes.AttributeContainer method*), 84
transact_get_items() (*pynamodb.connection.Connection method*), 93
transact_write_items() (*pynamodb.connection.Connection method*), 93
TransactGet (*class in pynamodb.transactions*), 90
TransactGetError, 96
Transaction (*class in pynamodb.transactions*), 90
TransactWrite (*class in pynamodb.transactions*), 90
TransactWriteError, 96
TTLAttribute (*class in pynamodb.attributes*), 88

U

UnicodeAttribute (*class in pynamodb.attributes*), 89
UnicodeSetAttribute (*class in pynamodb.attributes*), 89
update() (*pynamodb.models.Model method*), 82
update_item() (*pynamodb.connection.Connection method*), 93
update_item() (*pynamodb.connection.TableConnection method*), 95
update_table() (*pynamodb.connection.Connection method*), 93
update_table() (*pynamodb.connection.TableConnection method*), 95
update_time_to_live() (*pynamodb.connection.Connection method*), 94
update_time_to_live() (*pynamodb.connection.TableConnection method*), 95
update_ttl() (*pynamodb.models.Model class method*), 83
UpdateError, 96
UTCDateTimeAttribute (*class in pynamodb.attributes*), 89

V

VersionAttribute (*class in pynamodb.attributes*), 89