ADVANCED DATABASE MANAGEMENT SYSTEM LAB

ASSIGNMENT-1

GROUP 1
ABHILASH JOHN
ABHISHEK M
AGNA C ANTONY
AMAL BHAS
AMAL S

SUBMITTED BY

AGNA C ANTONY
20MCA203

Assignment No: 1

Design database schemas and implement min 10 queries using Hive/ Hbase/ Cassandra column-based databases

Aim
To Design database schemas and implement queries using Cassandra databases

Objective(s)	
1	Study of NOSQL Cassandra.
2	Study the procedure to execute a query using Apache Cassandra.
3	Execute min 10 queries using Cassandra column-based database.

1.STUDY OF NOSQAL CASSANDRA

Apache Cassandra is a distributed database management system that is built to handle large amounts of data across multiple data centres and the cloud. Key features include:

- Highly scalable
- Offers high availability
- Has no single point of failure

Written in Java, it's a NoSQL database offering many things that other NoSQL and relational databases cannot. Cassandra stands out among database systems and offers some advantages over other systems. Its ability to handle high volumes makes it particularly beneficial for major corporations. As a result, it's currently being used by many large businesses including Apple, Facebook, Instagram, Uber, Spotify, Twitter, Cisco, Rackspace, eBay, and Netflix.

NoSQL Database

A NoSQL, often referred to as "not only SQL", database is one that stores and retrieves data without requiring data to be stored in tabular format. Unlike relational databases, which require a tabular format, NoSQL databases allow for unstructured data. This type of database offers:

- A simple design
- Horizontal scaling
- Extensive control over availability

NoSQL databases do not require fixed schema, allowing for easy replication. With its simple API, I like Cassandra for it's overall consistency and its ability to handle large amounts of data.

That said, there are pros and cons of using this type of database. While NoSQL databases offer many benefits, they also have drawbacks. Generally, NoSQL databases:

- Only support simply query language (SQL)
- Are just "eventually consistent

- Don't support transactions

Nevertheless, they are effective with huge amounts of data and offer easy, horizontal scaling, making this type of system a good fit for many large businesses. Some of the most popular and effective NoSQL databases include:

- Apache Cassandra
- Apache HBase
- MongoDB

Cassandra is one of the most efficient and widely-used NoSQL databases. One of the key benefits of this system is that it offers highly-available service and no single point of failure. This is key for businesses that can afford to have their system go down or to lose data. With no single point of failure, it offers truly consistent access and availability.

Another key benefit of Cassandra is the massive volume of data that the system can handle. It can effectively and efficiently handle huge amounts of data across multiple servers. Plus, it is able to fast write huge amounts of data without affecting the read efficiency. Cassandra offers users "blazingly fast writes," and the speed or accuracy is unaffected by large volumes of data. It is just as fast and as accurate for large volumes of data as it is for smaller volumes.

Another reason that so many enterprises utilize Cassandra is its horizontal scalability. Its structure allows users to meet sudden increases in demand, as it allows users to simply add more hardware to accommodate additional customers and data. This makes it easy to scale without shutdowns or major adjustments needed. Additionally, its linear scalability is one of the things that helps to maintain the system's quick response time.

Some other benefits of Cassandra include:

- **Flexible data storage.** Cassandra can handle structured, semi-structured, and unstructured data, giving users flexibility with data storage.
- **Flexible data distribution.** Cassandra uses multiple data centres, which allows for easy data distribution wherever or whenever needed.
- **Supports ACID.** The properties of ACID (atomicity, consistency, isolation, and durability) are supported by Cassandra.

Cleary, Apache Cassandra offers some discrete benefits that other NoSQL and relational databases cannot. With continuous availability, operational simplicity, easy data distribution across multiple data centres, and an ability to handle massive amounts of volume, it is the database of choice for many enterprises.

How does Cassandra work?

Apache Cassandra is a peer-to-peer system.

The basic architecture consists of a cluster of nodes, any and all of which can accept a read or write request. This is a key aspect of its architecture, as there are no master nodes. Instead, all nodes communicate equally.

While nodes are the specific location where data lives on a cluster, the cluster is the complete set of data centers where all data is stored for processing. Related nodes are grouped together in data centers. This type of structure is built for scalability and when additional space is needed, nodes can simply be added. The result is that the system is easy to expand, built for volume, and made to handle concurrent users across an entire system.

Its structure also allows for data protection. To help ensure data integrity, Cassandra has a commit log. This is a backup method and all data is written to the commit log to ensure data is not lost. The data is then indexed and written to a **memtable**. The memtable is simply a data structure in the memory where Cassandra writes. There is one active memtable per table.

When memtables reach their threshold, they are flushed on a disk and become immutable SSTables. More simply, this means that when the commit log is full, it triggers a flush where the contents of memtables are written to SSTables. The commit log is an important aspect of Cassandra's architecture because it offers a failsafe method to protect data and to provide data integrity.

CASSANDRA ARCHITECTURE

The design goal of Cassandra is to handle big data workloads across multiple nodes without any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster.

- All the nodes in a cluster play the same role. Each node is independent and at the same time interconnected to other nodes.

- Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.

- When a node goes down, read/write requests can be served from other nodes in the network.

Data Replication in Cassandra

In Cassandra, one or more of the nodes in a cluster act as replicas for a given piece of data. If it is detected that some of the nodes responded with an out-of-date value, Cassandra will return the most recent value to the client. After returning the most recent value, Cassandra performs a **read repair** in the background to update the stale values.

The following figure shows a schematic view of how Cassandra uses data replication among the nodes in a cluster to ensure no single point of failure.
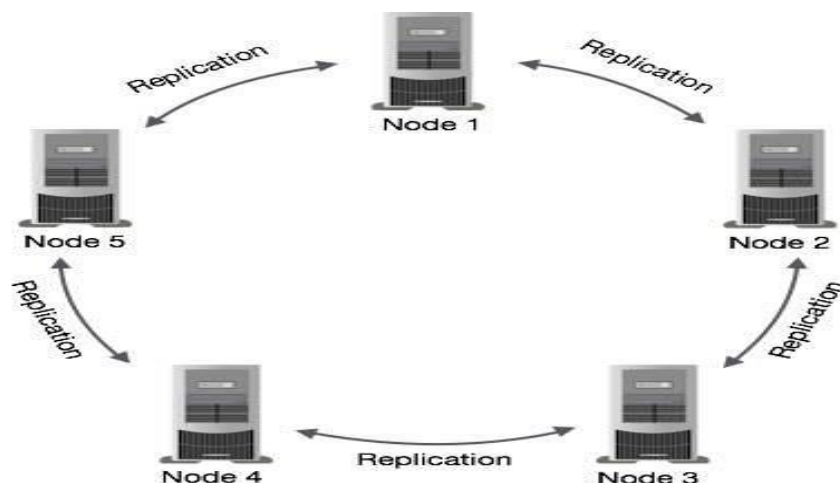
Cassandra uses the **Gossip Protocol** in the background to allow the nodes to communicate with each other and detect any faulty nodes in the cluster.

Components of Cassandra

The key components of Cassandra are as follows −

- **Node** − It is the place where data is stored.

- **Data centre** − It is a collection of related nodes.

- **Cluster** − A cluster is a component that contains one or more data centers.

- **Commit log** − The commit log is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.

- **Mem-table** − A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.

- **SSTable** − It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.

- **Bloom filter** − These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

Cassandra Query Language

Users can access Cassandra through its nodes using Cassandra Query Language (CQL). CQL treats the database **(Keyspace)** as a container of tables. Programmers use **cqlsh:** a prompt to work with CQL or separate application language drivers.

Clients approach any of the nodes for their read-write operations. That node (coordinator) plays a proxy between the client and the nodes holding the data.

Write Operations

Every write activity of nodes is captured by the **commit logs** written in the nodes. Later the data will be captured and stored in the **mem-table.** Whenever the mem-table is full, data will be written into the **SStable** data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates the SSTables, discarding unnecessary data.

Read Operations

During read operations, Cassandra gets values from the mem-table and checks the bloom filter to find the appropriate SSTable that holds the required data.

CASSANDRA-DATA MODEL

Cassandra database is distributed over several machines that operate together. The outermost container is known as the Cluster. For failure handling, every node contains a replica, and in case of a failure, the replica takes charge. Cassandra arranges the nodes in a cluster, in a ring format, and assigns data to them.

Keyspace

Keyspace is the outermost container for data in Cassandra. The basic attributes of a Keyspace in Cassandra are −

- **Replication factor** − It is the number of machines in the cluster that will receive copies of the same data.

- **Replica placement strategy** − It is nothing but the strategy to place replicas in the ring. We have strategies such as **simple strategy** (rack-aware strategy), **old network topology strategy** (rack-aware strategy), and **network topology strategy** (datacenter-shared strategy).

- **Column families** − Keyspace is a container for a list of one or more column families. A column family, in turn, is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. Each keyspace has at least one and often many column families.

The syntax of creating a Keyspace is as follows −

CREATE KEYSPACE Keyspace name

WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

The following illustration shows a schematic view of a Keyspace.
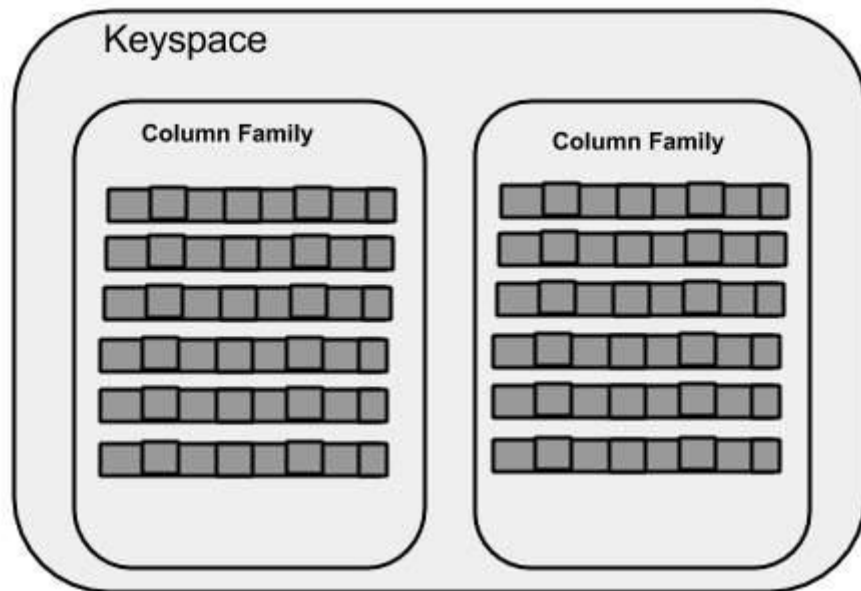
Column Family

A column family is a container for an ordered collection of rows. Each row, in turn, is an ordered collection of columns. The following table lists the points that differentiate a column family from a table of relational databases.

Relational Table	Cassandra column Family
A schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value.	In Cassandra, although the column families are defined, the columns are not. You can freely add any column to any column family at any time.
Relational tables define only columns and the user fills in the table with values.	In Cassandra, a table contains columns, or can be defined as a super column family.

A Cassandra column family has the following attributes −

- **keys_cached** − It represents the number of locations to keep cached per SSTable.

- **rows_cached** − It represents the number of rows whose entire contents will be cached in memory.

- **preload_row_cache** − It specifies whether you want to pre-populate the row cache.

Note − Unlike relational tables where a column family's schema is not fixed, Cassandra does not force individual rows to have all the columns.

The following figure shows an example of a Cassandra column family.
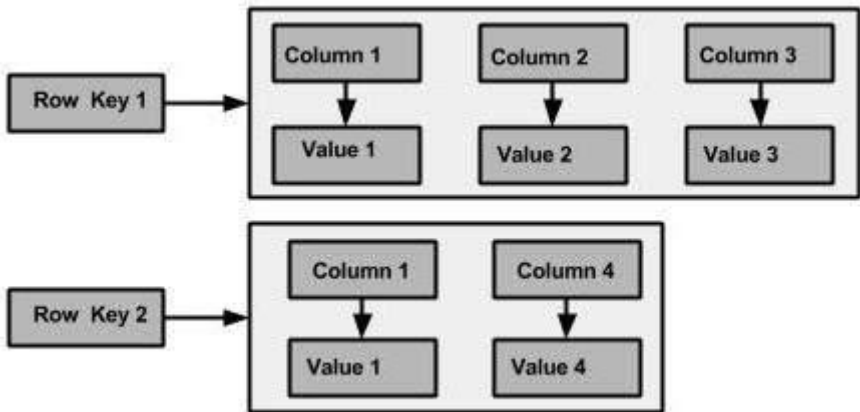
Column

A column is the basic data structure of Cassandra with three values, namely key or column name, value, and a time stamp. Given below is the structure of a column.

SuperColumn

A super column is a special column; therefore, it is also a key-value pair. But a super column stores a map of sub-columns.

Generally, column families are stored on disk in individual files. Therefore, to optimize performance, it is important to keep columns that you are likely to query together in the same column family, and a super column can be helpful here. Given below is the structure of a super column.

2. <u>Study the procedure to execute a query using Apache Cassandra.</u>

DataStax Community Edition must be installed on system before installing Cassandra. Verify the Cassandra installation using the following command:

- $ Cassandra version

- If Cassandra is already installed on system, then you will get the following response:

- Connected to Test Cluster at 127.0.0.1:9042.
- [cqlsh 5.0.1 | Cassandra 2.2.4 | CQL spec 3.3.1 | Native protocol v4] Use HELP for help.

Step 1 – start CQL shell (cqlsh)

cqlsh is a Python-based command-line client for Cassandra. It allows you to run CQL queries directly onto Cassandra database.

On Unix

On a Unix-flavored machine, you have virtually nothing to do in order to start cqlsh.

CASSANDRA_HOME/bin>./cqlsh

By default, Cassandra is started on localhost at port 9160. You can provide different values on the command line if you wish, similar to the following command:

CASSANDRA_HOME/bin>./cqlsh <host> <port>

On Windows

Starting cqlsh on Windows requires installing Python, and then running cqlsh as an option with it.

Install and configure Python

Python can be installed and configured on Windows in three easy steps:

1. Install Python from http://www.python.org/download/releases/2.7.5/ (Choose Windows x86 MSI Installer option) and set python.exe into your path environment variable. Latest version of python may not work correctly due to incompatibility.
2. Install Python thrift module from https://pypi.python.org/pypi/thrift/0.9.0 and run from the install location given as follows:
3. **thrift-0.9.0\python setup.py install**
4. Install the Python CQL module using the following command:
5. **CASSANDRA_HOME/pylib/python setup.py install**

Run cqlsh

Goto bin folder under CASSANDRA_HOME, and run Python with cqlsh as an option using the following command:

CASSANDRA_HOME\bin>python cqlsh

The following query is executed to create a database named userdb:

cqlsh> userdb;

Or

cqlsh> CREATE SCHEMA userdb;

Step 2 – create and use a keyspace

Keyspace in Cassandra is the equivalent of database/schema in relational databases. While creating a keyspace, you need to specify replication settings:

cqlsh> CREATE KEYSPACE testkeyspace WITH replication = {'class':'SimpleStrategy','replication_factor':1};

We'll be using SimpleStrategy to keep things simple, because our Cassandra setup is just single node. In production environment, where it's usually common to have

multiple data centers, NetworkTopologyStrategy is generally used because it distributes data across data centers better.

Replication factor = 1 means there will be single copy of a row on a particular node. Higher replication factors are set up in real systems for creating multiple replicas that ensures data availability in case of disk failures.

To be able to work with tables, you need to use your keyspace, as shown in the following statement:

cqlsh> USE testkeyspace;

Another option is to prefix the table name with the keyspace name in all queries.

Step 3 – describe and list keyspace

At any time, you can DESCRIBE the keyspace, use the following command to do that:

cqlsh> DESCRIBE KEYSPACE testkeyspace;

If you wish to list all keyspaces present in the database, a Cassandra reserve keyspace named **system** comes in handy. It contains many system-defined tables for database objects definition and cluster configuration. Let's list all records from the schema_keyspaces table that contains records for each keyspace, using the following command:

cqlsh> SELECT * FROM system.schema_keyspaces;

The output of this command looks as follows:

Step 4 – create table and insert records

Run the following query on cqlsh console for creating a table called users. For those who are acquainted with SQL, the following syntax should look very familiar and almost identical.

CREATE TABLE users(user_id varchar, age int, email varchar, city varchar,PRIMARY KEY (user_id));

While creating a table, you provide names of each column with their data type.

In Cassandra, an INSERT operation is actually an "Upsert" (UPDATE or INSERT), which means columns are updated in case a row for the given primary key already exists, otherwise all columns are freshly inserted.

INSERT INTO users(user_id, age, email, city)VALUES ('jsmith',32,'john.smith@example.com','Dallas');

INSERT INTO users(user_id, city) VALUES ('pscott','New Jersey');

INSERT INTO users(user_id, age, email) VALUES ('davidb',37,'david.bergin@example.com');

You can view a table metadata using the DESCRIBE command, as shown in the following statement:

cqlsh> DESCRIBE TABLE users;

Step 5 – display records

SELECT command lists content of a table; a filter can be applied using the WHERE clause:

cqlsh>SELECT * FROM users;

cqlsh:testkeyspace> SELECT * FROM users WHERE user_id='jsmith';

```
amresh@ubuntu: ~/apache-cassandra-1.2.5/bin
cqlsh:testkeyspace> SELECT * FROM users;

 user_id | age  | city        | email
---------+------+-------------+--------------------------
  jsmith |   32 |      Dallas |   john.smith@example.com
  pscott | null | New Jersey  |                     null
  davidb |   37 |        null | david.bergin@example.com

cqlsh:testkeyspace> SELECT * FROM users WHERE user_id='jsmith';

 user_id | age | city   | email
---------+-----+--------+--------------------------
  jsmith |  32 | Dallas | john.smith@example.com

cqlsh:testkeyspace>
```

How is data actually stored?

Result of previous SELECT queries may make you wonder—Oh! This looks so similar to RDBMS tables, is this how data is stored in Cassandra as well?

The answer to the preceding questions is—although CQL gives you a nice looking interface similar to a RDBMS you are used to, data is stored in Cassandra according to its own data-model.

A graphical representation of data stored in Cassandra as a result of preceding INSERT queries is shown in the following diagram:
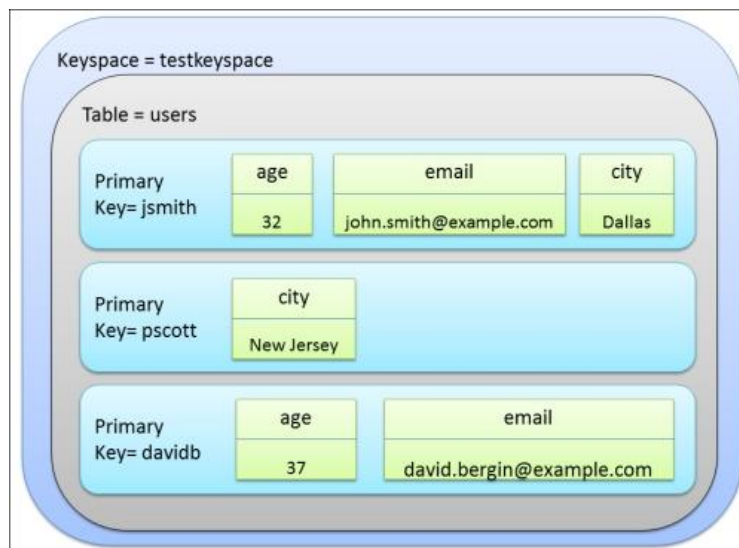
In Cassandra, a Keyspace is container for your application data, and can be compared with schema/database in RDBMS. In keyspaces, you would have Tables (formerly known as column families). Each table has multiple rows identified by a Primary Key (earlier called as row key).

Cassandra is a schema-optional database. While a CREATE TABLE statement appears to suggest that a table has fixed number of columns in each row, they actually may have an arbitrary number of columns, as is shown above.

Unlike RDBMS, Cassandra tables are usually denormalized by data modellers to avoid runtime joins. This introduces redundancies in data, but gives better performance.

Primary keys may have multiple components. Such keys are called *Composite Keys,* and are stored in clusters differently. Composite keys are discussed later in the book.

Step 6 – deleting data

Data can be deleted using CQL in one of the following ways:

- DELETE: It deletes column(s) or entire row(s)
- TRUNCATE: It deletes all rows from the table
- USING TTL: It sets time to live on column(s) within a row; after expiration of specified period, columns are automatically deleted by Cassandra.

DELETE

The DELETE command is used to delete columns(s) or row(s).

The following command deletes city and email columns for the primary key davidb:

DELETE email FROM users WHERE user_id = 'davidb';

Don't specify any column after the DELETE keyword if you want to delete the entire row:

DELETE FROM users WHERE user_id = 'pscott';

Below is the output of above two deletions:

SELECT * FROM users;

```
amresh@ubuntu: ~/apache-cassandra-1.2.5/bin
cqlsh:testkeyspace> DELETE email FROM users WHERE user_id = 'davidb';
cqlsh:testkeyspace> DELETE FROM users WHERE user_id = 'pscott';
cqlsh:testkeyspace> SELECT * FROM users;

 user_id | age | city   | email
---------+-----+--------+-------------------------
  jsmith |  32 | Dallas | john.smith@example.com
  davidb |  37 |   null |                    null

cqlsh:testkeyspace>
```

TRUNCATE

If you ever need to remove all records from a table, CQL has the following shorthand:

TRUNCATE <Table name>;

Try the following and see yourself; none of the records previously inserted should be returned:

TRUNCATE users;

SELECT * FROM users;

USING TTL

Another way to delete data in Cassandra is to specify an expiration time named TTL (time to live) while inserting data. We'll explore this feature in the later section—*Top 3 features you need to know about*. It's too early to get into those things, right?

Dropping TABLE and KEYSPACE

Tables and keyspaces can be dropped by running:

DROP TABLE users;

DROP KEYSPACE testkeyspace;

The DROP operation is complete, immediate, and recursive in nature. All data they hold is completely removed with immediate effect.

3.Execute minimum 10 queries using Cassandra column-based database
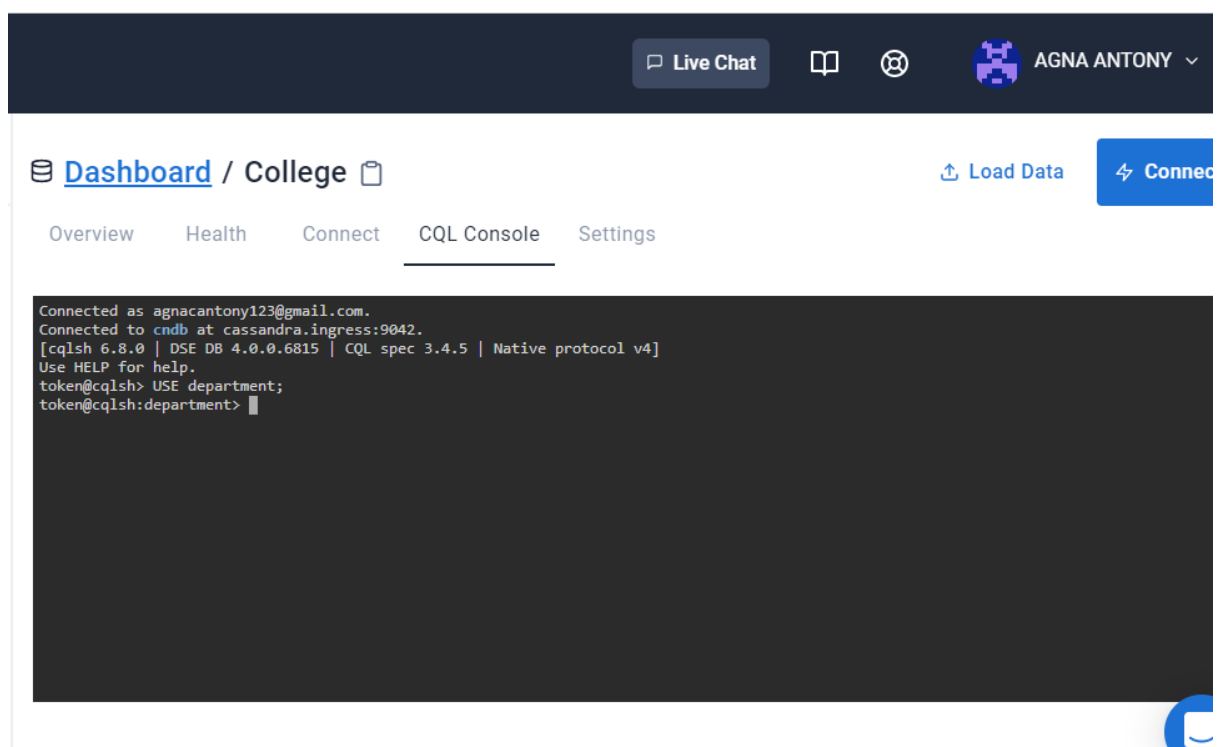
INITIAL PROCEDURE:

Step 1: Here we consider the database "College"
Step 2: Creating and updating a key space , here keyspace is "department"
Step 3: Creating a table "Department"
Step 4: Describe table "Department"

QUERIES

1. INSERTING DATA INTO A TABLE

2. RETRIEVAL OF ALL DATA

SELECT * FROM department.Department;

3. RETRIEVE THE DETAILS OF STAFFS USING CONDITIONS

4. ALTER TABLE command

```
(1 rows)
token@cqlsh:department> ALTER TABLE Department ADD placement text;
token@cqlsh:department> ALTER TABLE Department ADD salary int;
token@cqlsh:department> SELECT * FROM department.Department;

 dept_id | dept_name | faculty | placement | salary | students
---------+-----------+---------+-----------+--------+----------
     456 |     civil |      20 |      null |   null |      430
     678 |  chemical |      15 |      null |   null |      350
     345 |     mtech |      12 |      null |   null |      560
     180 |       mca |       6 |      null |   null |      200

(4 rows)
token@cqlsh:department>
```

🖘 **Dashboard** / College 🗋   ⬆ Load Data   ⚡ Connect

Overview   Health   Connect   CQL Console   Settings

```
 dept_id | dept_name | faculty | placement | salary | students
---------+-----------+---------+-----------+--------+----------
     456 |     civil |      20 |      null |   null |      430
     678 |  chemical |      15 |      null |   null |      350
     345 |     mtech |      12 |      null |   null |      560
     180 |       mca |       6 |      null |   null |      200

(4 rows)
token@cqlsh:department> ALTER TABLE Department DROP salary;
token@cqlsh:department> SELECT * FROM department.Department;

 dept_id | dept_name | faculty | placement | students
---------+-----------+---------+-----------+----------
     456 |     civil |      20 |      null |      430
     678 |  chemical |      15 |      null |      350
     345 |     mtech |      12 |      null |      560
     180 |       mca |       6 |      null |      200

(4 rows)
token@cqlsh:department>
```

5. RETRIEVING TIMESTAMPS

Timestamps

Each time you write data into Cassandra, a timestamp is generated for each column value that is updated. Internally, Cassandra uses these timestamps for resolving any conflicting changes that are made to the same value.

```
(4 rows)
token@cqlsh:department> SELECT dept_name,students,writetime(faculty) FROM Department;

 dept_name | students | writetime(faculty)
-----------+----------+--------------------
      civil |      430 |     1630515962753998
   chemical |      350 |     1630516087797655
      mtech |      560 |     1630515865896038
        mca |      200 |     1630516029775638

(4 rows)
token@cqlsh:department>
```

6. RETRIEVE THE TTL VALUE FOR ' chemical', dept_name'

```
ING
token@cqlsh:department> SELECT dept_name,students,TTL(students) FROM Department WHERE dept_name='chemical' ALLOW FILTERING;

 dept_name | students | ttl(students)
-----------+----------+---------------
   chemical |      350 |          null

(1 rows)
token@cqlsh:department>
```

7. QUERY TO ADD THE IDENTIFIER USING A UUID

```
(1 rows)
token@cqlsh:department> ALTER TABLE Department ADD id uuid;
token@cqlsh:department> SELECT * FROM department.Department;

 dept_id | dept_name | faculty | id   | placement | students
---------+-----------+---------+------+-----------+----------
     456 |      civil |      20 | null |      null |      430
     678 |   chemical |      15 | null |      null |      350
     345 |      mtech |      12 | null |      null |      560
     180 |        mca |       6 | null |      null |      200

(4 rows)
token@cqlsh:department>
```

8. QUERY TO INSERT AN ID FOR MARY USING UUID() FUNCTION AND THEN VIEW THE RESULTS

uuid

A universally unique identifier (UUID) is a 128-bit value in which the bits conform to one of several types, of which the most commonly used are known as Type 1 and Type 4. The CQL uuid type is a Type 4 UUID, which is based entirely on random numbers. UUIDs are typically represented as dash-separated sequences of hex digits.

```
(4 rows)
token@cqlsh:department> UPDATE Department SET id=uuid() WHERE dept_id=180;
token@cqlsh:department> UPDATE Department SET id=uuid() WHERE dept_id=678;
token@cqlsh:department> SELECT dept_id, id FROM Department WHERE dept_id=180;

 dept_id | id
---------+--------------------------------------
     180 | f9608cf2-77d8-4979-a5fa-242f4bbddd26

(1 rows)
token@cqlsh:department> SELECT dept_id, id FROM Department WHERE dept_id=678;

 dept_id | id
---------+--------------------------------------
     678 | 31dbbfd1-3d43-4c63-b9d5-5fa373b690d8

(1 rows)
token@cqlsh:department>
```

9. QUERY TO MODIFY OUR TABLE TO ADD A SET OF STUDENTS

set

The set data type stores a collection of elements. The elements are unordered, but cqlsh returns the elements in sorted order. For example, text values are returned in alphabetical order. One advantage of using set is the ability to insert additional items without having to read the contents first.

```
token@cqlsh:department> UPDATE Department SET students=569 WHERE dept_id=678;
token@cqlsh:department> SELECT * FROM department.Department;

 dept_id | dept_mail | dept_name | faculty | id                                   | placement | students
---------+-----------+-----------+---------+--------------------------------------+-----------+----------
     456 |      null |     civil |      20 |                                 null |      null |      430
     678 |      null |  chemical |      15 | 31dbbfd1-3d43-4c63-b9d5-5fa373b690d8 |      null |      569
     345 |      null |     mtech |      12 |                                 null |      null |      560
     180 |      null |       mca |       6 | f9608cf2-77d8-4979-a5fa-242f4bbddd26 |      null |      200

(4 rows)
token@cqlsh:department>
```

10. QYERY TO DELETE DATA USING CONDITION

```
token@cqlsh:department>  DELETE students FROM Department WHERE dept_id=180;
token@cqlsh:department> SELECT * FROM department.Department;

 dept_id | dept_mail | dept_name | faculty | id                                   | placement | students
---------+-----------+-----------+---------+--------------------------------------+-----------+----------
     456 |      null |     civil |      20 |                                 null |      null |      430
     678 |      null |  chemical |      15 | 31dbbfd1-3d43-4c63-b9d5-5fa373b690d8 |      null |      569
     345 |      null |     mtech |      12 |                                 null |      null |      560
     180 |      null |       mca |       6 | f9608cf2-77d8-4979-a5fa-242f4bbddd26 |      null |     null

(4 rows)
token@cqlsh:department>
```

11. TRUNCATE TABLE FROM THE DATABASE

```
(4 rows)
token@cqlsh:department> TRUNCATE Department;
token@cqlsh:department> SELECT * FROM Department;

 dept_id | dept_mail | dept_name | faculty | id | placement | students
---------+-----------+-----------+---------+----+-----------+---------

(0 rows)
token@cqlsh:department>
```