Performance Analysis of Cloud Relational Database Services

Jialin Li
lijl@cs.washington.edu
Naveen Kr. Sharma
naveenks@cs.washington.edu
Adriana Szekeres
aaazs@cs.washington.edu

June 7, 2013

1 Introduction and Motivation

The growing trend of cloud applications and big data brings a lot of interesting challenges as a software developer and computer engineer. Firstly, engineers have to install and build a reliable and scalable database system and monitor their status constantly to deal with failures. Further, developers have to customize their applications according to the underlying system and adapt to changes. This is time consuming as well as a wastage of resources. As a result, software giants like Amazon, Google and Microsoft have started offering data management services in the cloud. This is attractive for small to medium scale applications/companies who do not want to spend a lot of resources on maintaining expensive data management services. The developers need not worry about backend issues and can rapidly create, scale and extend applications into the cloud. They can also use the tools offered by these services to efficiently analyze large datasets quickly and efficiently.

As many big companies now offer this kind of service in the cloud, the clients, faced with multiple service providers, must make a decision on which of the available services to use. One of the questions he/she might have is about the value for money of each of the services he/she considers. On the other hand, he/she might be interested in just the performance or scalability of the services. In this project we address these questions by evaluating and benchmarking two such services from Amazon and Google.

2 Project Description and Approach

The goal of the project is to analyze the performance, value for money and scalability of several relational databases offered as cloud services for big data analysis. We will use the well known TPC-H benchmark to measure the performance and, if information is available, we will normalize the results taking into account the infrastructure on which each service is running. Our approach to benchmarking can be split up into the following phases:

Selecting the TPC Benchmark: First we select appropriate benchmark to evaluate the systems under consideration. The criteria here is that it should stress the specific parts of the database system which we want to measure. Since we are interested in high performance cloud services, we care only about performance (query evaluation time) and scalability (trend of query evaluation time with increasing data size and number of nodes).

Choosing the Cloud Services: Next, we choose the cloud services we will be benchmarking. Since we want a fair comparison, we only choose services which provide the same set of features, in our case high performance and scalable querying service for massive datasets. We had several options such as Amazon Redshift[6], Google BigQuery[2], Google CloudSQL[3], Microsoft Azure SQL[4], Amazon Dynamo[5] and several others. We describe our chosen services later in this section.

Running the Benchmark: This phase involves running the benchmark queries on different services with varying workloads and configuration parameters. As a baseline, we run the same queries with same dataset on a commodity desktop machine running PostgreSQL 9.1.
• Measure query response time for different types of queries in the TPC-H benchmark.
• Measure query response time for a specific query while varying dataset sizes.
• Measure query response time for a specific query while varying number of server nodes.

In the subsequent subsections, we describe the benchmark we chose and justify our choice. We also describe the two services which we decided to benchmark and what type of features they provide.

2.1 TPC-H Benchmark

There are two types of queries that are usually executed during the life span of a database: OLTP (Online Transaction Processing) and DSS (Decision Support). While OLTP queries are for information retrieval and information update, the latter type of queries help users make business decisions, e.g. determine quickly the trends in sales. The DSS queries are usually more complex and must deal with a larger volume of data. They are basically used for big data analysis. Both BigQuery and Redshift are decision support data warehouses that bring big data analysis into the cloud. They are optimized for high-performance analysis. Therefore, to evaluate the performance of the two systems, we will use a decision support benchmark, the TPC-H[1] benchmark.

The TPC-H database consists of 8 tables (part, partsupp, lineitem, orders, customer, supplier, nation, region) and the data can be generated for a set of fixed scale factors (SF) (1, 10, 30, 100, 300, 1000, 3000, 10000, 30000, 100000). Each table will have the following number of lines:

<table>
<thead>
<tr>
<th># Rows</th>
<th>part</th>
<th>partsupp</th>
<th>lineitem</th>
<th>orders</th>
<th>customer</th>
<th>supplier</th>
<th>nation</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF*200,000</td>
<td>SF*800,000</td>
<td>SF*6,000,000</td>
<td>SF*1,500,000</td>
<td>SF*150,000</td>
<td>SF*10,000</td>
<td>25</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

The scale factor also specifies how many GB of data will be generated. For example, for a SF of 30, 30GB of data will be generated, out of which the lineitem table will have 23GB of data. For our experiments we will generate data for a maximum of 100GB, i.e. using SF 1, 10, 30 and 100.

While Redshift required no changes to support TPC-H queries and data, BigQuery required substantial changes to the generated datasets, schemas and queries. First, the generated TPC-H data to populate the tables needed to be transformed into either CSV or JSON in order to be loaded by BigQuery. The CSV format is not appropriate for the TPC-H datasets because there are commas in the values of several fields. Therefore we had to transform the data into the JSON format. Second, BigQuery uses a simple SQL-like language that doesn’t have the same syntax as the TPC-H generated queries. Therefore, we had to do the following changes to the schemas and queries generated by TPC-H (See Appendix for the exact modifications):

• BigQuery does not support the DATE type, which appears in several places in the TPC-H schemas. Therefore, we used INTEGER instead of DATE and converted the DATE values into the equivalent POSIX time, defined as the number of seconds that have elapsed since midnight Coordinated Universal Time (UTC), 1 January 1970. We preserved the modification across the Redshift schemas. Also, the other data types supported by BigQuery are STRING, FLOAT, BOOLEAN, RECORD and TIMESTAMP. Therefore, we had to adapt the schemas generated by TPC-H to use these types.

• BigQuery does not support the LIKE operator, but instead it has the CONTAINS and REG-EXP.MATCH operators that we used.

• BigQuery does not support the EXISTS operator.

• In BigQuery one has to explicitly use the JOIN/JOIN EACH operator for the joins and each SELECT statement must contain at most one JOIN/JOIN EACH clause.

• BigQuery doesn’t support constructs like ” WHERE column_name <(SELECT .. FROM)” therefore we replaced these with joins.
2.2 Google BigQuery

Google BigQuery is a cloud-based interactive query service for massive datasets. BigQuery is the external implementation of one of Google’s core technologies called Dremel[7]. BigQuery can scan millions of rows without an index in a second by massively parallelizing each query and running them on tens of thousands of servers simultaneously. BigQuery uses the same underlying technology as Dremel which includes columnar storage for high scan throughput and high compression ratio, and tree architecture for dispatching queries and aggregating results across thousands of machines in a few seconds. BigQuery provides a simple interface to upload table(s) and run SQL queries on them. It abstracts away the underlying implementation or runtime details, hence the user has no knowledge of how many nodes are being used for processing. As a result BigQuery charges users per GB of data processed.

2.3 Amazon Redshift

Amazon Redshift is a fully managed cloud database management system. It uses columnar storage technology and automatically parallelize and distribute queries across multiple nodes. The interface exposed to users is a simple SQL like interface: Redshift supports most of the standard SQL queries and data types. No explicit parallel or distributed query commands are required, therefore, users can use the same set of queries regardless of the number of nodes. During setup, the number of Redshift nodes and the type of nodes need to be specified. There are two types of nodes available,

- a smaller node (XL) with 2 cores, 15GB of memory and 2TB of local storage
- a larger node (8XL) with 16 cores, 120GB of memory and 16TB of local storage

However, users still need to decide on data partitioning scheme. Redshift provides DIST KEY key word to indicate which field to use as data partition hash key. Skew may happen if a bad DIST KEY is selected and will inversely affect parallel query performance. Amazon charges Redshift users only according to the uptime of all Redshift servers reserved. Data transfer and processing do not incur charges.

3 Related Work

The ability to process large amounts of data into the cloud has been around for some time. Google has been using MapReduce since 2004 to process its big data. However, these solutions still required a fair amount of time and programming skills. Therefore, new systems, like BigQuery and Redshift, emerged that made possible interactive/real-time queries on mega amount of information. A query on billions of rows can take just seconds. These technologies are very new and we didn’t find any comparison studies on such systems. Instead, there are comparison studies on previous cloud based databases.

Prior work has looked at benchmarking cloud-based data management systems. Shi et al.[8] benchmarked some representative cloud-based DBMS and compared their performance under different workloads. Four systems were studied: HBase, Cassandra, Hive and HadoopDB. The first three are filesystem-based while HadoopDB is based on traditional DBMS. Two types of benchmarks were tested: data read/write (including both random and sequential) and structured SQL queries. The results showed that DBMS systems which use MapReduce as the framework perform better on queries. The two filesystem-based DBMS, HBase and Hive, have similar or even better performance compare to the DBMS based HadoopDB on SQL queries, even though they do not support SQL queries and require programmers to rewrite the queries using their own APIs. Our work differs from theirs in that we focus on the performance of large scale cloud DBMS on complex SQL queries. We use the TPC-H benchmark which contains more sophisticated queries than the simple grep, range queries. In addition, speed-ups and scale-ups will be the major tests on these systems. Shi et al.’s work only tested systems’ scalability on data read/write, but not on general SQL queries. We will perform experiments with TPC-H queries on different data sizes and node numbers.
4 Evaluation

In this section, we present the results of our benchmarking. We are interested in answering three main questions. First, how well do these queries speedup as we increase processing power. Second, what is the scaleup of these services, i.e. how does the query runtime vary with increasing dataset size. Lastly, we want to see how cost effective these services are, and which of them give better value for money.

4.1 Queries

We were able to port all TPC-H queries for BigQuery, except those queries that use views, which are not supported in BigQuery. We have included all the modified queries in the appendix. Although we ran most of the queries on both systems, we will present the results for only a few of them (Q1, Q2, Q6 and Q16), as they capture all the characteristics and trends exhibited by all the queries.

4.2 Redshift

There are two major sets of experiments we tested on Amazon’s Redshift: speed-ups and scale-ups. In the speed-up test, we keep the data size constant (100GB), increase the number of nodes and measure the time each query takes. In the ideal case, query execution time should decrease linearly as we increase the number of nodes. In the scale-up test, we increase the data size as well as the number of nodes, at the same rate. Therefore, we will have queries running on 1GB of data and on 1 node, 2GB of data on 2 nodes, 4GB of data on 4 nodes, etc. Query run time should ideally keep constant as we increase the scale factor (both data size and number of nodes). We first present the result for speed-up.

As shown in figure 1(a), the run time for two out of the four queries (query 1 and 6) decrease almost linearly as we increase the number of nodes. We found out by analyzing the query that these two queries are embarrassingly parallel. It means that Redshift has good speed-up when running queries that have high degree of parallelism. The speed-up for the other two queries, especially query 2, are worse mostly because of the low degree of parallelism. One thing we observed from our experiment is that there are some caching effects on Redshift. It means that running similar queries repeatedly will decrease query run time. To account for this effect, we also report the results for the warmed-up caches. This is shown in figure 1(b). We observed slightly better speed-ups for all queries, but the general trends remain the same.

![Figure 1: RedShift Speedup with 100GB dataset](image-url)

Figure 2(a) shows the scale-up effect of Redshift. The scale on the x-axis means the scaling factors we applied to both the number of nodes and the data size. As shown in the figure, the runtime for all the queries stay relatively constant across all scales. There are some fluctuations but there is no general increasing trend in any of the queries. It means that Redshift have good scale-up for different kinds of queries. We also report the results for the warmed-up cache in figure 2(b) which shows similar trends.
4.3 BigQuery

As described earlier, BigQuery provides the user with a blackbox query executor, i.e., the user has no control over how many server nodes are actually used for processing. Hence to evaluate how well BigQuery scales, we measure the runtime of all TPC-H queries with increasing dataset sizes. Figure 3 shows the results for TPC-H queries 1, 2, 6 and 16.

Queries 1 and 6 exhibit very good scaling, and BigQuery takes almost the same time to execute the query irrespective of dataset size. This is because queries 1, 6 are simple scans over the dataset to compute aggregate values and hence are massively parallelizable. Thus, BigQuery can use many more backend server nodes to process these two queries. However the runtime of query 2 and 16 increases significantly with increasing dataset size. This is because these two queries are inherently difficult to parallelize as they include joins over multiple tables and nested subqueries. This shows that BigQuery does an excellent job at speeding up easily parallelizable queries, however does not scale up queries with complex joins and subqueries.

While running queries on very large datasets, we frequently encountered "Resources exceeded during query execution" error in BigQuery. This was usually the case when the query involved joins on multiple tables and a group by clause. In some cases, we were able to get around the problem by rewriting the query manually. However, 6 queries still failed to run on the 100GB dataset.

4.4 Comparison

In this subsection, we compare head-to-head the performance of BigQuery and RedShift. By performance here, we mean the absolute runtime of a query. As a baseline performance, we run the same queries on same datasets on a commodity desktop machine running PostgreSQL 9.1 database management software. We show the results for TPC-H queries 1 and 2 in Figure 4. We can see several interesting trends in these figures.
First, as expected both BigQuery and Redshift are much faster than baseline PostgreSQL and the difference increases as we increase the dataset size. For TPC-H query 1 (massively parallelizable) Figure 4(a), we see that BigQuery clearly outperforms RedShift (with 1 node, as well as 8 nodes). Even the hot cache performance of RedShift is inferior to BigQuery. This is probably because BigQuery engine uses servers proportional to datasize to solve the massively parallel query.

Figure 4(b) shows the same plots for TPC-H query 2, and here the trends are very different. For smaller dataset sizes, BigQuery is faster that RedShift. However as we increase the dataset size, RedShift outperforms BigQuery easily. This is because TPC-H query 2 is inherently difficult to parallelize and has multiple joins and nested subqueries. RedShift does a much better job at scaling up complex queries, as compared to RedShift. It should be noted here that BigQuery is not designed to handle such complex queries (with multiple joins). It expects the data to be nested in itself as described in [7].

We also see some cache effects (difference in the lines hot, cold in Figure 4) in RedShift, and running the same query (with different parameters) on small datasets resulted in very fast query runtimes. However these effects disappeared with increasing dataset size as expected.

4.5 Pricing

BigQuery and Redshift use very different pricing schemes. While Googles BigQuery charges per unit of data processed, Amazons Redshift charges per hour per node. Depending on the queries, there might be advantages and disadvantages to both the schemes. In Figure 5(a) we computed the cost charged by the two systems on two queries, Q1 qnd Q2, executed on different data sizes. The graph shows that the prices are very fluctuant, depending a lot on the query characteristics. While Q1 touches the biggest table used in the TPC-H schema, i.e. lineitem, Q2 only touches the small tables. Therefore, BigQuery charges more than Amazon Redshift for Q1. Even though Q2 processes less amount of data, it is fast enough to be charged less by Redshift. In Figure 5(b) we computed the value per money. For Redshift’s run of Q1 it is noticeable how the value per money fluctuates, while for the others it is either decreasing or increasing. It would be interesting, for future work, to explore whether we could construct a learning algorithm or use static analysis of the query to decide on which system a query will give better value per money.

5 Conclusions

In this project, we ran TPC-H benchmark queries on two popular cloud based data processing services: Google BigQuery and Amazon RedShift. Based on the results and our own experiences using these services, we have identified the several advantages and disadvantages of both these services. BigQuery is very easy to setup and run queries and does not require any manual configuration of clusters; it automatically scales up according the dataset size. But this can be a disadvantage as well, since the user cannot tune the system according to his/her needs. However it has limited SQL language support and does not scale up well on complex queries involving multiple joins and nested subqueries.
RedShift on the other hand is almost fully SQL compatible, and all TPC-H queries run without any modification. However, it requires setting up and managing the cluster on which queries are run. This might be slightly difficult for users with limited background on such services. But this is a good feature for expert users, as it allows them to tune their cluster according to their requirements.

References
6 Appendix

<table>
<thead>
<tr>
<th>SQL</th>
<th>BigQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>select</td>
</tr>
<tr>
<td><code>s acctbal, s name, n name, p_partkey, p_mfgr, p_size, p_type, s address, s phone, s comment</code></td>
<td><code>s acctbal, s name, n name, p_partkey, p_mfgr, p_size, p_type, s address, s phone, s comment</code></td>
</tr>
<tr>
<td>from (select</td>
<td>from (select</td>
</tr>
<tr>
<td><code>s acctbal, s name, n name, p_partkey, p_mfgr, p_size, p_type, s address, s phone, s comment</code></td>
<td><code>s acctbal, s name, n name, p_partkey, p_mfgr, p_size, p_type, s address, s phone, s comment</code></td>
</tr>
<tr>
<td>from part supplier,</td>
<td>from p_partkey, p_mfgr, p_size, p_type, s_supkey, n name, n_regionkey, s name, s acctbal, s address, s phone, s supplycost, s comment</td>
</tr>
<tr>
<td>partsupp, nation region where</td>
<td>from</td>
</tr>
<tr>
<td><code>p_partkey = p_supkey</code> and <code>p_size = 42</code> and <code>p_type like ‘STEEL’</code> and <code>s_regionkey = n_regionkey</code> and <code>s_name = ‘AMERICA’</code> and <code>p_supplycost &lt;= (select min(p_supplycost) from part supplier, nation region where </code>p_partkey = p_supkey<code>and</code>s_supkey = p_supkey<code>and</code>s_regionkey = n_regionkey<code>and</code>s_name = ‘AMERICA’) ORDER by <code>s acctbal</code> desc, <code>n name</code>, <code>s name</code>, <code>p_partkey</code></td>
<td><code>p_partkey, p_mfgr, p_name, p_size, p_type, p_supplykey, p_supplycost, s_comment</code></td>
</tr>
</tbody>
</table>
Q3

```
SQL

SELECT
    l_orderkey,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue,
    o_orderdate,
    s_shippriority
FROM
    customer,
    orders,
    lineitem,
WHERE
    e_mktsegment = 'MACHINERY'
AND t1.c_key = o.c_key
AND o_orderdate < 795427200
AND l_shipdate > 795427200
GROUP BY
    o_orderdate,
    s_shippriority
ORDER BY
    revenue DESC;
```

```
BigQuery

SELECT
    t3.l_orderkey,
    SUM(t3.l_extendedprice * (1 - t3.l_discount)) AS revenue,
    j1.t2.o_orderdate,
    j1.t2.s_shippriority
FROM
    t3,
    t2,
    t1,
    j4,
    j3,
    j2,
    j1,
    t5,
    t4,
    c_suppkey,
    s_suppkey,
    c_nationkey,
    s_nationkey,
    c_regionkey,
    s_regionkey,
    c_name = 'EUROPE',
    s_name = 'EUROPE',
    o_orderdate >= 725875200
AND o_orderdate < 725875200 + 31556926
GROUP BY
    j1.t2.o_orderdate;
```

Q5

```
SQL

SELECT
    n_name,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
WHERE
    s_suppkey = s_suppkey
AND l_orderkey = l_orderkey
AND c_nationkey = c_nationkey
AND c_regionkey = c_regionkey
AND s_nationkey = s_nationkey
AND s_regionkey = s_regionkey
AND c_name = 'EUROPE',
s_name = 'EUROPE',
o_orderdate >= 725875200
AND o_orderdate < 725875200 + 31556926
GROUP BY
    n_name
ORDER BY
    revenue DESC;
```

```
BigQuery

SELECT
    j4.j3.j2.j1.t2.n_name,
    SUM(t6.l_extendedprice * (1 - t6.l_discount)) AS revenue
FROM
    t6,
    j4,
    j3,
    j2,
    j1,
    t5,
    t4,
    c_suppkey,
    s_suppkey,
    c_nationkey,
    s_nationkey,
    c_regionkey,
    s_regionkey,
    c_name = 'EUROPE',
s_name = 'EUROPE',
o_orderdate >= 725875200
AND o_orderdate < 725875200 + 31556926
GROUP BY
    j4.j3.j2.j1.t2.n_name
ORDER BY
    revenue DESC;
```

9
Q7

<table>
<thead>
<tr>
<th>SQL</th>
<th>BigQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>select </code></td>
<td><code>select </code></td>
</tr>
<tr>
<td><code>supp_nation, </code></td>
<td><code>supp_nation, </code></td>
</tr>
<tr>
<td><code>cust_nation, </code></td>
<td><code>cust_nation, </code></td>
</tr>
<tr>
<td><code>l_year, </code></td>
<td><code>l_year, </code></td>
</tr>
<tr>
<td><code>sum(volume) as revenue </code></td>
<td><code>sum(volume) as revenue </code></td>
</tr>
<tr>
<td>from</td>
<td>from</td>
</tr>
<tr>
<td><code>(select </code></td>
<td><code>(select </code></td>
</tr>
<tr>
<td><code>n1.name as supp_nation, </code></td>
<td><code>n1.name as supp_nation, </code></td>
</tr>
<tr>
<td><code>n2.name as cust_nation, </code></td>
<td><code>n2.name as cust_nation, </code></td>
</tr>
<tr>
<td><code>floor(l_shipdate/3600/24/365.25) as l_year, </code></td>
<td><code>floor(l_shipdate/3600/24/365.25) as l_year, </code></td>
</tr>
<tr>
<td><code>l_extendedprice * (1 - l_discount) as volume </code></td>
<td><code>l_extendedprice * (1 - l_discount) as volume </code></td>
</tr>
<tr>
<td>from</td>
<td>from</td>
</tr>
<tr>
<td><code>supplier, </code></td>
<td><code>supplier, </code></td>
</tr>
<tr>
<td><code>lineitem, </code></td>
<td><code>lineitem, </code></td>
</tr>
<tr>
<td><code>orders, </code></td>
<td><code>orders, </code></td>
</tr>
<tr>
<td><code>customer, </code></td>
<td><code>customer, </code></td>
</tr>
<tr>
<td><code>nation n1, </code></td>
<td><code>nation n1, </code></td>
</tr>
<tr>
<td><code>nation n2 </code></td>
<td><code>nation n2 </code></td>
</tr>
<tr>
<td>where</td>
<td>where</td>
</tr>
<tr>
<td><code>l_suppkey = l_suppkey </code></td>
<td><code>l_suppkey = l_suppkey </code></td>
</tr>
<tr>
<td>and <code>l_orderkey = l_orderkey </code></td>
<td>and <code>l_orderkey = l_orderkey </code></td>
</tr>
<tr>
<td>and <code>l_custkey = l_custkey </code></td>
<td>and <code>l_custkey = l_custkey </code></td>
</tr>
<tr>
<td>and <code>l_nationkey = n1.n_nationkey </code></td>
<td>and <code>l_nationkey = n1.n_nationkey </code></td>
</tr>
<tr>
<td>and <code>(a1.n_name = 'EGYPT' </code></td>
<td>and <code>(a1.n_name = 'EGYPT' </code></td>
</tr>
<tr>
<td>and <code>n2.n_name = 'ETHIOPIA' </code>) <code>and</code>n2.n_name = 'ETHIOPIA' `)</td>
<td></td>
</tr>
<tr>
<td>or <code>(a1.n_name = 'ETHIOPIA' </code></td>
<td>or <code>(a1.n_name = 'ETHIOPIA' </code></td>
</tr>
<tr>
<td>and <code>n2.n_name = 'EGYPT' </code>) <code>and</code>n2.n_name = 'EGYPT' `)</td>
<td></td>
</tr>
<tr>
<td>and <code>l_shipdate &gt;= 788947200 </code></td>
<td>and <code>l_shipdate &gt;= 788947200 </code></td>
</tr>
<tr>
<td>and <code>l_shipdate &lt;= 852019200 </code></td>
<td>and <code>l_shipdate &lt;= 852019200 </code></td>
</tr>
<tr>
<td>) as shipping `</td>
<td>) as shipping `</td>
</tr>
<tr>
<td>group by</td>
<td>group by</td>
</tr>
<tr>
<td><code>supp_nation, </code></td>
<td><code>supp_nation, </code></td>
</tr>
<tr>
<td><code>cust_nation, </code></td>
<td><code>cust_nation, </code></td>
</tr>
<tr>
<td><code>l_year </code></td>
<td><code>l_year </code></td>
</tr>
<tr>
<td>order by</td>
<td>order by</td>
</tr>
<tr>
<td><code>supp_nation, </code></td>
<td><code>supp_nation, </code></td>
</tr>
<tr>
<td><code>cust_nation, </code></td>
<td><code>cust_nation, </code></td>
</tr>
<tr>
<td><code>l_year </code></td>
<td><code>l_year </code></td>
</tr>
</tbody>
</table>
```
Q9

<table>
<thead>
<tr>
<th>SQL</th>
<th>BigQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>nation, o_year; sum(amount) as sum_profit from (select n_name as nation, floor(o_orderdate/3600/24/365.25) as o_year, l_extendedprice * (1 - l_discount) - p_supplycost * l_quantity as amount from (select l_extendedprice, l_quantity, o_orderdate, p_supplycost, p_name, n_name from (select l_extendedprice, l_quantity, l_orderkey, n_namekey, p_name, l_discount from (select l_suppkey, l_partkey, l_extendedprice, l_quantity, l_orderkey, n_namekey, l_discount from TPCCH10 supplier as T1 JOIN EACH TPCCH10 lineitem as T2 ON T1.l_suppkey = T2.l_suppkey) as T3 JOIN EACH TPCCH10 part as T4 ON T3.l_partkey = T4.p_partkey) as T5 JOIN EACH TPCCH10 orders as T6 ON T5.l_orderkey = T6.o_orderkey) as T7 JOIN EACH TPCCH10 nation as T8 ON T7.n_nationkey = T8.n_nationkey where p_name contains 'papaya' ) as T11 ) as profit group by nation, o_year order by nation, o_year desc;</td>
</tr>
</tbody>
</table>
```
### Q10

<table>
<thead>
<tr>
<th>SQL</th>
<th>BigQuery</th>
</tr>
</thead>
</table>
| **select**  
| `c_custkey`, `c_name`,  
| `sum(l_extendedprice * (1 - l_discount))` as revenue,  
| `c_acctbal`, `n_name`, `c_address`, `c_phone`, `c_comment`  
| from `customer`, `orders`, `lineitem`, `nation`  
| where `c_custkey` = `o_custkey`  
| and `o_orderkey` = `l_orderkey`  
| and `l_orderdate` >= 754732800  
| and `l_orderdate` < 754732800 + 7889229  
| and `l_returnflag` = 'R'  
| and `c_nationkey` = `n_nationkey`  
| group by `c_custkey`, `c_name`, `c_acctbal`, `c_address`, `c_phone`, `c_comment`  
| order by revenue desc; | **select**  
| `c_custkey`, `c_name`, `sum(l_extendedprice * (1 - l_discount))` as revenue,  
| `c_acctbal`, `n_name`, `c_address`, `c_phone`, `c_comment`  
| from (select `c_custkey`, `c_name`, `n_name`, `c_acctbal`, `c_address`, `c_phone`, `c_comment`, `n_nationkey`, `l_extendedprice`, `l_discount`  
| from (select `c_custkey`, `c_name`, `c_acctbal`, `c_address`, `c_phone`, `c_comment`  
| from `customer`, `orders`, `lineitem`, `nation`  
| where `c_custkey` = `o_custkey`  
| and `o_orderkey` = `l_orderkey`  
| and `o_orderdate` >= 754732800  
| and `o_orderdate` < 754732800 + 7889229  
| and `l_returnflag` = 'R'  
| and `c_nationkey` = `n_nationkey`  
| ) as T3  
| JOIN EACH `TPCH10.nation` AS T6  
| ON T5.c_nationkey = T6.n_nationkey) as T7  
| group by `c_custkey`, `c_name`, `c_acctbal`, `c_address`, `c_phone`, `c_comment`  
| order by revenue desc; |}

### Q13

<table>
<thead>
<tr>
<th>SQL</th>
<th>BigQuery</th>
</tr>
</thead>
</table>
| **select**  
| `c_count`, `count(*)` as custdist  
| from (select `c_custkey`, `count(*)` as orderkey  
| from `customer` left outer join `orders` on `c_custkey` = `o_custkey`  
| and `c_comment` not like '%express%accounts%'  
| ) as T1  
| order by custdist desc; | **select**  
| `f1`, `count(*)` as custdist  
| from (select `t1.c_custkey as fl`,  
| `count(t2.o_orderkey)` as f2  
| from `TPCH10.customer` AS t1  
| LEFT OUTER JOIN EACH `TPCH10.orders` AS T2  
| ON t1.c_custkey = t2.o_custkey  
| WHERE NOT REGEXP`MATCH(t2.o_comment, '.express.accounts.*')`  
| group by `t1`)  
| order by custdist desc, f2 desc; |
### Q16

<table>
<thead>
<tr>
<th>SQL</th>
<th>BigQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>select p.brand, p.type, p.size, count(DISTINCT ps.suppkey) as supplier_cnt from partsupp, part where p.partkey = ps.partkey and p.brand &lt;&gt; 'Brand#41' and p.type not like 'PROMO_BRUSHED' and p.size in (15, 46, 47, 34, 9, 22, 17, 43) and ps.suppkey not in (select suppkey from supplier where s.comment like '%Customer%Complaints%') group by p.brand, p.type, p.size order by supplier_cnt desc, p.brand, p.type, p.size;</td>
<td>select p.brand, p.type, p.size, count(DISTINCT ps.suppkey) as supplier_cnt from TPCH10.partsupp AS t1 JOIN EACH TPCH10.part AS t2 ON t2.p.partkey = t1.p.partkey where p.brand &lt;&gt; 'Brand#41' and NOT REGEXPMATCH(p.type, 'PROMO_BRUSHED') and p.size in (15, 46, 47, 34, 9, 22, 17, 43) and ps.suppkey not in (select suppkey from TPCH10.supplier where REGEXPMATCH(s.comment, '.*Customer.<em>Complaints.</em>') ) group by p.brand, p.type, p.size order by supplier_cnt desc, p.brand, p.type, p.size;</td>
</tr>
</tbody>
</table>

### Q17

<table>
<thead>
<tr>
<th>SQL</th>
<th>BigQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>select sum(l.extendedprice) / 7.0 as avg_yearly from lineitem, part where l.partkey = p.partkey and p.brand = 'Brand#42' and p.container = 'MED_DRUM' and l.quantity &lt; (select 0.2 * avg(l.quantity) as average from lineitem group by l.partkey ) order by l.partkey = p.partkey;</td>
<td>select sum(l1.t1.l.extendedprice) / 7.0 as avg_yearly from (select t1.l.quantity, t1.l.extendedprice, t2.p.container from TPCH10.lineitem AS t1 JOIN EACH TPCH10.part AS t2 ON t2.p.partkey = t1.l.partkey) AS j1 JOIN EACH (select l.partkey, 0.2 * avg(l.quantity) as average from TPCH10.lineitem GROUP EACH BY l.partkey ) AS j2 ON j1.t2.p.partkey = j2.l.partkey where j1.t2.p.brand = 'Brand#42' and j1.t2.p.container = 'MED_DRUM' and j1.t2.l.quantity &lt; j2.average;</td>
</tr>
<tr>
<td>SQL</td>
<td>BigQuery</td>
</tr>
<tr>
<td>-----</td>
<td>----------</td>
</tr>
</tbody>
</table>
| select
c.name,
c.custkey,
o.orderkey,
o.totalprice,
sum(l.quantity)
from
customer,
o.rders,
litem
where
o.orderkey in
(select
  l.orderkey
from
  litem
group by
  l.orderkey
having
  sum(l.quantity) > 315
) and c.custkey = o.custkey and o.orderkey = l.orderkey
group by
c.name,
c.custkey,
o.orderkey,
o.orderdate,
o.totalprice
order by
  o.totalprice desc,
o.orderdate; | select
c.name,
c.custkey,
o.orderkey,
o.orderdate,
o.totalprice,
sum(l.quantity)
from
customer,
o.rders,
litem
where
o.orderkey in
(select
  l.orderkey
from
  litem
group by
  l.orderkey
having
  sum(l.quantity) > 315
) and c.custkey = o.custkey and o.orderkey = l.orderkey
group by
c.name,
c.custkey,
o.orderkey,
o.orderdate,
o.totalprice
order by
  o.totalprice desc,
o.orderdate; |

Q18