Assignment 8

Cassandra Clusters

We will have **two** Cassandra clusters for this part of the course. The distinction is important.

One cluster will three small nodes will be used to explore how Cassandra handles failures. It can be contacted at `unreliable1.local`. One of the nodes in this cluster will appear the disappear so we can see **failures** and replication in action.

The cluster we will use for real work consists of four relatively large nodes that are **reliable**. It can be contacted at `node1.local`. This cluster can be used for real data storage: if you try to store data on the unreliable test cluster, you're going to have a bad time.

The CQL Shell & CQL Basics

Let's start looking at Cassandra using the interactive shell, CQLSH. On the cluster gateway,

```
cqlsh unreliable1.local
```

In this shell, you can type CQL statements.

Cassandra organizes its tables into "keyspaces", which is nice because it will let us keep our data separate. Create a keyspace named after **your SFU userid**:

```
CREATE KEYSPACE <userid> WITH REPLICATION = {
'class': 'SimpleStrategy', 'replication_factor': 2 };
```

When connecting, you need to activate that keyspace. In CQLSH, that means:

```
USE <userid>;
```

Create a simple table that we can use to experiment with:

```
CREATE TABLE test ( id int PRIMARY KEY, data text );
```

You can add a few rows to see how CQL works (like SQL in many ways).

```
INSERT INTO test (id, data) VALUES (1, 'initial');
INSERT INTO test (id, data) VALUES (2, 'secondary');
INSERT INTO test (id, data) VALUES (3, 'third');
UPDATE test SET data='tertiary' WHERE id=3;
SELECT * FROM test;
```

But notice that primary keys **must** be unique. Inserting another record with the same primary key overwrites it. This is sometimes called an *upsert*. [❓]

```
INSERT INTO test (id, data) VALUES (2, 'double');
SELECT * FROM test;
```

Filtering data by things other than their primary key is possibly expensive, so you have to confirm that you know the operation is doing a full table scan.

```
SELECT * FROM test WHERE data='initial';
SELECT * FROM test WHERE data='initial' ALLOW FILTERING;
```

But Cassandra does support secondary indexes, which will allow that column to be efficiently queried.

```
CREATE INDEX data_index ON test (data);
SELECT * FROM test WHERE data='initial';
```

You can also perform `INSERT`, `UPDATE`, and `DELETE` operations in an atomic batch:

```
BEGIN BATCH
  INSERT INTO test (id, data) VALUES (4, 'square');
  INSERT INTO test (id, data) VALUES (5, 'cinq');
APPLY BATCH;
```

Replication and Failures

There is one node in our Cassandra cluster that is extremely unreliable (or perhaps "is reliably unreliable"). The node `unreliable3.local` is up for 10 minutes at a time from 0–10 after the hour, then down from 10–20 minutes after the hour, up from 20–30, and so on. You can verify this by trying to connect directly to that node (with the command `cqlsh unreliable3.local`).

When you were working on the question above, you probably had no idea that this was happening. Because your replication factor was >1, the cluster would have transparently compensated for the missing node.

Let's try pushing our luck. Alter your keyspace so data is only replicated once:

```
ALTER KEYSPACE <userid> WITH REPLICATION = {
'class': 'SimpleStrategy', 'replication_factor': 1 };
```

Try a query on some of your data while the flaky node is up or down. [❓]

Restore your replication factor, and try the queries again to verify that it works even when one node is down. (You may have to wait for an up/down cycle to give Cassandra a chance to restore your replication factor with data from the missing node.)

```
ALTER KEYSPACE <userid> WITH REPLICATION = {
'class': 'SimpleStrategy', 'replication_factor': 2 };
```

After restoring the replication factor, you may have to wait for an up/down cycle to fully restore your keyspace's data.

While the unreliable node is down with replication 2, we can also experiment with Cassandra consistency levels, which control how many replicas must confirm a particular fact. [❓]

```
CONSISTENCY ONE;
INSERT INTO test (id, data) VALUES (6, 'hexadecimal');
SELECT * FROM test;
CONSISTENCY ALL;
INSERT INTO test (id, data) VALUES (7, 'sevenish');
INSERT INTO test (id, data) VALUES (9, 'neun');
SELECT * FROM test;
SELECT * FROM test WHERE id=1;
SELECT * FROM test WHERE id=2;
SELECT * FROM test WHERE id=3;
SELECT * FROM test WHERE id=4;
SELECT * FROM test WHERE id=5;
```

Loading Data Into Cassandra

Of course, the CQL shell is a good tool for experimenting or configuring basic things, but not how you usually interact with a database. Let's load some data into a Cassandra table with **(non-Spark)** Python. We will again use the NASA web server log data (on the cluster gateway's filesystem at `/home/bigdata/nasa-logs-1` and `/home/bigdata/nasa-logs-2` or http://cmpt732.csil.sfu.ca/datasets/).

Use the **reliable cluster** (contacts `['node1.local', 'node2.local']`) for this question and all other Cassandra work. You will need to create a another keyspace for yourself (named with your username) on this cluster as well.

We will read and store the same fields as last time: the requesting host, the datetime, the path, and the number of bytes.

```
CREATE TABLE nasalogs (
  host TEXT,
  datetime TIMESTAMP,
  path TEXT,
  bytes INT,
  -- possibly more fields for the primary key?
  PRIMARY KEY (???)
);
```

You can create the table manually: no need to have code that creates it in your program.

Primary Keys

This table **doesn't have a primary key**, so we need to add one before creating. We should choose a good one, which means something like:

1. Each record has a unique primary key. (This one is a strict requirement.)
2. Data that you want to operate on together is on the same node [or small number of nodes], so it can be fetched together.
3. The key distributes your data evenly around the nodes [that you want to use, which we will assume is all of them].

Cassandra gives us a way to easily control how data is distributed among nodes: the primary key can be compound:

```
CREATE TABLE example ( … PRIMARY KEY (x, y, z));
```

With a compound key, the **first component** will be used to determine which node(s) the record will be stored by. In this case, records with the same value for `x` will be on the same node, but the tuple `(x, y, z)` must be unique for each record.

All of this implies that you need to know something about how the data is queried in order to select the right primary key. Hint: we will do the same host bytes-transferred correlation calculation in assignment 7, so we will be aggregating data by host. To make the key unique, the right answer is likely to **add a UUID field** to make your overall primary key unique.

Create the table manually and record the statement in your `answers.txt`. [❓]

Actually Load The Data

Create a program `load_logs.py` with arguments for input directory, output keyspace, and table name. That is, the command line will be like this:

```
python3 load_logs.py /home/bigdata/nasa-logs-2 <userid> nasalogs
```

The the `cassandra-driver` package is already available on the cluster gateway; you can use pip to install it (and `cqlsh` if you like) on your own computer if you need to. Here is a minimalist hint on how it's used:

```
from cassandra.cluster import Cluster
cluster = Cluster(['node1.local', 'node2.local'])
session = cluster.connect(keyspace)
rows = session.execute('SELECT path, bytes FROM nasalogs WHERE host=%s', [somehost])
```

Note that this code connects to the reliable cluster, not the unreliable one you used above. **Use the reliable cluster all future Cassandra work**.

Since we're writing non-Spark Python here, we receive a reminder of how much Spark is doing for us. In particular: opening, uncompressing and iterating over input files. Since we're not using Spark, we have to do it manually. Here's a hint:

```
for f in os.listdir(input_dir):
    with gzip.open(os.path.join(input_dir, f), 'rt', encoding='utf-8') as logfile:
        for line in logfile:
            print(line)
```

Doing many independent `INSERT` queries will get your data into the cluster, but very slowly. You should **package your inserts (a few hundred at a time) into batch statements**. That will make a huge difference in the running time.

If you'd like to clear the data in your table in between runs, you can use this statement in CQLSH:

```
TRUNCATE nasalogs;
```

Checking The Data

Use CQLSH to run a query to again determine the total number of bytes transferred in the data set. [This will probably time out: we want the query you tried, but don't need the actual result.] [❓]

Loading Data With Spark

The `load_logs.py` was a good example of using Cassandra by itself, but waiting for it was boring. Let's use the cluster to work with Cassandra faster. We can throw away the data we loaded the slow way.

```
TRUNCATE nasalogs;
```

Write a Spark application `load_logs_spark.py` that does the same task but builds a DataFrame (similar to how we approached the log files last week) and uses the spark-cassandra-connector to write the data to the Cassandra cluster.

See Cassandra + Spark + Python instructions for how to get Spark talking to Cassandra.

Your program should take an input directory, output keyspace, and output table name, as before. Since we're now using HDFS data, the command will be like:

```
spark-submit --packages com.datastax.spark:spark-cassandra-connector_2.12:3.1.0 --conf spark.sql
```

Is loading the data going very slow? Remember that the number of partitions in your DataFrame controls the amount of parallelism in any task. Check the Spark frontend to see how many tasks your (Cassandra writing) job is being split into and fix it.

Server Log Correlation with Cassandra Data

Repeat the server log correlation question, using Spark and getting the input data from the Cassandra table you populated above.

Your program should be called `correlate_logs_cassandra.py` and take command line arguments for the input keyspace and table name. As before, simply `print` the `r` and `r**2` values.

```
spark-submit … correlate_logs_cassandra.py <userid> nasalogs
```

This should be as simple as combining pieces from the Cassandra instructions with your previous implementation of the correlation calculation.

Questions

In a text file `answers.txt`, answer these questions:

1. What happened when you inserted another row with the same primary key as an existing row?
2. What happened when you query a keyspace with replication factor 1 and one node down? How did it behave with replication factor 2 and a node down?
3. How did the consistency level affect the results with the node up/down?
4. Which of the `WHERE id=` values returned successfully with `CONSISTENCY ALL` when one of the nodes was down? Why do you think some could be returned but not others?
5. What was the `CREATE TABLE` statement you used for the `nasalogs` table? What was the primary key you choose, and why?
6. What was the CQL query you used (or tried) to get the total number of bytes?

Submission

Submit your files to the CourSys activity Assignment 8.

Updated Mon Oct. 31 2022, 09:07 by sgbaker.