Assignment 4B

Good News!

Our cluster will be more reliable after 4A is over: 199.60.17.136 will no longer have the ten minute up/down cycle.

Spark + Cassandra First Steps

We will use pyspark-cassandra *(https://github.com/anguenot/pyspark-cassandra)* to bring Spark and Cassandra together. It's not necessarily an easy pairing: there are some library incompatibilities that can be solved with an environment module on the cluster. When you run a Spark job using this library, you need to include the corresponding Spark Package *(https://spark-packages.org/)* :

```
spark-submit --packages anguenot:pyspark-cassandra:0.6.0 …
```

You need to set up a slightly different SparkContext:

```
cluster_seeds = ['199.60.17.171', '199.60.17.188']
from pyspark import SparkConf
import pyspark_cassandra
conf = SparkConf().setAppName('example code') \
        .set('spark.cassandra.connection.host', ','.join(cluster_seeds))
sc = pyspark_cassandra.CassandraSparkContext(conf=conf)
# spark = SparkSession.builder.getOrCreate()
assert sys.version_info >= (3, 5)  # make sure we have Python 3.5+
assert sc.version >= '2.2'  # make sure we have Spark 2.2+
```

This is the same as the Spark contexts we're used to, but it adds the `sc.cassandraTable` method *(https://github.com/anguenot/pyspark-cassandra#pyspark_cassandracassandrasparkcontext)* that creates an RDD from a Cassandra table, and adds a `.saveToCassandra` method *(https://github.com/anguenot/pyspark-cassandra#pysparkrdd)* to all RDDs, allowing them to be written to a Cassandra table.

See Cassandra + Spark + Python instructions for a few more details.

Loading Cassandra Data

The `load_logs.py` in 4A 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 as last week, but builds and RDD and uses the `.saveToCassandra` method to write the data to the Cassandra cluster. It 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 … load_logs_spark.py /courses/732/nasa-logs-2 <userid> nasalogs
```

Is loading the data going very slow? Remember two facts: the number of partitions in your RDD controls the amount of parallelism in any task; a compressed file cannot be split into multiple partitions while reading it because Spark can't start reading part way into the file. Check the Spark frontend to see how many tasks your (Cassandra writing) job is being split into and fix it.

Using Cassandra Data

Repeat the log correlation question from 2B, using Spark and getting the input data from the Cassandra table you just populated.

Your program should be called `correlate_logs.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.py <userid> nasalogs
```

It may be worth writing a quick function to get a Cassandra table into an RDD or DataFrame. These are mine, but you may have to adapt them:

```
def rdd_for(keyspace, table, split_size=None):
    rdd = sc.cassandraTable(keyspace, table, split_size=split_size,
        row_format=pyspark_cassandra.RowFormat.DICT).setName(table)
    return rdd
def df_for(keyspace, table, split_size=None):
    df = sqlContext.createDataFrame(sc.cassandraTable(keyspace, table, split_size=split_size
    df.createOrReplaceTempView(table)
    return df
```

Getting Relational Data

For this question, we will look at the TPC-H *(http://www.tpc.org/tpch/default.asp)* data set, which is a benchmark data set for relational databases. Since it's designed for relational databases, the assumption is that we're going to `JOIN` a lot. CQL doesn't have a `JOIN` operation, so we're going to have a mismatch.

The TPC-H specification *(http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf)* contains the full details, but we will leave a quick reference relationship diagram (figure 2 from page 13). On the cluster's HDFS, we have `/courses/732/tpch-0` (a scale factor 0.2 data set, 200MB), `tpch-1` (scale factor 0.5, 500MB), and `tpch-2` (scale factor 3.0, 3GB). The first two can be downloaded from http://cmpt732.csil.sfu.ca/datasets/ *(http://cmpt732.csil.sfu.ca/datasets/)* .

You will find the `tpch-2` data on our Cassandra cluster in the keyspace `tpch`. The tables there were created with these create tables statements, and loaded with this Spark job.

For this problem, we imagine that we want to frequently display a particular order with the names of the parts that were ordered. The SQL to get that is something like:

```
SELECT o.*, p.name FROM
  orders o
  JOIN lineitem l ON (o.orderkey = l.orderkey)
  JOIN part p ON (l.partkey = p.partkey)
WHERE …
```

Our command line will take the input keyspace, the output directory, and several orderkey values from the data set. You can get those like this:

```
keyspace = sys.argv[1]
outdir = sys.argv[2]
orderkeys = sys.argv[3:]
```

The command (for each version of the solution to this problem) will be like this:

```
spark-submit … tpch_orders_X.py tpch out1 2094435 6567939 5331751 797573 17487812
```

We want to produce a summary of each of the orders specified by the orderkey. We want the orderkey, the totalprice (a field on the orders table), and a list of the names of the parts ordered. We'll output each order on a line with the part names comma separated like this:

```
Order #17487812 $121518.34: navy coral lawn purple burlywood, peru blush grey rose spring, p
Order #2094435 $18792.15: slate blue orchid medium peach, turquoise brown lavender bisque gh
Order #6567939 $186200.28: linen aquamarine smoke olive lawn, misty mint slate beige floral,
Order #5331751 $213175.18: khaki tomato cream gainsboro firebrick, burnished rosy cream medi
Order #797573 $170408.23: peru medium firebrick ivory puff, magenta ghost dim chiffon grey,
```

The order of the lines, and of the part names within the lines, may vary. The actual output will be relatively small: we will give no more than ten or so orderkeys, and each order in the data set has at most seven parts.

Controlling Usage

Our cluster has dynamic allocation enabled by default: your jobs get more executors if there are tasks queued, and give them up if they are idle. In order to place nicely on the cluster, please include this in your SparkConf object for code in this question:

```
conf = SparkConf().….set('spark.dynamicAllocation.maxExecutors', 20)
```

Attempt #1: Let Spark SQL Do It

For our first attempt, we will fall back on old habits: we can use `sc.cassandraTable` to create an RDD and then DataFrame of the tables we need. Name this program `tpch_orders_sql.py`.

Create DataFrames from the Cassandra data, and use Spark SQL to join the tables and extract the data you need. In this scenario, Cassandra isn't much smarter than the files we have been using as input up to this point.

Attempt #2: Select What You Need

If we're using Spark SQL to hold the data from the three tables needed above, one fact is unescapable: we don't need most of that data. We know we need to output a small number of rows, so reading the entire table into a DataFrame shouldn't be necessary.

A CassandraRDD *(https://github.com/anguenot/pyspark-cassandra#pyspark_cassandracassandrardd)* (produced by the `.cassandraTable` method) gives us methods to build a smarter query for Cassandra. In particular, the `.where` method which lets us add a `WHERE` condition to the `SELECT` query we're implying.

Use the CassandraRDD more intelligently to get just the data you need out of the database. If you do this right, you are doing almost no Spark work (since all of the data sets are tiny). Name this program `tpch_orders_cassandra.py`.

Reshape The Data

The real problem here is that the data is in the wrong shape for Cassandra. Cassandra doesn't expect foreign keys and joins, and it's not good at dealing with them. All of the above was an attempt to force Cassandra to behave like a relational database, with Spark and/or Python picking up the slack.

The solution is to reshape our data so it can be efficiently queried for the results we need. Doing this requires us to know what queries we're going to do in the future, but we do for this question.

Denormalize The Data

Cassandra has a list data type *(https://docs.datastax.com/en/cql/3.1/cql/cql_using/use_list_t.html)* : we can use it to store the part names in the order table, so they're right there when we need them.

In your own keyspace, create a table `orders_parts` that has all of the same columns as the original `orders` table, plus a set of the part names for this order:

```
CREATE TABLE orders_parts (
  part_names set<text>,
  ⋮
);
```

Create a program `tpch_denormalize.py` that copies the data from the `orders` table and populates the `part_names` column as it's inserted. Your program should take two arguments: the input keyspace (with the TPC-H data), and the output keyspace (where the `orders_parts` table will be populated).

Hints… The DataFrame aggregate function `collect_list` or `collect_set` will likely be helpful here. If you have a DataFrame with the same schema as a Cassandra table, this is a convenient pattern:

```
dataframe.rdd.map(lambda r: r.asDict()).saveToCassandra(keyspace, table)
```

This will be an expensive, but one-time operation. We should be able to query the data we need quickly once it's done.

Attempt #3: Now Select What You Need

Repeat the above problem (giving your keyspace instead of `tpch`, since that's where the data is) and select the relevant data from the `orders_items` table. Name this program `tpch_orders_denorm.py`.

You should give the same output as above. Again, there will be very little Spark work here: really just fetching the Cassandra data and outputting.

Questions

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

1. Your “filter with Cassandra” and “query denormalized table” solutions were both likely fast enough that you couldn't distinguish between them. Under what circumstances do you think one would be noticeably faster than the other, or for what other reasons would choose one over the other? What kinds of queries/frequency would make either one better?
2. The “join in Spark SQL” solution was certainly slower. Are there situations where that would be the best choice, over one of the ones that were faster here?
3. Consider the logic that you would have to implement to **maintain** the denormalized data (assuming that the `orders` table had the `part_names` query in the main data set). Write a few sentences on what you'd have to do when inserting/updating/deleting data in this case.

Submission

Submit your files `load_logs_spark.py`, `correlate_logs.py`, `tpch_orders_sql.py`, `tpch_orders_cassandra.py`, `tpch_orders_denorm.py`, and `answers.txt` to the CourSys activity Assignment 4B.

Updated Tue Oct. 17 2017, 11:02 by ggbaker.