Cassandra

Cassandra is a columnar NoSQL store. Cassandra strives to be highly available and easily scalable. It provides a SQL-like interface with the Cassandra Query Language (CQL). While it looks like a SQL data store, it provides many additional benefits. One of the key benefits is that every node in a Cassandra cluster acts like every other node. Writes can be sent to any node. Cassandra is a powerful NoSQL data store.

Setup

This will help you install and setup Cassandra on ubuntu 16.04.

Install Java

On every node that Cassandra will be installed on you will need to install Java.
You can install Java by running the following commands

```
sudo apt-get update
sudo apt-get install openjdk-8-jdk
```

You can then check to make sure java is installed with

```
java -version
```

Installation

Add the repository for Cassandra to every node

```
echo "deb http://debian.datastax.com/community stable main" | sudo tee -a
/etc/apt/sources.list.d/datastax.community.list
```

Add the Cassandra repository keys to every node

```
curl -L https://debian.datastax.com/debian/repo_key | sudo apt-key add -
```

Update the repositories and install python support

```
sudo apt-get update
wget http://launchpadlibrarian.net/109052632/python-support_1.0.15_all.deb
sudo dpkg -i python-support_1.0.15_all.deb
```

Install Cassandra

```
sudo apt-get install dsc21=2.1.5-1 Cassandra=2.1.5 Cassandra-tools=2.1.5 -y
```

Clustering

In order to setup Cassandra for clustering, first stop Cassandra

```
sudo service Cassandra stop
```

Then we can edit the Cassandra config file in `/etc/Cassandra/Cassandra.yaml` on every node

```
vim /etc/Cassandra/Cassandra.yaml
```

Find `-seeds:` and change the `"127.0.0.1"` to have a comma separated list of the ip addresses of all the nodes for the cluster

In each nodes config file, find `listen_address` and set it to the ip address of that node. Then find the `rpc_address`and remove `localhost`, leaving it blank. Save all of the files.

Restart Cassandra

```
sudo service Cassandra restart
```

If everything worked, you should be able to run

```
sudo nodetool status
```

and see output similar to

```
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address        Load       Tokens       Owns (effective)  Host ID
Rack
UN  137.112.89.75  468.63 KiB  256          68.3%             e4059108-8bcf-4412-
a4b7-0eb041e6ca20  rack1
UN  137.112.89.76  318.69 KiB  256          69.5%             60f1e39f-3898-4f60-
87e5-99aff0a44545  rack1
```

```
UN  137.112.89.78  320.06 KiB  256          62.2%             5633afff-c470-4a97-
b529-12733fe9a5b4  rack1
```

Python Driver

Install the python driver

```
sudo -H pip install Cassandra-driver
```

Export a variable to override bundled driver

```
export CQLSH_NO_BUNDLED=true
```

You should be able to connect using python by

```
from Cassandra.cluster import Cluster
cluster = Cluster(['<Node1 IP Address>', '<Node2 IP Address>'])
session = cluster.connect()
```

Any CQL queries can be executed with `.execute` by passing in the query. Cassandra also supports the notion of prepared statements, which can be used to save data transfer from some language to Cassandra.

```
session.execute("SELECT * FROM 'table';")
```

Basic CQL Commands

The Cassandra Query Language (CQL) looks very much like standard SQL. Many of the standard create, update, insert, select queries will look the same, often times with extra arguments available. A full list of the options provided by CQL can be found online.

Keyspaces

One of the key differences between CQL and SQL is the notion of a keyspace. A keyspace in Cassandra is namespace that defines the data replication on nodes. This acts similarly to the notion of creating a database in SQL. Keyspaces can be created with

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

Where properties are either `replication` or `durable_writes`. You can learn more about how they work by checking the documentation for create keyspace.

Create

A create table (column family) in Cassandra looks exactly like a SQL create table command with an additional `WITH` clause which can be used to define different table properties. The additional table properties are optional but can be used to tune data handling, including I/O operations, compression, and compaction.

Insertion into a table is similar to a standard SQL insert command. The fields composing the primary key must be included. Inserts will overwrite existing data if it already exists. Insert does not support incrementing counters.

- An example command creating a table call cyclist_name where all ids are unique.

```
CREATE TABLE cyclist_name (
   id UUID PRIMARY KEY,
   lastname text,
   firstname text );
```

- An example `INSERT` query.

```
INSERT INTO armor(name, type, defense) VALUES('Rathian Hat', 'Helm', 10);
```

Read

Select statements can be used to query Cassandra. However, cql does not support arbitrary where clauses. If you attach a where clause onto something that Cassandra knows that it cannot do efficiently, it will warn you with a message similar to

```
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"
```

This usually means that you need to add a secondary index or reconsider the way your data is modeled. This is because Cassandra cannot ensure good performance for this query. If you know that most of your data will fit the `WHERE` clause, it may be okay to just add `ALLOW FILTERING` to your query. Otherwise you should **NOT** just allow filtering.

- An example of a `SELECT` statement in CQL.
  - *This query is likely to give the above warning*

```
SELECT * FROM armor WHERE name LIKE 'Rathian';
```

Update

`UPDATE` statements look and perform much like insert statements. The key difference is that updates will allow counters. Both update and insert can specify TTL (Time to Live). TTL works as an optional expiration date for values inserted into the column. These are on the individual values and not the whole column. Updates to the value will reset the TTL.

```
UPDATE Movies SET title = 'New Title' WHERE movieID = 11;
```

Delete

`DELETE` works by marking the data with a tombstone and then after a grace period, removes it.

```
DELETE name FROM armor WHERE id = 12;
```

Indexing

Indexes can be created on column fields to optimize query results. Cassandra will also allow custom indices for different types of searches. For instance, if you want to query something that isn't a primary key using `LIKE`, you'll want to index it. An example custom index to allow this is below.

```
CREATE CUSTOM INDEX employee_firstname_idx ON bth.employee (firstname) USING
'org.apache.Cassandra.index.sasi.SASIIndex' WITH OPTIONS = {'analyzer_class':
'org.apache.Cassandra.index.sasi.analyzer.StandardAnalyzer', 'case_sensitive':
'false'};
```

Data Modeling

Data modeling in Cassandra is almost completely query driven. Typically, when designing a data model, the goal is to minimize writes and data duplication. These are not as big a concern with Cassandra. Since Cassandra will allow you to write anywhere, they are pretty efficient and extra writes can be used to help improve read queries. Cassandra does not support `JOIN` operations so data duplication is expected and is not a bad thing, because Cassandra expects it. However, you still want the data to be spread around evenly and we want to minimize the number of partitions that have to be read from in order to answer a query.

Partitioning

A partition in Cassandra is a group of rows that share the same partition key. The partition key is the first component of the `PRIMARY_KEY`. Cassandra will spread the partitions around the cluster based on a hash of the partition key. Because of that, we want a good primary key for the hashing. We also want to make sure that our queries are not running on multiple nodes, that is to say, ideally a query only reads from a single partition. These two goals are at odds and will need to be balanced.

Query Based Modeling

The best way to solve the partitioning balance problem is to design your tables for the queries that will be run against them. It is important to think about the types of queries that will be run. For instance are you filtering on a field, ordering by one, or does something need to be unique?

One good place to start with is by looking at creating a table to answer a query. If you have a set of data where you want to filter on two different fields, there is no reason not to have two separate tables already constructed to answer either query. Remember, writes are fairly inexpensive with Cassandra.

Remember to consider how many elements will be under each partition key. Having 100 partition keys where 80% of the data is under 1 key is still a bad thing. In order to help fix this, you can add additional columns to the primary key to form a compound partition key.

Scalability Options

Cassandra clusters operates in a ring. All nodes in the ring are considered equal. There is no master node so there is no reason to worry about which node you are writing to or reading from. There is no single point of failure.

Cassandra also supports the notion of a Data Center with options for both virtual and physical data centers. A cassandra cluster can span multiple data centers. Cassandra will allow us to query either the entire cluster or just the local data center. The benefit of this is that we can act like we are talking to the same cluster regardless of where we are, but only query the local nodes if we do not need the information from a data center in another country. This can greatly decrease the latency and amount of data needed to be transferred.

Cassandra also supports replication out of the box. In cassandra these replicas are complete copies of the data such that if a single node goes down, the data can be retrieved from another node. These replica sets are customizable at the table/column-family level. This customization allows for choosing how many copies of the data should exist and where on the server-rack they should be.
Replication can be in single data center, across multiple data centers, or across different cloud providers. When replicating across multiple data centers, the number of copies in each data center is configurable.

Scaling cassandra can be done in one of two ways, horizontally or vertically. Horizontal scaling in cassandra is adding more data centers. Vertical scaling is adding more nodes to a single data center. With cassandra there are no special cares that need to be taken when adding a new node, as it comes with a load balancer that will ensure the node takes on a portion of the current data-set when it comes online.

Cassandra will be eventually consistent using asynchronous updating. We can write to any node which will propose the write request. The write request will be processed once a configured number of nodes accepts the write. Cassandra automatically handles a node going down through a self-healing process which includes scrubbing any corrupt data then reattaching the node to the ring just like a new node being added. This ensures that if a write was accepted by the number of nodes specified, which can be done on a by-query basis using consistency levels, the write will eventually be persisted. The consistency levels are used to ensure that the set of nodes have persisted the write to their commit log and memtable before verifying that the write occurred. We can also use read consistencies to ensure that we never read stale data, configuring a number of nodes that need to acknowlegde the value returned for a read query.

Walk-through Lab (The Library Example)

Let's say that we want to use Cassandra as a database for a library system. Here are some of the features we might want to have.

1. Add book(Title, Author, ISBN, #of Pages) to library
2. Delete book from library
3. Edit book information
4. Search by title, author, or isbn

The first thing to do is to design the data model. Looking at the library system, it seems like we want to be able to do a lot of reads, with probably not as many writes. For instance, consider how often book information should change. Because of this, it is probably okay to make our writing complicated and include tables that help optimize our sorting and searching functions.

Since we want to search or books by any of Title, Author, ISBN, or Number of Pages, we can create tables for each of those. Each row in the below table will become a table in our data model. Cassandra supports lists, sets, and maps as well as User Defined Data types (UDTs) so we can easily use a set for the authors of a book.

Primary Key			
isbn : int	authors : set	title : text	pages : int
title : text	isbn : int	title : text	pages : int
pages : int	isbn : int	authors : set	title : text
author : text, isbn : int		title : text	pages : int

This will optimize our ability to query the data set. Here we can look for a book by any of the fields and we only have to query one table and one partition. Note that one table will have author as a primary key. If a book has two authors, it will show up in this table multiple times. That is okay for our purposes as it makes the query perform better. The author table will have to have a composite key so that we can have the same author linked to different ISBNs.

NOTE: There is one additional feature that we might want with this database. We might want the ability to return results in a sorted order. This is something that Cassandra does not support well. It is possible to attempt to force Cassandra to return things in a sorted order using tactics like introducing dummy partition keys, or using a Byte Ordered Partition (BOP) this is **not** a good idea as it introduces load balancing problems and encourages bad data modeling. Data in Cassandra can only be sorted within a partition key. That is, if you want results to be sorted, you can specify a clustering column to sort on or use the `ORDER BY` clause when the `WHERE` clause specifies a partition to sort.

```
-- This query will be allowed for our tables because the WHERE clause identifies a
partition and the clustering key 'isbn' allows ordering
SELECT * FROM authortobook WHERE author = 'J. R. R. Tolkein' ORDER BY isbn;
```

```
-- The following query is something that you might want to try, but will not work
SELECT * FROM isbntobook WHERE title = 'The Two Towers' ORDER BY authors;
-- This will inform you that this command requires 'ALLOW FILTERING', which as
discussed earlier is a bad idea. In this case, adding ALLOW FILTERING still will
not help. Run the following:
SELECT * FROM isbntobook WHERE title = 'The Two Towers' ORDER BY authors ALLOW
FILTERING
-- The next error that you will receive will inform you that ORDER BY is only
supported when the partition key is restricted to either = or IN. This is telling
you that the WHERE clause must restrict the query to some partition(s)
```

Queries to Create the Library

0. Create a Keyspace and tables
    - We need to create the keyspace then create the tables for our books. Note that putting single quotes around things will keep the case, otherwise Cassandra will remove the casing.

```
-- Create a keyspace called library with 2 replicas on multiple nodes
CREATE KEYSPACE library WITH replication = {'class': 'SimpleStrategy',
'replication_factor': 2};
-- Switch to that keyspace
USE library;
-- Create our table for ISBN searches
CREATE TABLE isbntobook (isbn int, title text, pages int, authors set<text>,
PRIMARY KEY (isbn));
-- Create the table for title searches
CREATE TABLE titletobook (isbn int, title text, pages int, authors set<text>,
PRIMARY KEY(title, isbn));
-- Create the table for pages searches
CREATE TABLE pagestobook (isbn int, title text, pages int, authors set<text>,
PRIMARY KEY(pages, isbn));
-- Create table for author searches
CREATE TABLE authortobook (isbn int, title text, pages int, author text, PRIMARY
KEY(author, isbn));
-- To verify that all of the tables were created successfully and look correct we
can use
DESCRIBE KEYSPACE;
-- This will output the schema for the tables as well as show the settings for
each table
```

1. Add book(Title, Author, ISBN, #of Pages) to library
    - In order to add a book, we will need to use multiple insert statements.

```
-- Insert book The Two Towers
INSERT INTO isbntobook (isbn, title, pages, authors) VALUES (0395315557, 'The Two
Towers', 878, {'J. R. R. Tolkein'});

INSERT INTO titletobook (isbn, title, pages, authors) VALUES (0395315557, 'The Two
Towers', 878, {'J. R. R. Tolkein'});

INSERT INTO pagestobook (isbn, title, pages, authors) VALUES (0395315557, 'The Two
Towers', 878, {'J. R. R. Tolkein'});

INSERT INTO authortobook (isbn, title, pages, author) VALUES (0395315557, 'The Two
Towers', 878, 'J. R. R. Tolkein');

-- A duplicate Two Towers rip off with a different isbn for demo purposes
INSERT INTO isbntobook (isbn, title, pages, authors) VALUES (0395315558, 'The Two
Towers', 878, {'J. R. R. Tolkien'});

INSERT INTO titletobook (isbn, title, pages, authors) VALUES (0395315558, 'The Two
Towers', 878, {'J. R. R. Tolkien'});

INSERT INTO pagestobook (isbn, title, pages, authors) VALUES (0395315558, 'The Two
Towers', 878, {'J. R. R. Tolkien'});

INSERT INTO authortobook (isbn, title, pages, author) VALUES (0395315558, 'The Two
Towers', 878, 'J. R. R. Tolkien');

-- Inserting a book with multiple authors
INSERT INTO isbntobook (isbn, title, pages, authors) VALUES (1501192272, 'The
Talisman', 560, {'Stephen King', 'Peter straub'});

INSERT INTO titletobook (isbn, title, pages, authors) VALUES (1501192272, 'The
Talisman', 560, {'Stephen King', 'Peter straub'});
INSERT INTO pagestobook (isbn, title, pages, authors) VALUES (1501192272, 'The
Talisman', 560, {'Stephen King', 'Peter straub'});
INSERT INTO authortobook (isbn, title, pages, author) VALUES (1501192272, 'The
Talisman', 560, 'Peter straub');
INSERT INTO authortobook (isbn, title, pages, author) VALUES (1501192272, 'The
Talisman', 560, 'Stephen King');

-- Inserting another book
INSERT INTO isbntobook (isbn, title, pages, authors) VALUES (0345339703, 'The
Fellowship of the Ring', 967, {'J. R. R. Tolkein'});

INSERT INTO titletobook (isbn, title, pages, authors) VALUES (0345339703, 'The
Fellowship of the Ring', 967, {'J. R. R. Tolkein'});

INSERT INTO pagestobook (isbn, title, pages, authors) VALUES (0345339703, 'The
Fellowship of the Ring', 967, {'J. R. R. Tolkein'});

INSERT INTO authortobook (isbn, title, pages, author) VALUES (0345339703, 'The
Fellowship of the Ring', 967, 'J. R. R. Tolkein');
```

2. Delete book from library
    - Note that we have to use the primary key in the query. Cassandra will not allow a query like `DELETE FROM titletobook where isbn = 0345339703;`. You need to provide all parts of the partition key.

```
DELETE FROM isbntobook where isbn = 0345339703;

DELETE FROM titletobook where title = 'The Fellowship of the Ring' and isbn =
0345339703;

DELETE FROM pagestobook where isbn = 0345339703 and pages = 967;

DELETE FROM authortobook where isbn = 0345339703 and author = 'J. R. R. Tolkein';
```

3. Edit book information
    - Updates become a little tricky because of how the other attributes are being used as keys in other tables.
    - When we change a value, one of our tables will require a delete and reinsert, because we cannot change primary keys.
    - We also have to be careful to change the `set` of authors correctly in the other queries.

```
-- Let's change the author Peter straub to fix his last name
UPDATE isbntobook SET authors = authors - {'Peter straub'} WHERE isbn =
1501192272;
UPDATE isbntobook SET authors = authors + {'Peter Straub'} WHERE isbn =
1501192272;

UPDATE pagestobook SET authors = authors - {'Peter straub'} WHERE isbn =
1501192272 and pages = 560;
UPDATE pagestobook SET authors = authors + {'Peter Straub'} WHERE isbn =
1501192272 and pages = 560;

UPDATE titletobook SET authors = authors - {'Peter straub'} WHERE isbn =
1501192272 and title = 'The Talisman';
UPDATE titletobook SET authors = authors + {'Peter Straub'} WHERE isbn =
1501192272 and title = 'The Talisman';

DELETE FROM authortobook WHERE isbn = 1501192272 and author = 'Peter straub';
INSERT INTO authortobook (isbn, title, pages, author) VALUES (1501192272, 'The
Talisman', 560, 'Peter Straub');
```

4. Search by title, author, or isbn
    - Title

    ```
    -- Return books by this title. Should return 2 books; the real and fake
    copies
    SELECT * FROM titletobook WHERE title = 'The Two Towers';
    ```

    - Author

    ```
    ```

```
-- Note this doesn't return the copy of The Two Towers because the authors
name is different
SELECT * FROM authortobook where author = 'J. R. R. Tolkein';
```

- Isbn

```
SELECT * FROM isbntobook where isbn = 1501192272;
```

Taking the Lab Farther (Adding Borrowers)

Now that you have an idea of the commands to make a book store, try implementing this in a python script (see installation section for the driver and examples) and then implementing borrowers.

Some things that the application should be able to do with borrowers.

1. Add Borrower's (Name, Username, Phone) to library
2. Delete Borrowers from library
3. Edit Borrower information
4. Search by name, username
5. Allow Borrowers to checkout books (Can only checkout if a book is available) and return books.
6. Track number of books checked out by a given user & Track which user has checked out a book

Note that deleting and updating become more difficult when borrowers are allowed to checkout books. How should you handle a request to delete a borrower with books checked out?

After completing this, try setting up prepare statements in your python driver. This is one of 4 simple rules suggested by DataStax. The prepared statements will allow only the fields to be transferred instead of the whole query. This can improve the amount of data that has to be transferred.