Cassandra Group Documentation

Cassandra install steps

1. Sudo yum -y update (Updates your box)
2. yum -y install java (Installs java up to the newest version)
3. vim /etc/yum.repos.d/datastax.repo (Makes a new .repo file for the yum install)
4. Copy EVERYTHING from this file in between these lines and paste into your datastax.repo file. Then leave and save with :wq

```
----------------------------------------------------------------------------------------------------------------------------
-
[datastax]
name = DataStax Repo for Apache Cassandra
baseurl = http://rpm.datastax.com/community
enabled = 1
gpgcheck = 0
----------------------------------------------------------------------------------------------------------------------------
-
```

5. yum -y install dsc20 (Installs Cassandra)

Helpful commands

```
sudo service cassandra start/stop/restart
```

Cassandra doesn’t like systemctl so try and use the service commands to work your way around that. Also make sure the yum install is done correctly.

Cassandra CRUD, Query, & Index

Quick Notes

1. Cassandra is a **partitioned row store**, where rows are organized into tables with a required primary key.
2. The API to cassandra is CQL, the Cassandra Query Language.
3. To connect to the cluster you can use either cqlsh or through a client driver.
4. Cqlsh is a command line interface for interacting with Cassandra through CQL
   a. Can be found in the *bin/* directory.
5. CQL offers a model close to SQL in the sense that data is put into ***tables*** containing ***rows*** of ***columns***.

Before beginning, it might be a good idea to make a small glance over this:
http://www.guru99.com/cassandra-data-model-rules.html
It's about how to Model your data. Was thinking on putting this into the document, but I feel that looking at it might be better, with the limited time we have.

A quick getting started

As briefly mentioned in Installation portion a couple commands were listed that will be used here to get started with Cassandra:

- `sudo service cassandra start`: This will start the Cassandra server.
- `sudo service cassandra stop`: This will stop the cassandra server.
- `cqlsh localhost`: This will connect you to your server and put you in Cassandra's default test cluster.

From there you are in the cqlsh shell and can start managing the database.
To leave just type in `quit`.

Keyspaces, Create, Alter & Drop

Syntax:
```
CREATE KEYSPACE <identifier> WITH <properties>
```

A keyspace is the outermost container for data in cassandra.
It is a namespace that defines data replication on nodes. A cluster contains one keyspace per node. Keyspaces have two properties **replication** and **durable_writes**.

Syntax Example:

```
CREATE KEYSPACE “KeySpace Name”
WITH replication = {'class': ‘Strategy name’, 'replication_factor' : ‘No.Of
replicas’} (;) < add one if you want to end the line, else continue it with:
AND durable_writes = ‘Boolean value’;
```

The **replication** option is to specify the replica placement strategy and the number of replicas wanted.

- `SimpleStrategy`: A simple strategy that defines a replication factor for the whole cluster. The only sub-options supported is 'replication_factor' to define that replication factor and is required.
- `NetworkTopologyStrategy`: A replication strategy that allows to set the replication factor independently for each data-center. The rest of the sub-options are key-value pairs where a key is a data-center name and its value is the associated replication factor.

With the **durable_writes** option you can instruct cassandra whether to use commitlog for updates on the current keyspace. This option is not mandatory and is default set to true.

Using a keyspace

Syntax:

```
USE <identifier>
```

Example:

```
cqlsh> USE University;
cqlsh:University>
```

Altering a keyspace

Syntax:

```
ALTER KEYSPACE <identifier> WITH <properties>
```

Syntax example:

```
ALTER KEYSPACE “KeySpace Name”
WITH replication = {'class': ‘Strategy name’, 'replication_factor' : ‘No.Of
replicas’};
```

Drop Keyspace

Syntax:

```
DROP KEYSPACE <identifier>
```

Example:

```
DROP KEYSPACE University;
```

To verify that the keyspace has been dropped you can use the **describe** command like so:

```
DESCRIBE keyspaces
```

Resources and references:
http://cassandra.apache.org/doc/latest/cql/ddl.html#create-keyspace

Tables

Syntax:
```
CREATE (TABLE | COLUMNFAMILY) <tablename>
('<column-definition>' , '<column-definition>')
(WITH <option> AND <option>)
```

Defining a column

Syntax:
```
column name1 data type,
column name2 data type,
```

Syntax example:
```
age int,
name text
```

Primary Key

- A primary key is a column that is used to uniquely identify a row.
- Defining a primary key is **mandatory** while creating a table.
- A primary key is made of one or more columns of a table.

Syntax:
```
CREATE TABLE tablename(
   column1 name datatype PRIMARYKEY,
   column2 name data type,
   column3 name data type.
   )
```
Or you could use something like this:
```
CREATE TABLE tablename(
   column1 name datatype PRIMARYKEY,
   column2 name data type,
   column3 name data type,
   PRIMARY KEY (column1)
   )
```

Example:
```
cqlsh> USE University;
cqlsh:University>; CREATE TABLE stud(
      stud_id int PRIMARY KEY,
      stud_name text,
      stud_city text,
      stud_sal varint,
      stud_phone variant
);
```

The select statement will return the schema, your table should look like this:

```
cqlsh:University> select * from stud;

 stud_id | stud_city | stud_name | stud_phone | stud_sal
---------+-----------+-----------+------------+----------
```

Ordering that table

Syntax:

```
) WITH CLUSTERING ORDER BY ( COLUMN (ASC | DESC))
```

You can order the queries you perform on that table by telling that table what column you want it to sort by and whether you want it ascending or descend.

The ascending order will be more efficient than descending.

You put this part right after the table creation chunk, example:

```
CREATE TABLE loads (
    machine inet,
    cpu int,
    mtime timeuuid,
    load float,
    PRIMARY KEY ((machine, cpu), mtime)
) WITH CLUSTERING ORDER BY (mtime DESC);
```

Altering that Table

Syntax:

```
ALTER (TABLE | COLUMNFAMILY) <tablename> <instruction>
```

With the Alter command you can do two operations:

- Add a column
- Drop a column

While adding columns you must make sure that the column name isn't conflicting with an already existing column, and that the new table is not defined with compact storage option.

Example:

```
cqlsh:University> ALTER TABLE stud
   ... ADD stud_email text;
```

Dropping a column

Syntax:

```
ALTER table name DROP column name;
```

Example:

```
cqlsh:University> ALTER TABLE stud DROP stud_email;
```

Dropping a Table

Syntax:

```
DROP TABLE <tablename>
```

Example:

```
cqlsh:University> DROP TABLE stud;
```

This will drop the whole stud table. Simple.

To verify if the table was successfully dropped you can use the describe command.

```
DESCRIBE COLUMNFAMILIES;
```

Truncating a table

Syntax:

```
TRUNCATE <tablename>
```

When you truncate a table all the **rows** of the table are deleted permanently. The columns will remain.

Reference:

http://www.guru99.com/cassandra-table-create-alter-drop-truncate.html

- C: INSERT
- R: SELECT
- U: UPDATE
- D: DELETE

Other Documented CQLSH shell commands here:

https://www.tutorialspoint.com/cassandra/cassandra_cqlsh.htm

Insert

Syntax:

```
insert into KeyspaceName.TableName(ColumnName1, ColumnName2, ColumnName3 ....)
values (Column1Value, Column2Value, Column3Value ....)
```

Command ‘Insert into’ writes data in cassandra columns in row form. It will store only in the columns that are given by the user.
You have to specify the primary key column.

It will not take any space for not given values and <u>no</u> results are returned after insertion.
Example:

```
Insert into University.Student(StudentId,Name,dept,Semester)
values(12345,’Joel’,’CS’,1);
```

Should make a table that looks somewhat like this:

```
 studentid | dept      | name                 | semester
-----------+-----------+----------------------+---------------
 12345     | CS        | Joel                 | 1
```

Update and Upsert

Syntax:

```
Update KeyspaceName.TableName
Set ColumnName1=new Column1Value,
      ColumnName2=new Column2Value,
      ColumnName3=new Column3Value,
       .
       .
       .
Where ColumnName=ColumnValue
```

If no results are returned after updating, it means that the data is successfully updated otherwise an error will be returned.
Column values are changed in `‘Set’` clause while data is filtered with `‘Where’` clause.
Example:

```
Update University.Student
       ... Set name=’Hayden’
       ... where studentid=12345;
```

Should edit the table created to look like this now:

```
 studentid | dept      | name                 | semester
-----------+-----------+----------------------+---------------
 12345     | CS        | Hayden               | 1
```

Cassandra Group Documentation

Upsert

Upsert will insert a row if a primary key does not exist already otherwise if a primary key already exists, it will update that row.

Delete

Syntax(s):

1.

```
Delete from KeyspaceName.TableName
      Where ColumnName1=ColumnValue
```

2.

```
Delete ColumnNames from KeyspaceName.TableName
      Where ColumnName1=ColumnValue
```

The first version will delete one or more rows depending on the data filtration in the `'Where'` clause.
The second version will delete some columns from the table.

Example:

```
Delete from University.Student where studentid=12345
```

This would delete the only entry in the table and empty it.

Read

Syntax:

```
Select ColumnNames from KeyspaceName.TableName Where ColumnName1=Column1Value AND
      ColumnName2=Column2Value AND
      .
      .
```

In cassandra data retrieval is a sensitive issue. The column is filtered in cassandre by creating an index on non-primary key columns.

```
cqlsh> select * from University.Student;
 studentid | dept     | name                 | semester
-----------+----------+----------------------+---------------
 1         | CS       | Hayden               | 1

 2         | CS       | Jasmine              | 3
```

Here is an example retrieval from Student table without data filtration. Two records were grabbed.

Now here is an example of the Student table retrieved with data filtration.

```
cqlsh> select * from University.Student where name=’Jasmine’;
 studentid | dept     | name                  | semester
-----------+----------+-----------------------+---------------
 2         | CS       | Jasmine               | 3
```

Data is filtered by the name column. All the records are retrieved that has name equaled to Jasmine.

Index

Syntax:

```
Create index IndexName on KeyspaceName.TableName(ColumnName);
```

Create index will create an index on the column you choose. If the data already exists for the column you want, cassandra creates indexes on the data during the ‘create index’ statement.

Example:

```
cqlsh> select * from University.Student where dept=’CS’
```

If you tried to filter a column without creating the index, you’ll get an error.
You’ll need to set the index like so:

```
cqlsh> Create index DeptIndex on University.Student(dept);
```

This would allow the command to return:

```
 studentid | dept     | name                  | semester
-----------+----------+-----------------------+---------------
 12345     | CS       | Joel                  | 1
```

Drop Index

The command will drop the specified index. If the index was not given a name during creation, the index will be given the default name of “TableName_ColumnName_idx”.

Syntax:

```
Drop index IF EXISTS KeyspaceName.IndexName
```

If the index does not exist, it will return an error unless “If Exists” is used.
During index creation you have to specify keyspace name with the index name, otherwise the index will be dropped from the current keyspace
For more on Indexing through something like java check here:
https://www.tutorialspoint.com/cassandra/cassandra_create_index.htm

Importing Data

Summary: Cassandra generally isn't considered the most ETL friendly platform, however, there are many options out there to be able to perform the operations. Specifically here, we will go over importing data from a CSV, and a little bit about SSTable transfers between Cassandra setups, though be aware, there's many other options out there, such as Talend and other apps.

The COPY command: (source) The copy command is generally good for importing from flat files, such as CSVs, TSVs, and the like. It can be used for both extracting and loading data, both of which we will briefly cover here

Importing Data: From within the Cassandra CQL shell use this:

```
COPY <column family name> [ ( column [, …] ) ]   FROM ( '<filename>' | STDIN )   [ WITH <option>='value' [AND …] ];
```

Example:

```
COPY airplanes (name, manufacturer, year, mach) FROM 'temp.csv';
```

Extracting Data: From within the Cassandra CQL shell use this:

```
COPY <column family name> [ ( column [, …] ) ]   TO ( '<filename>' | STDOUT )   [ WITH <option>='value' [AND …] ];
```

Example:

```
COPY airplanes (name, mach, year, manufacturer) TO 'temp.csv'
```

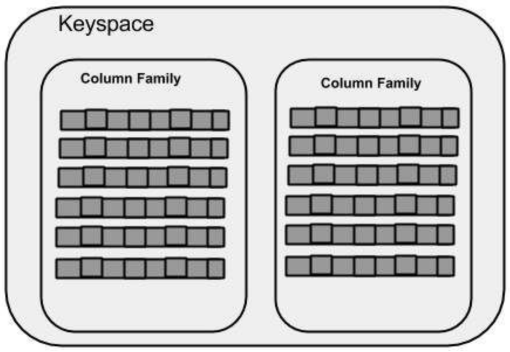
The Cassandra Bulk Loader:(source) The sstableloader is a tool that is primarily to be used with a distributed cluster of Cassandra nodes. It's purpose for being is to be able to load a set of sstables into a cluster, while only sending each node the data it needs to store, as opposed to sending the full data to every server, then letting it reduce down to only the data it needs to store according to the replication strategy of the cluster.

The general process for being able to use the cassandra bulk loader is as follows: Firstly you must acquire data in the sstable format, or convert it into the sstable format as explained here. After that has been completed, simply run sstableloader <dir containing your data>. After that just wait a short bit for Cassandra to use it’s gossip protocol, and the data will be shared around the cluster and stored according to it’s replication strategy.

Data Model

The Cassandra data model is deeply embedded with the idea of distributing data across many nodes, to provide a distributed database, resulting in fault tolerance, and enough redundancy to withstand hardware failures.

(source)



As a result of this the data model starts on the **cluster** level. Within a cluster are of course **nodes**, or the individual servers supporting the database. These nodes are arranged into a **ring**, much like Riak's data model. The Ring contains one or several **keyspace**s. The keyspace is the level at which the *replication factor* (or number of copies of data to hold in the cluster, similar to Riak's N value,) is stored and configured. In addition the keyspace is also responsible for the *replica placement strategy* (which is used for ensuring data is distributed in the most desirable way to the database admins [if you have multiple data centers you may wish to have a copy of the data in each data center, incase one of them has a catastrophic failure]). Finally the Keyspace also contains several column Families, which is where the actual structure of the data begins to come into place.

A **column family** stores an ordered collection of rows, and is also used to determine how much caching should be done with the data to help optimize data reads, and whether or not they should be pre-loaded. A **row** contains several columns, which in turn contain values. Since Cassandra doesn't enforce any sort of data structure at this level (such as all employees MUST have a name) I find it more helpful to think of these "rows" more so like Mongo's JSON objects, such that we have the row being the object, and the columns become keys to their respective values within the object.

Additional comparisons of Cassandra to RDBMS systems can be found in the (source) for this bit, as I've tried to construct it in the terms we've come to understand throughout this class, however it's there for you if you wanna read more.

Useful Links

- **http://www.datastax.com/dev/blog/ways-to-move-data-tofrom-datastax-enterprise-and-cassandra**
- **http://www.datastax.com/dev/blog/bulk-loading**
- **https://www.tutorialspoint.com/cassandra/cassandra_data_model.htm**

Other links that weren't the direct source for this document, but are still useful to read and understand.

- **http://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling**
- **https://wiki.apache.org/cassandra/MemtableSSTable**

```
package com.marxmart.persistence;
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Host;
import com.datastax.driver.core.Metadata;
import com.datastax.driver.core.Session;
import static java.lang.System.out;
/**
 * Class used for connecting to Cassandra database.
 */
public class CassandraConnector
{
   /** Cassandra Cluster. */
   private Cluster cluster;
   /** Cassandra Session. */
   private Session session;
   /**
      * Connect to Cassandra Cluster specified by provided node IP
      * address and port number.
      *
      * @param node Cluster node IP address.
      * @param port Port of cluster host.
      */
   public void connect(final String node, final int port)
   {
      this.cluster =
Cluster.builder().addContactPoint(node).withPort(port).build();
      final Metadata metadata = cluster.getMetadata();
      out.printf("Connected to cluster: %s\n", metadata.getClusterName());
      for (final Host host : metadata.getAllHosts())
      {
      out.printf("Datacenter: %s; Host: %s; Rack: %s\n",
            host.getDatacenter(), host.getAddress(), host.getRack());
      }
      session = cluster.connect();
   }
   /**
      * Provide my Session.
      *
      * @return My session.
      */
   public Session getSession()
   {
      return this.session;
   }
   /** Close cluster. */
   public void close()
   {
      cluster.close();
```

```
    }
}
public static void main(final String[] args)
{
  final CassandraConnector client = new CassandraConnector();
  final String ipAddress = args.length > 0 ? args[0] : "localhost";
  final int port = args.length > 1 ? Integer.parseInt(args[1]) : 9042;
  out.println("Connecting to IP Address " + ipAddress + ":" + port + "...");
  client.connect(ipAddress, port


final String createMovieCql =
        "CREATE TABLE movies_keyspace.movies (title varchar, year int, description
varchar, "
  + "mmpa_rating varchar, dustin_rating varchar, PRIMARY KEY (title, year))";
client.getSession().execute(createMovieCql);
);
  client.close();
}
package dustin.examples.cassandra;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.Row;
import java.util.Optional;
import static java.lang.System.out;
/**
* Handles movie persistence access.
*/
public class MoviePersistence
{
  private final CassandraConnector client = new CassandraConnector();
  public MoviePersistence(final String newHost, final int newPort)
  {
        out.println("Connecting to IP Address " + newHost + ":" + newPort + "...");
        client.connect(newHost, newPort);
  }
  /**
        * Close my underlying Cassandra connection.
        */
  private void close()
```

```
   {
          client.close();
   }
}
```

So its code to connect create and maintain the Cassandra DB. From what I’ve seen unlike the other apps you have to create and import a java driver for it to work as well as import several jars. Which can be easily done with maven. And you do need to go in to the Cassandra server files to get the port that its running on. Other then that its pretty straight forward.