Assignment 8

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 in the last assignment.

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.

Working With 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).

The data sets we have available are `/courses/732/tpch-1` (a scale factor 0.2 data set, 200MB uncompressed), `tpch-2` (scale factor 0.5, 500MB), `tpch-3` (scale factor 3.0, 3GB), and `tpch-4` (scale factor 6.0, 6GB). These were generated with tpch-dbgen *(https://github.com/electrum/tpch-dbgen)* . You will find these data sets loaded to our (reliable) Cassandra cluster in the keyspaces `tpch1`, `tpch2`, `tpch3`, `tpch4`. The tables there were created with these create tables statements.

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 would be something like:

```
SELECT o.*, p.name FROM
  orders o
  JOIN lineitem l ON orderkey
  JOIN part p ON 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 will be like this:

```
time spark-submit --packages datastax:spark-cassandra-connector:2.3.1-s_2.11 tpch_orders_df.py tp
```

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 in text file(s) in the output directory given on the command line) with the part names comma separated like this:

```
Order #28710 $43012.31: cream hot dodger peru green, deep firebrick slate dim misty, midnight cor
Order #151201 $193245.63: blue papaya pink plum grey, chartreuse lace almond linen peru, cream ho
```

```
Order #193734 $88031.31: drab papaya spring burnished royal, rosy salmon aquamarine lavender choc
Order #810689 $276159.84: almond cornflower black lemon burnished, lemon indian azure orange stee
Order #986499 $44583.01: peru khaki coral rose midnight
```

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. The lines should be sorted by order number; the part names within the line should be alphabetical.

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:

```
spark = SparkSession.….config('spark.dynamicAllocation.maxExecutors', 16).getOrCreate()
```

Attempt #1: Let Spark Do It

For our first attempt, we will fall back on old habits: we can use the spark-cassandra-connector to create DataFrames of the tables we need. Name this program `tpch_orders_df.py`.

Create DataFrames from the Cassandra data, and use Spark to join the tables and extract the data you need. In this scenario, Cassandra is just providing the data we need for the calculations done in Spark.

Once you have the data joined appropriately, notice the aggregate function collect_set *(https://spark.apache.org/docs/2.3.1/api/python/pyspark.sql.html#pyspark.sql.functions.collect_set)* to get all of the item names in a single row. Then sort by orderkey, switch to an RDD, and you can use a function like this to produce the required output format:

```
def output_line(orderkey, price, names):
    namestr = ', '.join(sorted(list(names)))
    return 'Order #%d $%.2f: %s' % (orderkey, price, namestr)
```

Why So Fast?

If you run this on the `tpch4` keyspace, it probably runs in about a minute.

That doesn't sound right: you likely created DataFrames for the lineitem, part, and orders tables. The lineitem table alone in `tpch4` contains 36M records.

Have a look at the execution plan for your final DataFrame in this problem and figure out why this code runs so fast. Hint: your DataFrames are actually small and you should see “PushedFilters” in your execution plan: if not, you have done something to defeat the optimizer. [**?**]

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 picking up the slack.

If we needed to do this query often (maybe as part of a production system), this isn't a reasonable way to do it.

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 set data type *(https://docs.datastax.com/en/cql/3.1/cql/cql_using/use_set_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 all of the data from the `orders` table and adds the `part_names` column (containing part names as we found in the previous part) 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).

This will be an expensive, but one-time operation: **try it on the `tpch2` input** so we can compare the running times. We should be able to query the data we need quickly once it's done…

Attempt #2: 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_parts` table. Name this program `tpch_orders_denorm.py`.

You should give the same output as above. 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. What did you see in the execution plan for the “join in Spark” solution? Why was the execution so fast (and the memory usage so small)?
2. What was the `CREATE TABLE` statement you used for the `orders_parts` table?
3. What were the running times of the two `tpch_orders_*` programs on the `tpch2` data on the cluster? These orderkeys have results in that data set: `2579142 2816486 586119 441985 2863331`.
4. Consider the logic that you would have to implement to **maintain** the denormalized data (assuming that the `orders` table had the `part_names` column 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 to the CourSys activity Assignment 8.

Updated Fri Nov. 02 2018, 11:46 by ggbaker.