Apache Cassandra >

CQL (Cassandra Query Language)

Download Cassandra from here and test dataset from here.

```
$ cd ~/Downloads/
$ tar xf apache-cassandra-3.10-bin.tar.gz
$ cd apache-cassandra-3.10
```

Start cassandra in foreground and keep in running. If you terminate, cassandra db will terminate as well.

```
$ bin/cassandra -f
```

Start cqlsh prompt for writing queries.

```
$ cd ~/Downloads/apache-cassandra-3.10
$ bin/cqlsh
```

View existing keyspaces. The output may be different on your system.

```
cassandra@cqlsh:demo> select * from system_schema.keyspaces ;
```

```
 keyspace_name      | durable_writes | replication
--------------------+----------------+--------------------------------
-------------------------------------------------
        system_auth |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'1'}
      system_schema |           True |
{'class': 'org.apache.cassandra.locator.LocalStrategy'}
 system_distributed |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'3'}
             system |           True |
{'class': 'org.apache.cassandra.locator.LocalStrategy'}
      system_traces |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'2'}
               demo |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'1'}

(6 rows)
```

Inside the cqlsh do the following

Create a demo keyspace

```
cqlsh> create KEYSPACE demo WITH replication = {'class': 'SimpleStrategy',
'replication_factor': 1};
```

Open another terminal view datasets

```
$ cd /Downloads/datasets/ml-latest-small
```

Sample from the dataset

```
$ head movies.csv
movieId,title,genres
1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy
2,Jumanji (1995),Adventure|Children|Fantasy
```

```
3,Grumpier Old Men (1995),Comedy|Romance
4,Waiting to Exhale (1995),Comedy|Drama|Romance
5,Father of the Bride Part II (1995),Comedy
...
```

$ head ratings.csv

```
userId,movieId,rating,timestamp
1,31,2.5,1260759144
1,1029,3.0,1260759179
1,1061,3.0,1260759182
1,1129,2.0,1260759185
...
```

Data Types

View cassandra data types
http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cql_data_types_c.html

Create table under demo keyspace

```
cqlsh> use demo;
cqlsh:demo> create table movies (movieId int primary key, title text,
genres text);
```

View existing tables

```
cqlsh:demo> DESC tables;
```

Describe movies table

```
cqlsh:demo> DESC table movies ;
```

Load movies.csv into movies table

```
cqlsh:demo> COPY movies from '~/Downloads/datasets/ml-latest-
small/movies.csv' WITH HEADER = true;
cqlsh:demo> select * from movies limit 10;
```

There was a mistake by not mentioning field in the file with those in the table in copy command. As a best practice always remember to specify the column in the order they appear in the csv load file.

Let's truncate the data and reload data.

```
cqlsh:demo> TRUNCATE movies ;
cqlsh:demo> COPY movies (movieId, title, genres) from
'~/Downloads/datasets/ml-latest-small/movies.csv' WITH HEADER = true;
```

Let's try to filter based on non primary key

```
cqlsh:demo> select * from movies where title = 'City Hall (1996)';
You get
InvalidRequest: Error from server: code=2200 [Invalid query]
message="Cannot execute this query as it might involve data filtering and
thus may have unpredictable performance. If you want to execute this query
despite the performance unpredictability, use ALLOW FILTERING"
```

Essentially the query works as expected but should not be used in general.

```
cqlsh:demo> select * from movies where title = 'City Hall (1996)' ALLOW
FILTERING;
```

So, what options we have
a. create an secondary index
b. create a materialized view

Option a: Create an index

```
cqlsh:demo> create INDEX on movies (title);
cqlsh:demo> select * from movies where title = 'City Hall (1996)';
```

Option b: Create a materialized view

```
cqlsh:demo> create MATERIALIZED VIEW demo.movies_by_title AS SELECT * from
movies where title is not null primary key (title, movieid);
```

Materialized view creates another table with copy of the data from the base table (movies). Materialized view will be updated by cassandra each time you update base table. Materialized view does not allow aggregate operation yet.

```
cqlsh:demo> select * from movies_by_title where title = 'City Hall
(1996)';
cqlsh:demo> select * from movies where movieid in (100);
```

Create ratings table.

```
cqlsh:demo> create table ratings(
    userid int,
    movieid int,
    rating float,
    rated_on bigint,
    primary key ((userId), rated_on, movieId)
) with clustering order by (rated_on desc);
```

Load data ratings.csv into ratings table

```
cqlsh:demo> COPY ratings (userid, movieid, rating, rated_on) from
'~/Downloads/datasets/ml-latest-small/ratings.csv' WITH HEADER = true;
```

Create a table using tags.csv dataset. I want see the unique tags that the users have associated with movie. Sort the tags by tag name.

```
create table movie_by_tag(
    movieid int,
    tag text,
    userid int,
    tagged_on bigint,
    primary key (movieId, tag)
);

cqlsh:demo> COPY movie_by_tag (userid, movieid, tag, tagged_on) from
'~/Downloads/datasets/ml-latest-small/tags.csv' WITH HEADER = true;
```

If you notice, the total number of records in the new table will be less than that in original dataset;

Using Timestamp

```
cqlsh:demo> create table user (id int primary key, name text);
cqlsh:demo> insert into user (id, name) values (1, 'user 1');
cqlsh:demo> select * from user;

 id | name
----+--------
  1 | user 1


cqlsh:demo> select id, name, writetime(name) from user;

 id | name   | writetime(name)
----+--------+------------------
  1 | user 2 | 1499685866217511
  2 | user 3 | 1499685905696619
```

Timestamp for the name field shows the time when the field value was set.
Timestamp is not available for the primary field column.

```
cqlsh:demo> UPDATE user USING TIMESTAMP 1499685905696619 set name = 'user
4' where id = 2;
cqlsh:demo> select id, name, writetime(name) from user;

 id | name   | writetime(name)
----+--------+------------------
  1 | user 2 | 1499685866217511
  2 | user 4 | 1499685905696619
```

Observe that the value of the name column for user with 2 has been updated but the timestamp remained as is.

Now, let's write an update statement with timestamp earlier than the existing timestamp.

```
cqlsh:demo> UPDATE user USING TIMESTAMP 1499685905696618 set name = 'user
5' where id = 2;
cqlsh:demo> select id, name, writetime(name) from user;

 id | name   | writetime(name)
----+--------+------------------
  1 | user 2 | 1499685866217511
  2 | user 4 | 1499685905696619
```

As you can see because timestamp was set to a earlier point (1499685905696618) that existing (1499685905696619), the value was ignored.

Use of TTL

Set record ttl. Note, specify the ttl value in seconds.

```
cqlsh:demo> UPDATE user USING TTL 60 SET name = 'user 10' where id = 2;
cqlsh:demo> select id, name, ttl(name) from user;
```

```
 id | name    | ttl(name)
----+---------+-----------
  1 |  user 2 |      null
  2 | user 10 |        51

(2 rows)
```

Run select after 60 secs.

```
cqlsh:demo> select id, name, ttl(name) from user;

 id | name   | ttl(name)
----+--------+-----------
  1 | user 2 |      null
```

Observe the record with id = 2 is automatically deleted.

Lightweight transaction

The Paxos protocol is implemented in Cassandra with linearizable consistency, that is sequential consistency with real-time constraints. Linearizable consistency ensures transaction isolation at a level similar to the serializable level offered by RDBMSs. This type of transaction is known as compare and set (CAS); replica data is compared and any data found to be out of date is set to the most consistent value.

```
cqlsh:demo> UPDATE user SET name = 'user 10' where id = 1 if name = 'user
1' ;

 [applied] | name
-----------+--------
     False | user 2

cqlsh:demo> INSERT INTO user (id, name) VALUES (1, 'user 101') IF NOT
EXISTS;
```

Add new column emails that is a list type

```
cqlsh:demo> alter table user add emails list<text>;

cqlsh:demo> update user set emails = emails + ['user@email.com'] where id
= 1;
cqlsh:demo> select * from user;

 id | emails              | name
----+---------------------+--------
  1 | ['user@email.com'] | user 2
```

Add a new email address to the user = 1

```
cqlsh:demo> update user set emails = emails + ['user2@gmail.com'] where id
= 1;
cqlsh:demo> select * from user;
```

```
 id | emails                                    | name
----+-------------------------------------------+--------
  1 | ['user@email.com', 'user2@gmail.com'] | user 2
```

Delete an email address for the user = 1

```
cqlsh:demo> update user set emails = emails - ['user@email.com'] where id
= 1;
cqlsh:demo> select * from user;

 id | emails               | name
----+----------------------+--------
  1 | ['user2@gmail.com'] | user 2
```

Custom Types

```
cqlsh:demo> CREATE TYPE address (street text, city text, postal_code text,
state text);
```

Add address field to the user table.

```
cqlsh:demo> alter table user add address address;

cqlsh:demo> update user set address = {street: '100 main street', city:
'ny'} where id = 1;
cqlsh:demo> select * from user;

 id | address
    | emails               | name
----+-----------------------------------------------------------------------
----+----------------------+--------
  1 | {street: '100 main street', city: 'ny', postal_code: null, state:
null} | ['user2@gmail.com'] | user 2

(1 rows)

cqlsh:demo> update user set address.street = '100 market street' where id
= 1;
```

UUID vs TimeUUID

Both UUID and TimeUUID are 128 bit value. First 64 bits in TimeUUID is replaced by timestamp. Timeuuid is sortable. UUID or TimeUUID can be created by uuid() and now() respectively.

Deleting Records from Cassandra table

Cassandra treats a delete as an insert or upsert. The data being added to the partition in the DELETE command is a deletion marker called a tombstone. The tombstones go through Cassandra's write path, and are written to SSTables on one or more nodes. The key difference feature of a tombstone: it has a built-in expiration date/time. At the end of its expiration period the tombstone is deleted as part of Cassandra's normal compaction process.

You can run delete statement by
A. delete record by primary key
B. delete record by partition key
C. delete a record or column by using TTL
D. delete records older than a certain timestamp
E. dropping keyspace and table immediately performs the delete without tombstone or GC grace period.