4ME307 Assignment 2: Data Management Approaches and Techniques

Rema Salman
School of Computer Science, Physics and Mathematics, Linnaeus University, Sweden
rs223bc@student.lnu.se

The purpose of this assignment is to get hands-on experience on data management approaches and techniques for developing large-scale Web applications, in particular using Cassandra. The assignment demonstrates my knowledge of data-model creation, using CQL in Cassandra; 2) data-handling using the CQL; and 3) practical implementation (a Maven/Java program) for record modification. Also, I briefly discuss the consistency of the data in large-scale distributed Web architecture. The report is structured based on the provided tasks in Assignment 2.

TASK-1: CASSANDRA INSTALLATION

In order to install Cassandra apache on my Mac, I needed to check the Java version by the command:

```
java -version
```

Because I have a higher version of java in comparison with the required JDK version-specification, I installed java 8 using Homebrew package manager via the command:

```
brew install --cask adoptopenjdk8
```

Then, I downloaded the latest Apache Cassandra 4.0 beta from the official website. After installing it, I navigated to the folder and created data directories, as provided in the the

To launch Cassandra and verify that it is running, I used the commands sudo `./cassandra -R` and `./cqlsh` from **Cassandra's_HOME directory/bin** to get the output, as visualized in Figure 1.

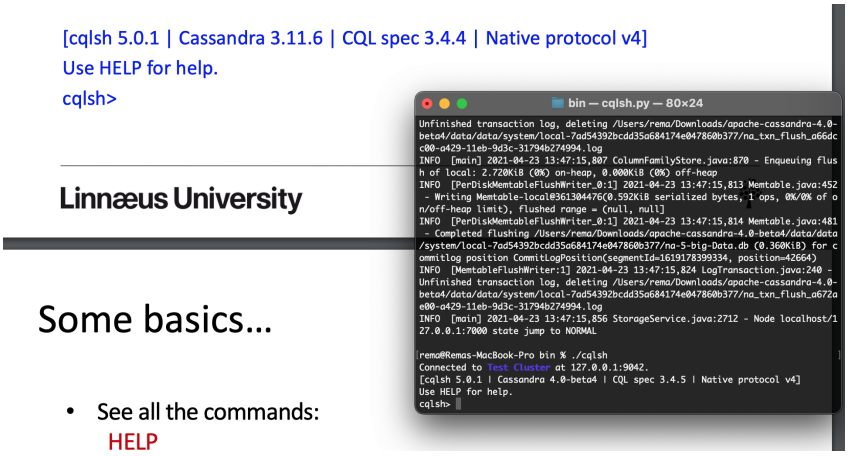

Figure 1. Screenshot of Cassandra Verification & Running on my Mechanic.

TASK-2: DATA MODEL CREATION (KEYSPACE & TABLES)

In order to create the data model the following command was executed in Cassandra using CQL.

```
CREATE KEYSPACE IF NOT EXISTS salmandatamodel WITH
REPLICATION = {
       'class' : 'NetworkTopologyStrategy',
       'datacenter1' : 3
       } AND DURABLE_WRITES = false;
```

`USE salmandatamodel` was executed for accessing and using the created keyspace. After that, two tables/entities were created to store their corresponding information. I used the data type "***Integer"*** for the employee_id instead of UUID, sinc the DB will not be imported to another DB and hence it will not result in any conflicts in the uniqueness of the primary key. In the case of following the best practices, I would have the primary key auto incremented if the population of the data was automated.

```
CREATE TABLE employee (
   employee_id int PRIMARY KEY,
   employee_name text,
   department_name text,
   position text,
   company_name text);
```

```
CREATE TABLE salary (
   employee_id int PRIMARY KEY,
   base_salary int,
   medical_allowance int,
   travel_allowance int,
   total_salary int);
```

TASK-3: DATA POPULATION & MANIPULATION

In order to populate information in the entities, 4 records were created for each table by executing the following INSERT commands in Cassandra using the CQL.

Inserting information into the employee table	***Inserting information into the salary table***
`INSERT INTO employee (employee_id, employee_name, department_name, position, company_name) VALUES (0, 'rema salman', 'Sales', 'HR', 'IKEA');`	`INSERT INTO salary (employee_id, base_salary, medical_allowance, travel_allowance, total_salary) VALUES (0, 4000, 4000, 4000, 4000);`
`INSERT INTO employee (employee_id, employee_name, department_name, position, company_name) VALUES (1, 'henry', 'Sales', 'Manager', 'IKEA');`	`INSERT INTO salary (employee_id, base_salary, medical_allowance, travel_allowance, total_salary) VALUES (1, 2000, 2000, 2000, 2000);`
`INSERT INTO employee (employee_id, employee_name, department_name, position,company_name) VALUES (2, 'henry', 'Design', 'designer', 'gif');`	`INSERT INTO salary (employee_id, base_salary, medical_allowance, travel_allowance, total_salary) VALUES (2, 5000, 5000, 5000, 5000);`
`INSERT INTO employee (employee_id, employee_name, department_name, position, company_name)VALUES (3, 'henry', 'Design', 'content creator', 'gif');`	`INSERT INTO salary (employee_id, base_salary, medical_allowance, travel_allowance, total_salary) VALUES (3, 4000, 4000, 4000, 4000);`

The commands for updating the data in both entities were as follows:

Updating employee name in the employee table	***Updating medical allowance in the salary table***
`UPDATE employee SET employee_name ='tylor' WHERE employee_id = 1;`	`UPDATE salary SET medical_allowance ='3000' WHERE employee_id = 1;`

The commands for deleting a value of a column and the entire row from the tables:

Deleting employee name from the employee table and the row where the employee ID is 1	*Deleting medical allowance from the salary table and the row where the employee ID is 1*
`DELETE employee_name FROM employee WHERE employee_id = 1;`	`DELETE medical_allowance FROM salary WHERE employee_id = 1;`
`DELETE FROM employee WHERE employee_id = 1;`	`DELETE FROM salary WHERE employee_id = 1;`

TASK-4: USERS, ROLES, & PERMISSION

As described in the official Cassandra documentation[1], when the option LOGIN is true, `CREATE USER` is equivalent to `CREATE ROLE`. Therefore, I used the role to create the three db user the following queries were executed:

```
CREATE ROLE superuser1 WITH PASSWORD = 'password_a' AND LOGIN = true AND SUPERUSER = true;
```

```
CREATE ROLE modifyuser1 WITH PASSWORD = 'password_b' AND LOGIN = true;
```

```
CREATE ROLE readuser1 WITH PASSWORD = 'password_c' AND LOGIN = true;
```

Superuser1 was granted the permission for executing all queries (INSERT, UPDATE, DELETE, and TRUNCATE) on any entity and row in the created keyspace, which was done by executing the command:

```
GRANT MODIFY ON KEYSPACE salmandatamodel TO superuser1;
```

Modifyuser1 was granted the permission for modifying the employee information only, through the command:

```
GRANT MODIFY ON salmandatamodel.employee TO modifyuser1;
```

Readuser1 was granted the permission only to read (SELECT) the information from the tables (employee and salary), which was done via the command:

```
GRANT SELECT ON KEYSPACE salmandatamodel TO readuser1;
```

TASK-5: PRACTICAL IMPLEMENTATION

In order to complete this task, I implemented a small program in Python. It creates a cluster object and a connection session to the created KEYSPACE (`salmandatamodel`) from the CQL. The Main Script executes the ***initial*** function, prints the records (prior updates), evokes ***updateRecords*** function, and prints the modified records. The functions of the program are commented to explain my provided solution, see ***app.py*** in the provided ZIP file.

TASK-6: DATA CONSISTENCY DISCUSSION

Discussing consistency in Apache Cassandra, usually split into discussions about replication and consistency since the latter is a way to ensure if the data replication is successfully done. However, the discussion will focus on consistency in this task. Cassandra fits in the AP intersection (meaning it provides Availability and Partition-Tolerance) when it comes to the CAP theorem in distributed systems. Partition-Tolerance refers to cluster continuation even if partition existence (nodes do not communicate) and Availability means the ability to access clusters even in case of node failure. Meanwhile, Consistency is the harder part in a distributed system.

[1] cassandra.apache.org/documentation

Even though Cassandra does not optimize for consistency, it provides flexibility by allowing tunable data consistency. This will allow tuning away from AP depending on how much consistency is needed for the data (strong or evental) but still maintaining the availability and partition. Tuning consistency works on a pre-operation basis and for both writes and reads. Also, Cassandra allows distributed operation across multi-data centers, while the strategies of reads and writes are summarized in Table 1.

Table 1. Read and Write strategies in Cassandra

Strategy	***For Writes***	***For Reads***	***Explanation of the Strategy***
Any	X	-	The write is successful on any available nodes.
One	X		The **write/read** is successful on/from any node, which is responsible for that row - it can be a primary or a replica node.
Quorum	X		The **write** is successful on replica nodes' quorum- it is determined by `(replication_factor/2)+1`. The **read** result is returned from a quorum of servers with the recent timestamp for the data.
Local_Quorum	X		The **write** is successful on replica nodes' quorum in the same data center as the *coordinator node* **.** The **read** result is returned from a quorum of servers with the recent timestamp for the data coming from the same center *(coordinator node).*
Each_Quorum	X		The **write** is successful on replica nodes' quorum in all data centers. The **read** result is returned from a quorum of servers with the most recent timestamp in all data centers.
All	X		The **write/read** is successful on/from all replica nodes for a row key.

There is two methods for configuring a consistency strategy, the first option is using the following cqlsh command: `USING CONSISTENCY [type of strategy]`**.** The other option would be using an appropriate driver for configuring the consistency level is via the program, e.g. *"call QueryBuilder.insertInto with a setConsistencyLevel argument using the Java driver"*[2].

It is also worth to mention that Cassandra offers a hinted handoff methodology for repairing any failures during the write operations. There, the consistency level instructs the coordinator to check the number of replicas and, in case of node unavailability (failure), a hint with the data row is stored until it is available to be updated according to the latest data. In the case of the read operations, Cassandra ensures data frequent-read stays consistent by the data comparisons from the replicas by the coordinator node (no updated copies). It updates the inconsistent replica so the next time a read occurs that data would be up-to-date.

[2] How is the consistency level configured? | Apache Cassandra 3.x