Lab 9 – Cassandra

CC5212-1 – May 26, 2021

Today we will play around with a NoSQL store. In particular, we will play around with Apache Cassandra, which is a column family/tabular store. We will have a look at adding and removing keyspaces, adding and removing tables, managing schema and types, how data are inserted and queried, etc. We are going to use the Cassandra Query Language (CQL) interface, which is a bit like a lightweight version of SQL. You may find this page helpful: `http://docs.datastax.com/en/cql/3.1/cql/cql_reference/cqlCommandsTOC.html`.

To start with, log into the cluster as in previous labs..

- You can use the following commands to get some info on the Cassandra nodes (each a machine):[1]

```
nodetool status
nodetool -h 10.10.10.1 info
nodetool -h 10.10.10.12 info
nodetool -h 10.10.10.14 info
nodetool -h 10.10.10.15 info
```

- Next we will launch the CQL (Cassandra Query Language) client. Run `cqlsh 10.10.10.1`.[2]
- **Note that you will be submitting the (working) commands you enter for the following parts so please copy and paste them into a text file as you proceed.**
- Now we first create a keyspace and some tables.
  - First we will create a keyspace with a simple replication strategy (hashing on key) and 3 copies:

```
CREATE keyspace YOURUSERNAME with replication =
   { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
```

    We can see the list of keyspaces:

    ```
    DESCRIBE keyspaces;
    ```

  - Next we want to move to that keyspace: `USE YOURUSERNAME;`
  - Now we can *try* create a table in that keyspace with two columns, one called username of type text, one called age of type int:

    ```
    CREATE TABLE UNIQUETABLENAME ( username text, age int);
    ```

    But wait, that didn't work! Cassandra tables are really maps and every map needs keys. We need to specify a primary key, which gives a unique identifier for each row:

    ```
    CREATE TABLE UNIQUETABLENAME ( username text PRIMARY KEY, age int);
    ```

    That's better. Note that if you are not in a given namespace, you can still reference a table using `namespace.table`. If you are inside the right namespace, you can just use the table name.
  - Now we can list the tables in the keyspace we are in:

    ```
    DESCRIBE tables;
    ```

    On second thought, let's add an awake column to the table:

    ```
    ALTER TABLE UNIQUETABLENAME ADD awake boolean;
    ```

[1] Machine `10.10.10.13` died the night of a project deadline †

[2] If there is a problem "`Connection error: ...`", try to run the following first: `nodetool enablethrift`

- Okay, now you have a taste for how to create keyspaces and tables. After all your hard work above, you're going to delete everything you just made. Please only drop your stuff! We will have more interesting data if we work from now on in one common keyspace/table.

  ```
  DROP TABLE UNIQUETABLENAME;
  DROP KEYSPACE YOURUSERNAME;
  ```

- Let's move to a common keyspace I made earlier where we can see how to update and query data.
  - First load the keyspace: `USE cc5212;`
  - Next, see what tables there are: `DESCRIBE tables;`
  - Next, see schema and details of table: `DESCRIBE TABLE students;`
  - What's inside the table? `SELECT * FROM students;`

- Now you will add your own data. In the following, I use my own data as an example. *Please replace my details with yours.* Please don't overwrite my row; it is very special to me.
  - First we will insert a name with initials:

    ```
    INSERT INTO students( username, name ) VALUES ( 'ahogan', 'A. Hogan' );
    ```

    Query the table again to make sure it's in there: `SELECT * FROM students;`
  - What happens if we try to insert a bad type:

    ```
    INSERT INTO students( username, name ) VALUES ( 'ahogan', true );
    ```

    (Doesn't work! Type of `name` is text, not boolean.)
  - Let's just delete the name

    ```
    DELETE name FROM students WHERE username = 'ahogan';
    ```
  - Actually let's delete the entire row

    ```
    DELETE FROM students WHERE username = 'ahogan';
    ```
  - Now let's try give our full name instead:

    ```
    INSERT INTO students( username, name ) VALUES ( 'ahogan', 'Aidan Hogan' );
    ```

    If you query the table again, you will see that the old value is overwritten. Unlike BigTable/Dynamo, etc., values are not versioned in Cassandra.
  - Let's try add more information, like our age:

    ```
    INSERT INTO students( age ) VALUES ( 33 );
    ```

    Doesn't work? Why not? (Whose age should we update? We need to give the key!)

    ```
    INSERT INTO students( username, age ) VALUES ( 'ahogan', 33 );
    ```

    Try query the table again. We see that age is added alongside name.
  - Fill in the rest of the details in the table for yourself following the above idea. Note that the syntax for expressing a set value is, e.g., `{'string1','string2'}`.

- Okay, so let's try some queries over the table now; hopefully it's filling up.
  - Let's get the number of folks in the table: `SELECT COUNT(*) AS scount FROM students;`
  - The name of a given user: `SELECT name FROM students WHERE username = 'ahogan';`
  - People who are a given age: `SELECT name FROM students WHERE age = 33;`

    Oops, that last one didn't work.

- Actually we are very limited in what we can do by default in Cassandra. Like a map, we can only lookup by keys: in this case, the primary key is `username`. To do more, we need to create indexes. But we can only create an index on one column.
  - As a first step, you should make a personal copy of the table by saving it to a file and reloading it (all one command here):

    ```
    COPY students (username, name, age, group, hobbies, awake, message)
    TO 'MYNAME .csv';
    ```

  - *Create a new table with your username (**using lowercase characters**) that fits the data*[3],
  - ... and then load the data from the file into your table MYUSERNAME:

    ```
    COPY MYUSERNAME (username, name, age, group, hobbies, awake, message)
    FROM 'MYNAME .csv';
    ```

    This is your own copy of the table to play with.
  - Let's say we're building an application that needs to lookup users with a given age. In Cassandra, we need to build an index for that.

    ```
    CREATE INDEX ageIndexMYUSERNAME ON MYUSERNAME (age);
    ```

  - Now we can try find users by age:

    ```
    SELECT name FROM MYUSERNAME WHERE age = 33;
    ```

    Okay great, now it works![4]
  - Now over your copy of the data, try to figure out how to get the following queries to work (check out the documentation linked at the top). Note that you can change the values in the queries (`profe`, `20`, `false`) as you wish to ensure that the query generates answers. *Hint*: For the second query, you may need to ignore warnings!
    * `SELECT name FROM MYUSERNAME WHERE group = 'profe';`
    * `SELECT name FROM MYUSERNAME WHERE age > 24 AND awake = true;`

- Now we get into primary keys in Cassandra, which is another important place where Cassandra differs from SQL engines. This can be confusing but there's additional discussion and examples here in case you don't follow this document: `http://goo.gl/QGaTv2`.
  - The only general way to query on multiple columns (without warnings) in Cassandra is to create a compound primary key on those columns. The most complex type of key in Cassandra looks like `PRIMARY KEY ((pk1, ..., pkn), ck1, ..., ckn)`. Like in a relational database, the entire key should only have one tuple for a given value of `pk1, ..., pkn, ck1, ..., ckn`. Unlike in a relational database, Cassandra is designed to be distributed. Here, the value of `(pk1, ..., pkn)` (partition key) will be used to partition the tuple: to decide what machine it goes on (e.g., by hashing the value of those columns for the tuple). Thereafter, on each machine, `ck1, ..., ckn` (clustering key) will be used to apply a sorting order on the data ordered by `ck1`, then `ck2`, etc. A partition key is obligatory but as we saw before, the clustering key is not. Note that if you specify a key without nested parentheses, like `PRIMARY KEY (pk, ck1, ..., ckn)`, by default, the first column will be a partition key and the rest will form the clustering keys!
  - For a key `PRIMARY KEY ((pk1, ..., pkn), ck1, ..., ckn)`, a query must give values for all partition key columns `pk1, ..., pkn`. It can optionally specify clustering values so long as the order is followed. So you can query on `(pk1, ..., pkn)` or `(pk1, ..., pkn, ck1)` or `(pk1, ..., pkn, ck1, ck2)`, etc., but not `pk1` or `(pk1, ..., pkn, ck2)`, etc.
  - Create a table with a custom primary key that can answer the following query without warning:

[3] The syntax for the set is `set`⟨*text*⟩

[4] Umm, if it doesn't, drop the index (`DROP INDEX` INDEXNAME) and recreate it. :(

    * `SELECT age FROM TABLENAME WHERE group = 'profe' AND username='ahogan';`[5]
  - Create one table with a custom primary key that can answer the following queries without warning. Again, feel free to change the particular values.
    * `SELECT name FROM TABLE2NAME WHERE group='profe';`
    * `SELECT name FROM TABLE2NAME WHERE group='profe' AND age>20;`
    * `SELECT name FROM TABLE2NAME WHERE group='profe' AND username='ahogan' AND age=33;`

- What about other queries like:
  - `SELECT group (COUNT(username) AS groupsize) FROM students GROUP BY group;`

  Cassandra does not allow such queries. However, the idea is that we should use some external service to create the table for us. We could use Hadoop/Pig/Spark to load the `students` table direct from Cassandra and then process the data to perform the counts per group, writing the results back to Cassandra as a new table with group as key and the count as value. In fact this is the idea of Cassandra: it offers very limited expressivity of queries that it can support efficiently. If you want to do more complex queries, you need to do process them externally and store the results as a tabular map that you can then query by a key. (Again, we should imagine a use-case like a web-page where the same queries are re-used over and over.)
- In U-Cursos, you will find a simple example called `test.pig` that reads the `testin` table and filters by value, writing the results to a table called `testout` in Cassandra. You should adapt this script to again read the `students` table, but this time to create a table in Cassandra with the group names and their size. Note that there are some complications: (1) to write to a Cassandra table, the Pig script must define the key structure of the Cassandra table; (2) Cassandra needs to be told which columns of the Pig relation correspond to which columns of the Cassandra table as part of a URL request. These aspects are covered in the `test.pig` script. Note that you may also want to filter empty strings from the group key before applying `GROUP BY` in Pig.
  - In Cassandra, create a table:
    * `CREATE TABLE TABLE3NAME ( group text, size bigint, PRIMARY KEY(group) );`
  - Modify the Pig script `test.pig` to generate the counts of members per group from the same `students` table and write the results back to TABLE3NAME. Run the script.
  - Check the results in Cassandra.

- Submit all the working CQL commands you typed in the lab as a text file along with the modified Pig script you created.
- Optional: For didactic/practical reasons we used small example data so we could quickly dump and re-index files to play with Cassandra's keys and indexing schemes. However, you can also think about adapting previous Pig scripts for IMDb to read and write from Cassandra.
- Questions: What sorts of use-cases does Cassandra target? When would it work well versus an SQL database and when not? Why does it not allow queries it might (not) be able to answer efficiently?

[5] This would be useful if a student could be in multiple groups.