cqlengine Documentation

Release 0.21.0

Blake Eggleston, Jon Haddad

April 24, 2016

Contents

Users of versions < 0.16, the default keyspace 'cqlengine' has been removed. Please read this before upgrading: *Breaking Changes*

cqlengine is a Cassandra CQL 3 Object Mapper for Python

Getting Started

CHAPTER 1

Download

Github

PyPi

CHAPTER 2

Contents:

2.1 Models

Users of versions < 0.4, please read this post before upgrading: Breaking Changes A model is a python class representing a CQL table.

2.1.1 Examples

This example defines a Person table, with the columns `first_name` and `last_name`

```
from cqlengine import columns
from cqlengine.models import Model

class Person(Model):
    id = columns.UUID(primary_key=True)
    first_name  = columns.Text()
    last_name = columns.Text()
```

The Person model would create this CQL table:

```
CREATE TABLE cqlengine.person (
    id uuid,
    first_name text,
    last_name text,
    PRIMARY KEY (id)
)
```

Here's an example of a comment table created with clustering keys, in descending order:

```
from cqlengine import columns
from cqlengine.models import Model

class Comment(Model):
    photo_id = columns.UUID(primary_key=True)
    comment_id = columns.TimeUUID(primary_key=True, clustering_order="DESC")
    comment = columns.Text()
```

The Comment model's `create table` would look like the following:

```
CREATE TABLE comment (
  photo_id uuid,
  comment_id timeuuid,
```

```
  comment text,
  PRIMARY KEY (photo_id, comment_id)
) WITH CLUSTERING ORDER BY (comment_id DESC)
```

To sync the models to the database, you may do the following:

```
from cqlengine.management import sync_table
sync_table(Person)
sync_table(Comment)
```

2.1.2 Columns

Columns in your models map to columns in your CQL table. You define CQL columns by defining column attributes on your model classes. For a model to be valid it needs at least one primary key column and one non-primary key column.

Just as in CQL, the order you define your columns in is important, and is the same order they are defined in on a model's corresponding table.

2.1.3 Column Types

Each column on your model definitions needs to an instance of a Column class. The column types that are included with cqlengine as of this writing are:

- *Bytes*
- *Ascii*
- *Text*
- *Integer*
- *BigInt*
- *DateTime*
- *UUID*
- *TimeUUID*
- *Boolean*
- *Float*
- *Decimal*
- *Set*
- *List*
- *Map*

Column Options

Each column can be defined with optional arguments to modify the way they behave. While some column types may define additional column options, these are the options that are available on all columns:

primary_key If True, this column is created as a primary key field. A model can have multiple primary keys. Defaults to False.

In CQL, there are 2 types of primary keys: partition keys and clustering keys. As with CQL, the first primary key is the partition key, and all others are clustering keys, unless partition keys are specified manually using `partition_key`

partition_key If True, this column is created as partition primary key. There may be many partition keys defined, forming a *composite partition key*

clustering_order `ASC` or `DESC`, determines the clustering order of a clustering key.

index If True, an index will be created for this column. Defaults to False.

db_field Explicitly sets the name of the column in the database table. If this is left blank, the column name will be the same as the name of the column attribute. Defaults to None.

default The default value for this column. If a model instance is saved without a value for this column having been defined, the default value will be used. This can be either a value or a callable object (ie: datetime.now is a valid default argument). Callable defaults will be called each time a default is assigned to a None value

required If True, this model cannot be saved without a value defined for this column. Defaults to False. Primary key fields always require values.

static Defined a column as static. Static columns are shared by all rows in a partition.

2.1.4 Model Methods

Below are the methods that can be called on model instances.

class `cqlengine.models.`**`Model`**`(`***values*`)`
Creates an instance of the model. Pass in keyword arguments for columns you've defined on the model.

Example

```
#using the person model from earlier:
class Person(Model):
    id = columns.UUID(primary_key=True)
    first_name  = columns.Text()
    last_name = columns.Text()

person = Person(first_name='Blake', last_name='Eggleston')
person.first_name  #returns 'Blake'
person.last_name  #returns 'Eggleston'
```

`save``()`
Saves an object to the database

Example

```
#create a person instance
person = Person(first_name='Kimberly', last_name='Eggleston')
#saves it to Cassandra
person.save()
```

`delete``()`
Deletes the object from the database.

`batch``(`*batch_object*`)`
Sets the batch object to run instance updates and inserts queries with.

timestamp(*timedelta_or_datetime*)
Sets the timestamp for the query

ttl(*ttl_in_sec*)
Sets the ttl values to run instance updates and inserts queries with.

if_not_exists()
Check the existence of an object before insertion. The existence of an object is determined by its primary key(s). And please note using this flag would incur performance cost.

if the insertion didn't applied, a LWTException exception would be raised.

Example

This method is supported on Cassandra 2.0 or later.

iff(***values*)
Checks to ensure that the values specified are correct on the Cassandra cluster. Simply specify the column(s) and the expected value(s). As with if_not_exists, this incurs a performance cost.

If the insertion isn't applied, a LWTException is raised

update(***values*)
Performs an update on the model instance. You can pass in values to set on the model for updating, or you can call without values to execute an update against any modified fields. If no fields on the model have been modified since loading, no query will be performed. Model validation is performed normally.

It is possible to do a blind update, that is, to update a field without having first selected the object out of the database. See *Blind Updates*

get_changed_columns()
Returns a list of column names that have changed since the model was instantiated or saved

2.1.5 Model Attributes

`Model.__abstract__`
Optional. Indicates that this model is only intended to be used as a base class for other models. You can't create tables for abstract models, but checks around schema validity are skipped during class construction.

`Model.__table_name__`
Optional. Sets the name of the CQL table for this model. If left blank, the table name will be the name of the model, with it's module name as it's prefix. Manually defined table names are not inherited.

`Model.__keyspace__`
Sets the name of the keyspace used by this model.

Prior to cqlengine 0.16, this setting defaulted to 'cqlengine'. As of 0.16, this field needs to be set on all non-abstract models, or their base classes.

`Model.__default_ttl__`
Sets the default ttl used by this model. This can be overridden by using the `ttl(ttl_in_sec)` method.

2.1.6 Table Polymorphism

As of cqlengine 0.8, it is possible to save and load different model classes using a single CQL table. This is useful in situations where you have different object types that you want to store in a single cassandra

row.

For instance, suppose you want a table that stores rows of pets owned by an owner:

```
class Pet(Model):
    __table_name__ = 'pet'
    owner_id = UUID(primary_key=True)
    pet_id = UUID(primary_key=True)
    pet_type = Text(polymorphic_key=True)
    name = Text()

    def eat(self, food):
        pass

    def sleep(self, time):
        pass

class Cat(Pet):
    __polymorphic_key__ = 'cat'
    cuteness = Float()

    def tear_up_couch(self):
        pass

class Dog(Pet):
    __polymorphic_key__ = 'dog'
    fierceness = Float()

    def bark_all_night(self):
        pass
```

After calling `sync_table` on each of these tables, the columns defined in each model will be added to the `pet` table. Additionally, saving `Cat` and `Dog` models will save the meta data needed to identify each row as either a cat or dog.

To setup a polymorphic model structure, follow these steps

1. Create a base model with a column set as the polymorphic_key (set `polymorphic_key=True` in the column definition)
2. Create subclass models, and define a unique `__polymorphic_key__` value on each
3. Run `sync_table` on each of the sub tables

About the polymorphic key

The polymorphic key is what cqlengine uses under the covers to map logical cql rows to the appropriate model type. The base model maintains a map of polymorphic keys to subclasses. When a polymorphic model is saved, this value is automatically saved into the polymorphic key column. You can set the polymorphic key column to any column type that you like, with the exception of container and counter columns, although `Integer` columns make the most sense. Additionally, if you set `index=True` on your polymorphic key column, you can execute queries against polymorphic subclasses, and a `WHERE` clause will be automatically added to your query, returning only rows of that type. Note that you must define a unique `__polymorphic_key__` value to each subclass, and that you can only assign a single polymorphic key column per model

2.1.7 Extending Model Validation

Each time you save a model instance in cqlengine, the data in the model is validated against the schema you've defined for your model. Most of the validation is fairly straightforward, it basically checks that you're not trying to do something like save text into an integer column, and it enforces the `required` flag set on column definitions. It also performs any transformations needed to save the data properly.

However, there are often additional constraints or transformations you want to impose on your data, beyond simply making sure that Cassandra won't complain when you try to insert it. To define additional validation on a model, extend the model's validation method:

```
class Member(Model):
    person_id = UUID(primary_key=True)
    name = Text(required=True)

    def validate(self):
        super(Member, self).validate()
        if self.name == 'jon':
            raise ValidationError('no jon\'s allowed')
```

Note: while not required, the convention is to raise a `ValidationError` (`from cqlengine import ValidationError`) if validation fails

2.1.8 Table Properties

Each table can have its own set of configuration options. These can be specified on a model with the following attributes:

`Model.__bloom_filter_fp_chance`

`Model.__caching__`

`Model.__comment__`

`Model.__dclocal_read_repair_chance__`

`Model.__default_time_to_live__`

`Model.__gc_grace_seconds__`

`Model.__index_interval__`

`Model.__memtable_flush_period_in_ms__`

`Model.__populate_io_cache_on_flush__`

`Model.__read_repair_chance__`

`Model.__replicate_on_write__`

Example:

```
from cqlengine import CACHING_ROWS_ONLY, columns
from cqlengine.models import Model

class User(Model):
    __caching__ = CACHING_ROWS_ONLY  # cache only rows instead of keys only by default
    __gc_grace_seconds__ = 86400  # 1 day instead of the default 10 days

    user_id = columns.UUID(primary_key=True)
    name = columns.Text()
```

Will produce the following CQL statement:

```
CREATE TABLE cqlengine.user (
    user_id uuid,
    name text,
    PRIMARY KEY (user_id)
) WITH caching = 'rows_only'
   AND gc_grace_seconds = 86400;
```

See the list of supported table properties for more information.

2.1.9 Compaction Options

As of cqlengine 0.7 we've added support for specifying compaction options. cqlengine will only use your compaction options if you have a strategy set. When a table is synced, it will be altered to match the compaction options set on your table. This means that if you are changing settings manually they will be changed back on resync. Do not use the compaction settings of cqlengine if you want to manage your compaction settings manually.

cqlengine supports all compaction options as of Cassandra 1.2.8.

Available Options:

`Model.__compaction_bucket_high__`

`Model.__compaction_bucket_low__`

`Model.__compaction_max_compaction_threshold__`

`Model.__compaction_min_compaction_threshold__`

`Model.__compaction_min_sstable_size__`

`Model.__compaction_sstable_size_in_mb__`

`Model.__compaction_tombstone_compaction_interval__`

`Model.__compaction_tombstone_threshold__`

For example:

```
class User(Model):
    __compaction__ = cqlengine.LeveledCompactionStrategy
    __compaction_sstable_size_in_mb__ = 64
    __compaction_tombstone_threshold__ = .2

    user_id = columns.UUID(primary_key=True)
    name = columns.Text()
```

or for SizeTieredCompaction:

```
class TimeData(Model):
    __compaction__ = SizeTieredCompactionStrategy
    __compaction_bucket_low__ = .3
    __compaction_bucket_high__ = 2
    __compaction_min_threshold__ = 2
    __compaction_max_threshold__ = 64
    __compaction_tombstone_compaction_interval__ = 86400
```

Tables may use *LeveledCompactionStrategy* or *SizeTieredCompactionStrategy*. Both options are available in the top level cqlengine module. To reiterate, you will need to set your *__compaction__* option explicitly in order for cqlengine to handle any of your settings.

2.1.10 Manipulating model instances as dictionaries

As of cqlengine 0.12, we've added support for treating model instances like dictionaries. See below for examples.

```
class Person(Model):
    first_name  = columns.Text()
    last_name = columns.Text()

kevin = Person.create(first_name="Kevin", last_name="Deldycke")
dict(kevin)  # returns {'first_name': 'Kevin', 'last_name': 'Deldycke'}
kevin['first_name']  # returns 'Kevin'
kevin.keys()  # returns ['first_name', 'last_name']
kevin.values()  # returns ['Kevin', 'Deldycke']
kevin.items()  # returns [('first_name', 'Kevin'), ('last_name', 'Deldycke')]

kevin['first_name'] = 'KEVIN5000'  # changes the models first name
```

2.2 Making Queries

Users of versions < 0.4, please read this post before upgrading: Breaking Changes

2.2.1 Retrieving objects

Once you've populated Cassandra with data, you'll probably want to retrieve some of it. This is accomplished with QuerySet objects. This section will describe how to use QuerySet objects to retrieve the data you're looking for.

Retrieving all objects

The simplest query you can make is to return all objects from a table.

This is accomplished with the `.all()` method, which returns a QuerySet of all objects in a table

Using the Person example model, we would get all Person objects like this:

```
all_objects = Person.objects.all()
```

Retrieving objects with filters

Typically, you'll want to query only a subset of the records in your database.

That can be accomplished with the QuerySet's `.filter(\*\*)` method.

For example, given the model definition:

```
class Automobile(Model):
    manufacturer = columns.Text(primary_key=True)
    year = columns.Integer(primary_key=True)
    model = columns.Text()
    price = columns.Decimal()
```

...and assuming the Automobile table contains a record of every car model manufactured in the last 20 years or so, we can retrieve only the cars made by a single manufacturer like this:

```
q = Automobile.objects.filter(manufacturer='Tesla')
```

You can also use the more convenient syntax:

```
q = Automobile.objects(Automobile.manufacturer == 'Tesla')
```

We can then further filter our query with another call to **.filter**

```
q = q.filter(year=2012)
```

Note: all queries involving any filtering MUST define either an '=' or an 'in' relation to either a primary key column, or an indexed column.

2.2.2 Accessing objects in a QuerySet

There are several methods for getting objects out of a queryset

- **iterating over the queryset**

```
for car in Automobile.objects.all():
    #...do something to the car instance
    pass
```

- **list index**

```
q = Automobile.objects.all()
q[0] #returns the first result
q[1] #returns the second result
```

- **list slicing**

```
q = Automobile.objects.all()
q[1:] #returns all results except the first
q[1:9] #returns a slice of the results
```

Note: CQL does not support specifying a start position in it's queries. Therefore, accessing elements using array indexing / slicing will load every result up to the index value requested

- **calling `get()` on the queryset**

```
q = Automobile.objects.filter(manufacturer='Tesla')
q = q.filter(year=2012)
car = q.get()
```

this returns the object matching the queryset

- **calling `first()` on the queryset**

```
q = Automobile.objects.filter(manufacturer='Tesla')
q = q.filter(year=2012)
car = q.first()
```

this returns the first value in the queryset

2.2.3 Filtering Operators

`Equal To`

The default filtering operator.

```
q = Automobile.objects.filter(manufacturer='Tesla')
q = q.filter(year=2012)  #year == 2012
```

In addition to simple equal to queries, cqlengine also supports querying with other operators by appending a `__<op>` to the field name on the filtering call

`in (__in)`

```
q = Automobile.objects.filter(manufacturer='Tesla')
q = q.filter(year__in=[2011, 2012])
```

`> (__gt)`

```
q = Automobile.objects.filter(manufacturer='Tesla')
q = q.filter(year__gt=2010)  # year > 2010

# or the nicer syntax

q.filter(Automobile.year > 2010)
```

`>= (__gte)`

```
q = Automobile.objects.filter(manufacturer='Tesla')
q = q.filter(year__gte=2010)  # year >= 2010

# or the nicer syntax

q.filter(Automobile.year >= 2010)
```

`< (__lt)`

```
q = Automobile.objects.filter(manufacturer='Tesla')
q = q.filter(year__lt=2012)  # year < 2012

# or...

q.filter(Automobile.year < 2012)
```

`<= (__lte)`

```
q = Automobile.objects.filter(manufacturer='Tesla')
q = q.filter(year__lte=2012)  # year <= 2012

q.filter(Automobile.year <= 2012)
```

2.2.4 TimeUUID Functions

In addition to querying using regular values, there are two functions you can pass in when querying TimeUUID columns to help make filtering by them easier. Note that these functions don't actually return a value, but instruct the cql interpreter to use the functions in it's query.

class `cqlengine.query.`**`MinTimeUUID`**(*datetime*)
returns the minimum time uuid value possible for the given datetime

class cqlengine.query.**MaxTimeUUID**(*datetime*)
returns the maximum time uuid value possible for the given datetime

Example

```
class DataStream(Model):
    time     = cqlengine.TimeUUID(primary_key=True)
    data     = cqlengine.Bytes()

min_time = datetime(1982, 1, 1)
max_time = datetime(1982, 3, 9)

DataStream.filter(time__gt=cqlengine.MinTimeUUID(min_time), time__lt=cqlengine.MaxTimeUUID(max_t
```

2.2.5 Token Function

Token functon may be used only on special, virtual column pk__token, representing token of partition key (it also works for composite partition keys). Cassandra orders returned items by value of partition key token, so using cqlengine.Token we can easy paginate through all table rows.

See http://cassandra.apache.org/doc/cql3/CQL.html#tokenFun

Example

```
class Items(Model):
    id       = cqlengine.Text(primary_key=True)
    data     = cqlengine.Bytes()

query = Items.objects.all().limit(10)

first_page = list(query);
last = first_page[-1]
next_page = list(query.filter(pk__token__gt=cqlengine.Token(last.pk)))
```

2.2.6 QuerySets are immutable

When calling any method that changes a queryset, the method does not actually change the queryset object it's called on, but returns a new queryset object with the attributes of the original queryset, plus the attributes added in the method call.

Example

```
#this produces 3 different querysets
#q does not change after it's initial definition
q = Automobiles.objects.filter(year=2012)
tesla2012 = q.filter(manufacturer='Tesla')
honda2012 = q.filter(manufacturer='Honda')
```

2.2.7 Ordering QuerySets

Since Cassandra is essentially a distributed hash table on steroids, the order you get records back in will not be particularly predictable.

However, you can set a column to order on with the `.order_by(column_name)` method.

Example

```
#sort ascending
q = Automobiles.objects.all().order_by('year')
#sort descending
q = Automobiles.objects.all().order_by('-year')
```

Note: Cassandra only supports ordering on a clustering key. In other words, to support ordering results, your model must have more than one primary key, and you must order on a primary key, excluding the first one.

For instance, given our Automobile model, year is the only column we can order on.

2.2.8 Values Lists

There is a special QuerySet's method `.values_list()` - when called, QuerySet returns lists of values instead of model instances. It may significantly speedup things with lower memory footprint for large responses. Each tuple contains the value from the respective field passed into the `values_list()` call — so the first item is the first field, etc. For example:

```
items = list(range(20))
random.shuffle(items)
for i in items:
    TestModel.create(id=1, clustering_key=i)

values = list(TestModel.objects.values_list('clustering_key', flat=True))
# [19L, 18L, 17L, 16L, 15L, 14L, 13L, 12L, 11L, 10L, 9L, 8L, 7L, 6L, 5L, 4L, 3L, 2L, 1L, 0L]
```

2.2.9 Batch Queries

cqlengine now supports batch queries using the BatchQuery class. Batch queries can be started and stopped manually, or within a context manager. To add queries to the batch object, you just need to precede the create/save/delete call with a call to batch, and pass in the batch object.

Batch Query General Use Pattern

You can only create, update, and delete rows with a batch query, attempting to read rows out of the database with a batch query will fail.

```
from cqlengine import BatchQuery

#using a context manager
with BatchQuery() as b:
    now = datetime.now()
    em1 = ExampleModel.batch(b).create(example_type=0, description="1", created_at=now)
    em2 = ExampleModel.batch(b).create(example_type=0, description="2", created_at=now)
    em3 = ExampleModel.batch(b).create(example_type=0, description="3", created_at=now)

# -- or --

#manually
b = BatchQuery()
now = datetime.now()
em1 = ExampleModel.batch(b).create(example_type=0, description="1", created_at=now)
em2 = ExampleModel.batch(b).create(example_type=0, description="2", created_at=now)
em3 = ExampleModel.batch(b).create(example_type=0, description="3", created_at=now)
```

```
    b.execute()

    # updating in a batch

    b = BatchQuery()
    em1.description = "new description"
    em1.batch(b).save()
    em2.description = "another new description"
    em2.batch(b).save()
    b.execute()

    # deleting in a batch
    b = BatchQuery()
    ExampleModel.objects(id=some_id).batch(b).delete()
    ExampleModel.objects(id=some_id2).batch(b).delete()
    b.execute()
```

Typically you will not want the block to execute if an exception occurs inside the *with* block. However, in the case that this is desirable, it's achievable by using the following syntax:

```
    with BatchQuery(execute_on_exception=True) as b:
        LogEntry.batch(b).create(k=1, v=1)
        mystery_function() # exception thrown in here
        LogEntry.batch(b).create(k=1, v=2) # this code is never reached due to the exception, but ar
```

If an exception is thrown somewhere in the block, any statements that have been added to the batch will still be executed. This is useful for some logging situations.

Batch Query Execution Callbacks

In order to allow secondary tasks to be chained to the end of batch, BatchQuery instances allow callbacks to be registered with the batch, to be executed immediately after the batch executes.

Multiple callbacks can be attached to same BatchQuery instance, they are executed in the same order that they are added to the batch.

The callbacks attached to a given batch instance are executed only if the batch executes. If the batch is used as a context manager and an exception is raised, the queued up callbacks will not be run.

```
    def my_callback(*args, **kwargs):
        pass

    batch = BatchQuery()

    batch.add_callback(my_callback)
    batch.add_callback(my_callback, 'positional arg', named_arg='named arg value')

    # if you need reference to the batch within the callback,
    # just trap it in the arguments to be passed to the callback:
    batch.add_callback(my_callback, cqlengine_batch=batch)

    # once the batch executes...
    batch.execute()

    # the effect of the above scheduled callbacks will be similar to
    my_callback()
    my_callback('positional arg', named_arg='named arg value')
    my_callback(cqlengine_batch=batch)
```

Failure in any of the callbacks does not affect the batch's execution, as the callbacks are started after the execution of the batch is complete.

Logged vs Unlogged Batches

By default, queries in cqlengine are LOGGED, which carries additional overhead from UNLOGGED. To explicitly state which batch type to use, simply:

```
from cqlengine.query import BatchType
with BatchQuery(batch_type=BatchType.Unlogged) as b:
    LogEntry.batch(b).create(k=1, v=1)
    LogEntry.batch(b).create(k=1, v=2)
```

2.2.10 QuerySet method reference

class `cqlengine.query.`**`QuerySet`**

`all``()`
Returns a queryset matching all rows

```
for user in User.objects().all():
    print(user)
```

`batch`(*batch_object*)
Sets the batch object to run the query on. Note that running a select query with a batch object will raise an exception

`consistency`(*consistency_setting*)
Sets the consistency level for the operation. Options may be imported from the top level `cqlengine` package.

```
for user in User.objects(id=3).consistency(ONE):
    print(user)
```

`count``()`
Returns the number of matching rows in your QuerySet

```
print(User.objects().count())
```

`filter`(***values*)

Parameters **`values`** – See *Retrieving objects with filters*

Returns a QuerySet filtered on the keyword arguments

`get`(***values*)

Parameters **`values`** – See *Retrieving objects with filters*

Returns a single object matching the QuerySet. If no objects are matched, a `DoesNotExist` exception is raised. If more than one object is found, a `MultipleObjectsReturned` exception is raised.

```
user = User.get(id=1)
```

`limit`(*num*)
Limits the number of results returned by Cassandra.

Note that CQL's default limit is 10,000, so all queries without a limit set explicitly will have an implicit limit of 10,000

```
for user in User.objects().limit(100):
    print(user)
```

order_by(*field_name*)

Parameters **field_name** (*string*) – the name of the field to order on. *Note: the field_name must be a clustering key*

Sets the field to order on.

```
from uuid import uuid1,uuid4

class Comment(Model):
    photo_id = UUID(primary_key=True)
    comment_id = TimeUUID(primary_key=True, default=uuid1) # auto becomes clustering key
    comment = Text()

sync_table(Comment)

u = uuid4()
for x in range(5):
    Comment.create(photo_id=u, comment="test %d" % x)

print("Normal")
for comment in Comment.objects(photo_id=u):
    print comment.comment_id

print("Reversed")
for comment in Comment.objects(photo_id=u).order_by("-comment_id"):
    print comment.comment_id
```

allow_filtering()

Enables the (usually) unwise practive of querying on a clustering key without also defining a partition key

timestamp(*timestamp_or_long_or_datetime*)

Allows for custom timestamps to be saved with the record.

ttl(*ttl_in_seconds*)

Parameters **ttl_in_seconds** (*int*) – time in seconds in which the saved values should expire

Sets the ttl to run the query query with. Note that running a select query with a ttl value will raise an exception

update(**values*)

Performs an update on the row selected by the queryset. Include values to update in the update like so:

```
Model.objects(key=n).update(value='x')
```

Passing in updates for columns which are not part of the model will raise a ValidationError. Per column validation will be performed, but instance level validation will not (*Model.validate* is not called). This is sometimes referred to as a blind update.

For example:

```
class User(Model):
    id = Integer(primary_key=True)
    name = Text()
```

```
setup(["localhost"], "test")
sync_table(User)

u = User.create(id=1, name="jon")

User.objects(id=1).update(name="Steve")

# sets name to null
User.objects(id=1).update(name=None)
```

The queryset update method also supports blindly adding and removing elements from container columns, without loading a model instance from Cassandra.

Using the syntax *.update(column_name={x, y, z})* will overwrite the contents of the container, like updating a non container column. However, adding __<*operation*> to the end of the keyword arg, makes the update call add or remove items from the collection, without overwriting then entire column.

Given the model below, here are the operations that can be performed on the different container columns:

```
class Row(Model):
    row_id      = columns.Integer(primary_key=True)
    set_column  = columns.Set(Integer)
    list_column = columns.Set(Integer)
    map_column  = columns.Set(Integer, Integer)
```

Set

- *add*: adds the elements of the given set to the column
- *remove*: removes the elements of the given set to the column

```
# add elements to a set
Row.objects(row_id=5).update(set_column__add={6})

# remove elements to a set
Row.objects(row_id=5).update(set_column__remove={4})
```

List

- *append*: appends the elements of the given list to the end of the column
- *prepend*: prepends the elements of the given list to the beginning of the column

```
# append items to a list
Row.objects(row_id=5).update(list_column__append=[6, 7])

# prepend items to a list
Row.objects(row_id=5).update(list_column__prepend=[1, 2])
```

Map

- *update*: adds the given keys/values to the columns, creating new entries if they didn't exist, and overwriting old ones if they did

```
# add items to a map
Row.objects(row_id=5).update(map_column__update={1: 2, 3: 4})
```

2.2.11 Per Query Timeouts

By default all queries are executed with the timeout defined in *~cqlengine.connection.setup()* The examples below show how to specify a per-query timeout. A timeout is specified in seconds and can be an int, float or None. None means no timeout.

```
class Row(Model):
    id = columns.Integer(primary_key=True)
    name = columns.Text()
```

Fetch all objects with a timeout of 5 seconds

```
Row.objects().timeout(5).all()
```

Create a single row with a 50ms timeout

```
Row(id=1, name='Jon').timeout(0.05).create()
```

Delete a single row with no timeout

```
Row(id=1).timeout(None).delete()
```

Update a single row with no timeout

```
Row(id=1).timeout(None).update(name='Blake')
```

Batch query timeouts

```
with BatchQuery(timeout=10) as b:
    Row(id=1, name='Jon').create()
```

NOTE: You cannot set both timeout and batch at the same time, batch will use the timeout defined in it's constructor. Setting the timeout on the model is meaningless and will raise an AssertionError.

2.2.12 Named Tables

Named tables are a way of querying a table without creating an class. They're useful for querying system tables or exploring an unfamiliar database.

```
from cqlengine.connection import setup
setup("127.0.0.1", "cqlengine_test")

from cqlengine.named import NamedTable
user = NamedTable("cqlengine_test", "user")
user.objects()
user.objects()[0]

# {u'pk': 1, u't': datetime.datetime(2014, 6, 26, 17, 10, 31, 774000)}
```

2.3 Columns

Users of versions < 0.4, please read this post before upgrading: Breaking Changes

class cqlengine.columns.**Bytes**
Stores arbitrary bytes (no validation), expressed as hexadecimal

```
columns.Bytes()
```

class cqlengine.columns.**Ascii**
Stores a US-ASCII character string

```
columns.Ascii()
```

class cqlengine.columns.**Text**
Stores a UTF-8 encoded string

```
columns.Text()
```

options

min_length Sets the minimum length of this string. If this field is not set , and the column is not a required field, it defaults to 0, otherwise 1.

max_length Sets the maximum length of this string. Defaults to None

class cqlengine.columns.**Integer**
Stores a 32-bit signed integer value

```
columns.Integer()
```

class cqlengine.columns.**BigInt**
Stores a 64-bit signed long value

```
columns.BigInt()
```

class cqlengine.columns.**VarInt**
Stores an arbitrary-precision integer

```
columns.VarInt()
```

class cqlengine.columns.**DateTime**
Stores a datetime value.

columns.DateTime()

class cqlengine.columns.**UUID**
Stores a type 1 or type 4 UUID.

columns.UUID()

class cqlengine.columns.**TimeUUID**
Stores a UUID value as the cql type 'timeuuid'

```
columns.TimeUUID()
```

classmethod from_datetime(*dt*)
generates a TimeUUID for the given datetime

Parameters **dt** (*datetime.datetime*) – the datetime to create a time uuid from

Returns a time uuid created from the given datetime

Return type uuid1

class cqlengine.columns.**Boolean**
Stores a boolean True or False value

```
columns.Boolean()
```

class cqlengine.columns.**Float**
Stores a floating point value

```
columns.Float()
```

options

double_precision If True, stores a double precision float value, otherwise single precision. Defaults to True.

class cqlengine.columns.**Decimal**
Stores a variable precision decimal value

```
columns.Decimal()
```

class cqlengine.columns.**Counter**
Counters can be incremented and decremented

```
columns.Counter()
```

2.3.1 Collection Type Columns

CQLEngine also supports container column types. Each container column requires a column class argument to specify what type of objects it will hold. The Map column requires 2, one for the key, and the other for the value

Example

```
class Person(Model):
    id          = columns.UUID(primary_key=True, default=uuid.uuid4)
    first_name  = columns.Text()
    last_name   = columns.Text()

    friends     = columns.Set(columns.Text)
    enemies     = columns.Set(columns.Text)
    todo_list   = columns.List(columns.Text)
    birthdays   = columns.Map(columns.Text, columns.DateTime)
```

class cqlengine.columns.**Set**
Stores a set of unordered, unique values. Available only with Cassandra 1.2 and above

```
columns.Set(value_type)
```

options

value_type The type of objects the set will contain

strict If True, adding this column will raise an exception during save if the value is not a python *set* instance. If False, it will attempt to coerce the value to a set. Defaults to True.

class cqlengine.columns.**List**
Stores a list of ordered values. Available only with Cassandra 1.2 and above

```
columns.List(value_type)
```

options

value_type The type of objects the set will contain

class cqlengine.columns.**Map**
Stores a map (dictionary) collection, available only with Cassandra 1.2 and above

```
columns.Map(key_type, value_type)
```

options

key_type The type of the map keys

value_type The type of the map values

Column Options

Each column can be defined with optional arguments to modify the way they behave. While some column types may define additional column options, these are the options that are available on all columns:

BaseColumn.**primary_key**
If True, this column is created as a primary key field. A model can have multiple primary keys. Defaults to False.

In CQL, there are 2 types of primary keys: partition keys and clustering keys. As with CQL, the first primary key is the partition key, and all others are clustering keys, unless partition keys are specified manually using `BaseColumn.partition_key`

BaseColumn.**partition_key**
If True, this column is created as partition primary key. There may be many partition keys defined, forming a *composite partition key*

BaseColumn.**index**
If True, an index will be created for this column. Defaults to False.

Note: Indexes can only be created on models with one primary key

BaseColumn.**db_field**
Explicitly sets the name of the column in the database table. If this is left blank, the column name will be the same as the name of the column attribute. Defaults to None.

BaseColumn.**default**
The default value for this column. If a model instance is saved without a value for this column having been defined, the default value will be used. This can be either a value or a callable object (ie: datetime.now is a valid default argument).

BaseColumn.**required**
If True, this model cannot be saved without a value defined for this column. Defaults to False. Primary key fields cannot have their required fields set to False.

BaseColumn.**clustering_order**
Defines CLUSTERING ORDER for this column (valid choices are "asc" (default) or "desc"). It may be specified only for clustering primary keys - more: http://www.datastax.com/docs/1.2/cql_cli/cql/CREATE_TABLE#using-clustering-order

2.4 Connection

Users of versions < 0.4, please read this post before upgrading: Breaking Changes The setup function in *cqlengine.connection* records the Cassandra servers to connect to. If there is a problem with one of the servers, cqlengine will try to connect to each of the other connections before failing.

cqlengine.connection.**setup**(*hosts*)

Parameters

- **hosts** (*list*) – list of hosts, strings in the <hostname>:<port>, or just <hostname>
- **default_keyspace** (*str*) – keyspace to default to
- **consistency** (*int*) – the consistency level of the connection, defaults to 'ONE'

see http://datastax.github.io/python-driver/api/cassandra.html#cassandra.ConsistencyLevel

Records the hosts and connects to one of them

See the example at *Getting Started*

2.5 Managing Schemas

Users of versions < 0.4, please read this post before upgrading: Breaking Changes Once a connection has been made to Cassandra, you can use the functions in `cqlengine.management` to create and delete keyspaces, as well as create and delete tables for defined models

`cqlengine.management.`**`create_keyspace`**(*name*)

Parameters **name** (*string*) – the keyspace name to create

creates a keyspace with the given name

`cqlengine.management.`**`delete_keyspace`**(*name*)

Parameters **name** (*string*) – the keyspace name to delete

deletes the keyspace with the given name

`cqlengine.management.`**`sync_table`**(*model*)

Parameters **model** (`Model`) – the `Model` class to make a table with

syncs a python model to cassandra (creates & alters)

`cqlengine.management.`**`drop_table`**(*model*)

Parameters **model** (`Model`) – the `Model` class to delete a column family for

deletes the CQL table for the given model

See the example at *Getting Started*

2.6 External Resources

2.6.1 Video Tutorials

Introduction to CQLEngine: https://www.youtube.com/watch?v=zrbQcPNMbB0

TimeUUID and Table Polymorphism: https://www.youtube.com/watch?v=clXN9pnakvI

2.6.2 Blog Posts

Blake Eggleston on Table Polymorphism in the .8 release: http://blakeeggleston.com/cqlengine-08-released.html

2.7 Related Projects

2.7.1 Cassandra Native Driver

- Docs: http://datastax.github.io/python-driver/api/index.html
- Github: https://github.com/datastax/python-driver
- Pypi: https://pypi.python.org/pypi/cassandra-driver/2.1.0

2.7.2 Sphinx Contrib Module

- Github https://github.com/dokai/sphinxcontrib-cqlengine

2.7.3 Django Integration

- Github https://cqlengine.readthedocs.org

2.8 Third party integrations

2.8.1 Celery

Here's how, in substance, CQLengine can be plugged to Celery:

```
from celery import Celery
from celery.signals import worker_process_init, beat_init
from cqlengine import connection
from cqlengine.connection import (
    cluster as cql_cluster, session as cql_session)

def cassandra_init():
    """ Initialize a clean Cassandra connection. """
    if cql_cluster is not None:
        cql_cluster.shutdown()
    if cql_session is not None:
        cql_session.shutdown()
    connection.setup()

# Initialize worker context for both standard and periodic tasks.
worker_process_init.connect(cassandra_init)
beat_init.connect(cassandra_init)

app = Celery()
```

For more details, see issue #237.

2.8.2 uWSGI

This is the code required for proper connection handling of CQLengine for a uWSGI-run application:

```
from cqlengine import connection
from cqlengine.connection import (
    cluster as cql_cluster, session as cql_session)

try:
    from uwsgidecorators import postfork
except ImportError:
    # We're not in a uWSGI context, no need to hook Cassandra session
    # initialization to the postfork event.
    pass
else:
    @postfork
    def cassandra_init():
        """ Initialize a new Cassandra session in the context.

        Ensures that a new session is returned for every new request.
        """
        if cql_cluster is not None:
            cql_cluster.shutdown()
        if cql_session is not None:
            cql_session.shutdown()
        connection.setup()
```

2.9 Development

2.9.1 Travis CI

Tests are run using Travis CI using a Matrix to test different Cassandra and Python versions. It is located here: https://travis-ci.org/cqlengine/cqlengine

Python versions:

- 2.7
- 3.4

Cassandra vesions:

- 1.2 (protocol_version 1)
- 2.0 (protocol_version 2)
- 2.1 (upcoming, protocol_version 3)

2.9.2 Pull Requests

Only Pull Requests that have passed the entire matrix will be considered for merge into the main codebase.

Please see the contributing guidelines: https://github.com/cqlengine/cqlengine/blob/master/CONTRIBUTING.md

2.9.3 Testing Locally

Before testing, you'll need to set an environment variable to the version of Cassandra that's being tested. The version cooresponds to the <Major><Minor> release, so for example if you're testing against Cassandra 2.1, you'd set the following:

```
export CASSANDRA_VERSION=20
```

At the command line, execute:

```
bin/test.py
```

This is a wrapper for nose that also sets up the database connection.

2.10 Frequently Asked Questions

2.10.1 Q: Why don't updates work correctly on models instantiated as Model(field=blah, field2=blah2)?

A: The recommended way to create new rows is with the models .create method. The values passed into a model's init method are interpreted by the model as the values as they were read from a row. This allows the model to "know" which rows have changed since the row was read out of cassandra, and create suitable update statements.

2.10.2 Q: How to preserve ordering in batch query?

A: Statement Ordering is not supported by CQL3 batches. Therefore, once cassandra needs resolving conflict(Updating the same column in one batch), The algorithm below would be used.

- If timestamps are different, pick the column with the largest timestamp (the value being a regular column or a tombstone)
- If timestamps are the same, and one of the columns in a tombstone ('null') - pick the tombstone
- If timestamps are the same, and none of the columns are tombstones, pick the column with the largest value

Below is an example to show this scenario.

```
class MyMode(Model):
    id    = columns.Integer(primary_key=True)
    count = columns.Integer()
    text  = columns.Text()

with BatchQuery() as b:
   MyModel.batch(b).create(id=1, count=2, text='123')
   MyModel.batch(b).create(id=1, count=3, text='111')

assert MyModel.objects(id=1).first().count == 3
assert MyModel.objects(id=1).first().text  == '123'
```

The largest value of count is 3, and the largest value of text would be '123'.

The workaround is applying timestamp to each statement, then Cassandra would resolve to the statement with the lastest timestamp.

```
with BatchQuery() as b:
    MyModel.timestamp(datetime.now()).batch(b).create(id=1, count=2, text='123')
    MyModel.timestamp(datetime.now()).batch(b).create(id=1, count=3, text='111')

assert MyModel.objects(id=1).first().count == 3
assert MyModel.objects(id=1).first().text  == '111'
```

CHAPTER 3

Getting Started

```
#first, define a model
from cqlengine import columns
from cqlengine import Model

class ExampleModel(Model):
    example_id      = columns.UUID(primary_key=True, default=uuid.uuid4)
    example_type    = columns.Integer(index=True)
    created_at      = columns.DateTime()
    description     = columns.Text(required=False)

#next, setup the connection to your cassandra server(s)...
>>> from cqlengine import connection

# see http://datastax.github.io/python-driver/api/cassandra/cluster.html for options
# the list of hosts will be passed to create a Cluster() instance
>>> connection.setup(['127.0.0.1'], "cqlengine")

# if you're connecting to a 1.2 cluster
>>> connection.setup(['127.0.0.1'], "cqlengine", protocol_version=1)

#...and create your CQL table
>>> from cqlengine.management import sync_table
>>> sync_table(ExampleModel)

#now we can create some rows:
>>> em1 = ExampleModel.create(example_type=0, description="example1", created_at=datetime.now())
>>> em2 = ExampleModel.create(example_type=0, description="example2", created_at=datetime.now())
>>> em3 = ExampleModel.create(example_type=0, description="example3", created_at=datetime.now())
>>> em4 = ExampleModel.create(example_type=0, description="example4", created_at=datetime.now())
>>> em5 = ExampleModel.create(example_type=1, description="example5", created_at=datetime.now())
>>> em6 = ExampleModel.create(example_type=1, description="example6", created_at=datetime.now())
>>> em7 = ExampleModel.create(example_type=1, description="example7", created_at=datetime.now())
>>> em8 = ExampleModel.create(example_type=1, description="example8", created_at=datetime.now())

#and now we can run some queries against our table
>>> ExampleModel.objects.count()
8
>>> q = ExampleModel.objects(example_type=1)
>>> q.count()
4
>>> for instance in q:
>>>     print instance.description
example5
```

```
example6
example7
example8

#here we are applying additional filtering to an existing query
#query objects are immutable, so calling filter returns a new
#query object
>>> q2 = q.filter(example_id=em5.example_id)

>>> q2.count()
1
>>> for instance in q2:
>>>     print instance.description
example5
```

Report a Bug

Users Mailing List

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

C

Symbols

A

B

C

D

F