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 quries 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 performace 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:

	part	partsupp	lineitem	orders	customer	supplier	nation	region
# Rows	SF*200,000	SF*800,000	SF*6,000,000	SF*1,500,000	SF*150,000	SF*10,000	25	5

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 REGEXP_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.

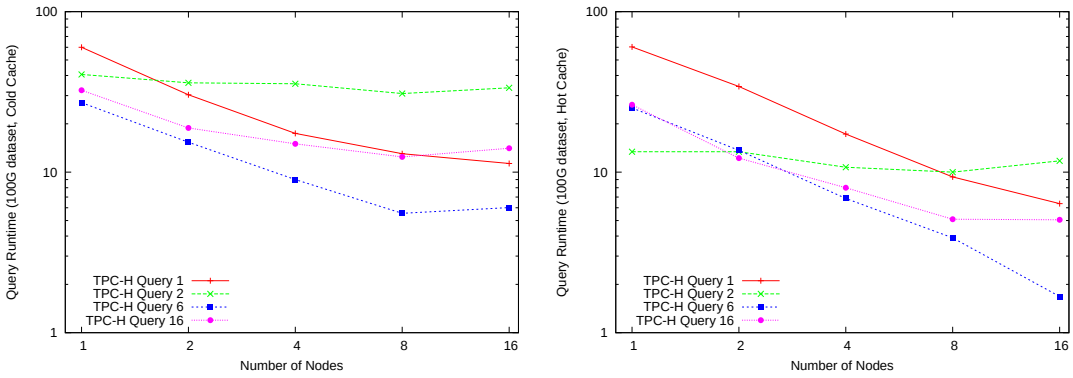
4.1 Queries

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 exibited 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.

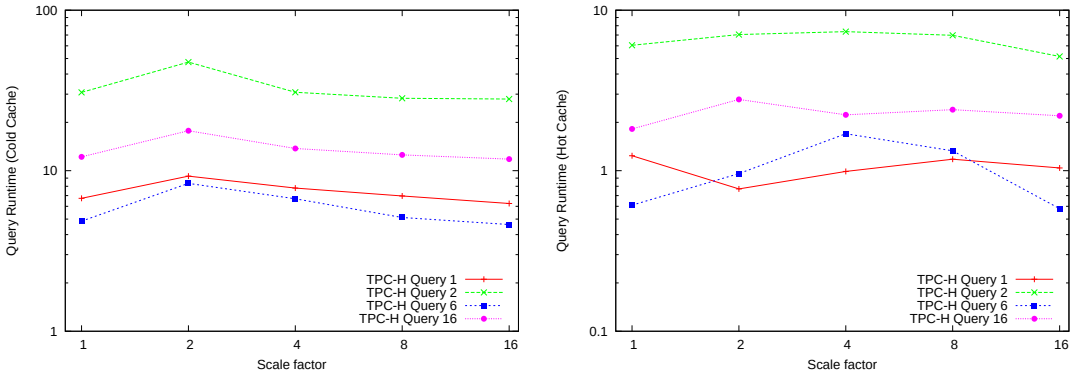


(a) Cold Cache

(b) Hot Cache

Figure 1: RedShift **Speedup** with 100GB dataset

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.



(a) Cold Cache (b) Hot Cache

Figure 2: RedShift **Scaleup** with 100GB dataset

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.

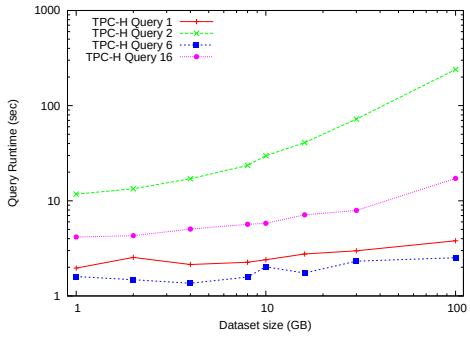


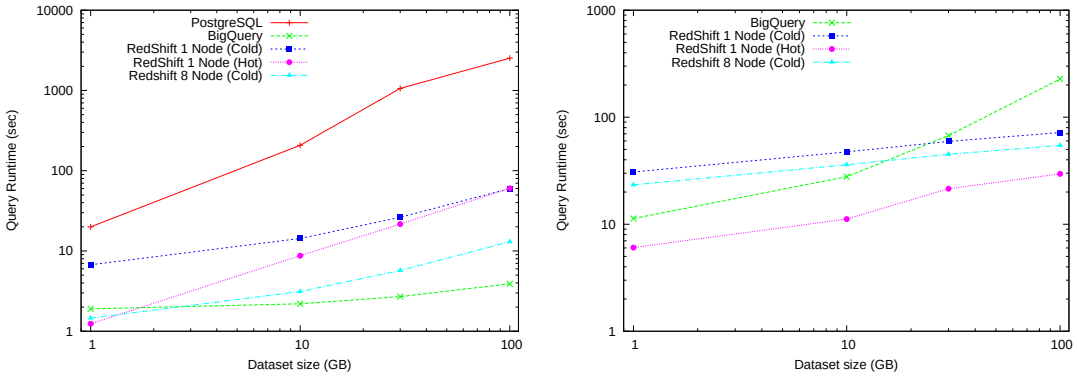
Figure 3: BigQuery Speedup and Scaleup

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 datasize. 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.



(a) TPC-H Query 1

(b) TPC-H Query 2

Figure 4: Comparison of query runtimes for different platforms

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 datasize, 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].

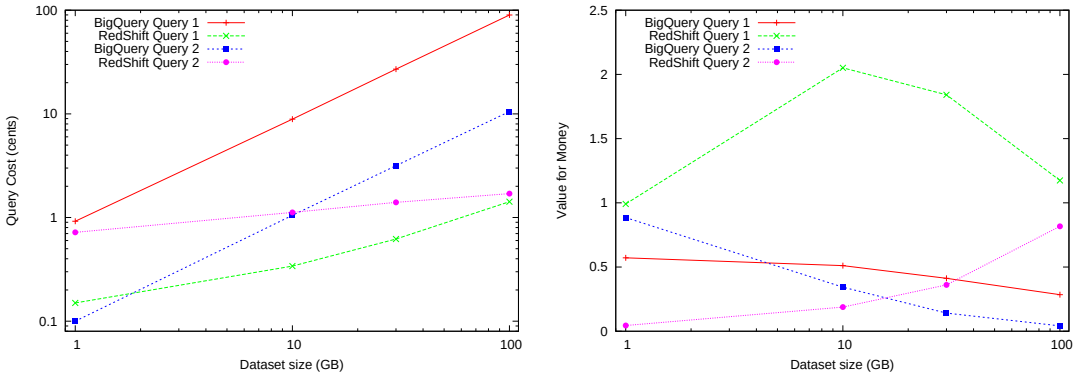
We also see some cache effects (difference in the lines hot, cold in Figure 4) in RedShift, and running the same query (with different parameters) on small datasets resulted in very fast query runtimes. However these effects disappeared with increasing dataset size as expected.

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 bigest 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.



(a) Cost of running a single query in cents

(b) Value for money of running a single query

Figure 5: Pricing comparison between BigQuery and Redshift

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 sligtly 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

[1] TPC `http://www.tpc.org`

[2] Google BigQuery `https://developers.google.com/bigquery/`

[3] Google CloudSQL `https://developers.google.com/cloud-sql/`

[4] Microsoft Azure SQL `http://www.windowsazure.com/en-us/manage/services/sql-databases/`

[5] Amazon Dynamo `http://aws.amazon.com/dynamodb/`

[6] Amazon Redshift `http://aws.amazon.com/redshift/`

[7] Melnik et al. Dremel: interactive analysis of web-scale datasets. VLDB 2010.

[8] Shi et al. Benchmarking cloud-based data management systems. CloudDB 2010.

6 Appendix

Q2

SQL

```
select
  s_acctbal,
  s_name,
  n_name,
  p_partkey,
  p_mfgr,
  s_address,
  s_phone,
  s_comment
from
  part,
  supplier,
  partsupp,
  nation,
  region
where
  p_partkey = ps_partkey
  and s_suppkey = ps_suppkey
  and p_size = 42
  and p_type like '%STEEL'
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = 'AMERICA'
  and ps_supplycost = (
    select
      min(ps_supplycost)
    from
      partsupp,
      supplier,
      nation,
      region
    where
      p_partkey = ps_partkey
      and s_suppkey = ps_suppkey
      and s_nationkey = n_nationkey
      and n_regionkey = r_regionkey
      and r_name = 'AMERICA')
order by
  s_acctbal desc,
  n_name,
  s_name,
  p_partkey;
```

BigQuery

```
select
  s_acctbal, s_name, n_name,
  p_partkey, p_mfgr, p_size,
  p_type, s_address, s_phone,
  r_name, s_comment
from
  (select
     s_acctbal, s_name, n_name,
     p_partkey, p_mfgr, p_size,
     p_type, s_address, s_phone,
     r_name, ps_supplycost, s_comment
   from
     (select
        p_partkey, p_mfgr,
        p_name, p_size, p_type,
        ps_suppkey, n_name, n_regionkey,
        s_name, s_acctbal, s_address,
        s_phone, ps_supplycost, s_comment
      from
        (select
           p_partkey, p_mfgr, p_name,
           p_size, p_type, ps_suppkey,
           ps_supplycost, s_address,
           s_phone, s_comment, s_acctbal,
           s_nationkey, s_name
         from
           (select
              p_partkey, p_mfgr, p_name,
              p_size, p_type, ps_suppkey,
              ps_supplycost
            from TPCH10.part as T1
                 JOIN EACH TPCH10.partsupp AS T2
                 ON T1.p_partkey = T2.ps_partkey) AS T3
           JOIN EACH TPCH10.supplier AS T4
           ON T3.ps_suppkey = T4.s_suppkey) AS T5
        JOIN EACH TPCH10.nation AS T6
        ON T5.s_nationkey = T6.n_nationkey) AS T7
     JOIN EACH TPCH10.region AS T8
     ON T7.n_regionkey = T8.r_regionkey) AS T9
  JOIN EACH (select
               min(ps_supplycost) AS min_ps_supplycost,
               ps_partkey
             from
               (select
                  ps_supplycost, ps_partkey, n_regionkey
                from
                  (select
                     ps_supplycost, ps_partkey, s_nationkey
                   from
                     (select
                        ps_suppkey, ps_partkey, ps_supplycost
                      from TPCH10.partsupp) AS T03
                     JOIN EACH TPCH10.supplier AS T04
                     ON T03.ps_suppkey = T04.s_suppkey) AS T05
                  JOIN EACH TPCH10.nation AS T06
                  ON T05.s_nationkey = T06.n_nationkey) AS T07
               JOIN EACH TPCH10.region AS T08
               ON T07.n_regionkey = T08.r_regionkey
             where r_name = 'AMERICA'
             GROUP EACH BY ps_partkey) AS T10
  ON T9.ps_supplycost = T10.min_ps_supplycost
     and T9.p_partkey = T10.ps_partkey
where
  p_size = 42
  and REGEXP_MATCH(p_type, '.*STEEL')
  and r_name = 'AMERICA'
order by
  s_acctbal desc,
  n_name,
  s_name,
  p_partkey;
```

Q3

SQL

```
select
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate,
  o_shippriority
from
  customer,
  orders,
  lineitem
where
  c_mktsegment = 'MACHINERY'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < 795427200
  and l_shipdate > 795427200
group by
  l_orderkey,
  o_orderdate,
  o_shippriority
order by
  revenue desc,
  o_orderdate;
```

BigQuery

```
select
  t3.l_orderkey,
  sum(t3.l_extendedprice * (1 - t3.l_discount))
    as revenue,
  j1.t2.o_orderdate,
  j1.t2.o_shippriority
from
  (select
     t1.c_mktsegment,
     t2.o_orderkey,
     t2.o_orderdate,
     t2.o_shippriority
   from TPCH10.customer as t1
        JOIN EACH TPCH10.orders as t2
        ON t1.c_custkey = t2.o_custkey) as j1
  JOIN EACH TPCH10.lineitem as t3
  ON t3.l_orderkey = j1.t2.o_orderkey
where
  j1.t1.c_mktsegment = 'MACHINERY'
  and j1.t2.o_orderdate < 795427200
  and t3.l_shipdate > 795427200
group by
  t3.l_orderkey,
  j1.t2.o_orderdate,
  j1.t2.o_shippriority
order by
  revenue desc,
  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
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and c_nationkey = s_nationkey
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = 'EUROPE'
  and 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
  (select
     t5.o_orderdate,
     t5.o_orderkey,
     j3.j2.t3.s_suppkey,
     j3.j2.j1.t2.n_name,
     j3.j2.j1.t1.r_name
   from
     (select
        t4.c_custkey,
        j2.t3.s_suppkey,
        j2.j1.t2.n_name,
        j2.j1.t1.r_name
      from
        (select
           t3.s_nationkey,
           t3.s_suppkey,
           j1.t2.n_name,
           j1.t1.r_name
         from
           (select
              t2.n_name,
              t2.n_nationkey,
              t1.r_name
            from TPCH10.region AS t1
                 JOIN EACH TPCH10.nation as t2
                 ON t1.r_regionkey = t2.n_regionkey) AS j1
           JOIN EACH TPCH10.supplier as t3
           ON t3.s_nationkey = j1.t2.n_nationkey) AS j2
        JOIN EACH TPCH10.customer as t4
        ON t4.c_nationkey = j2.t3.s_nationkey) AS j3
     JOIN EACH TPCH10.orders as t5
     ON t5.o_custkey = j3.t4.c_custkey) AS j4
  JOIN EACH TPCH10.lineitem as t6
  ON t6.l_orderkey = j4.t5.o_orderkey
     and t6.l_suppkey = j4.j3.j2.t3.s_suppkey

where
  j4.j3.j2.j1.t1.r_name = 'EUROPE'
  and j4.t5.o_orderdate >= 725875200
  and j4.t5.o_orderdate < 725875200 + 31556926
group by
  j4.j3.j2.j1.t2.n_name
order by
  revenue desc;
```

Q7

SQL

```
select
  supp_nation,
  cust_nation,
  l_year,
  sum(volume) as revenue
from
  (select
     n1.n_name as supp_nation,
     n2.n_name as cust_nation,
     floor(l_shipdate/3600/24/365.25) as l_year,
     l_extendedprice * (1 - l_discount) as volume
   from
     supplier,
     lineitem,
     orders,
     customer,
     nation n1,
     nation n2
   where
     s_suppkey = l_suppkey
     and o_orderkey = l_orderkey
     and c_custkey = o_custkey
     and s_nationkey = n1.n_nationkey
     and c_nationkey = n2.n_nationkey
     and (
          (n1.n_name = 'EGYPT'
              and n2.n_name = 'ETHIOPIA')
           or (n1.n_name = 'ETHIOPIA'
                and n2.n_name = 'EGYPT')
          )
     and l_shipdate >= 788947200
     and l_shipdate <= 852019200
  ) as shipping
group by
  supp_nation,
  cust_nation,
  l_year
order by
  supp_nation,
  cust_nation,
  l_year;
```

BigQuery

```
select
  supp_nation,
  cust_nation,
  l_year,
  sum(volume) as revenue
from
  (select
     n1_name as supp_nation,
     n2_name as cust_nation,
     floor(l_shipdate/3600/24/365.25) as l_year,
     l_extendedprice * (1 - l_discount) as volume
   from
     (select
        l_shipdate, l_extendedprice, l_discount,
        n1_name, n_name as n2_name
      from
        (select
           l_shipdate, l_extendedprice, c_nationkey,
           l_discount, n_name as n1_name
         from
           (select
              l_shipdate, l_extendedprice, s_nationkey,
              c_nationkey, l_discount
            from
              (select
                 o_custkey, l_shipdate,
                 l_extendedprice, s_nationkey,
                 l_discount
               from
                 (select
                    l_orderkey, l_shipdate,
                    l_extendedprice, s_nationkey,
                    l_discount
                  from TPCH10.supplier as T1
                       JOIN EACH TPCH10.lineitem as T2
                       ON T1.s_suppkey = T2.l_suppkey) as T3
                 JOIN EACH TPCH10.orders as T4
                 ON T3.l_orderkey = T4.o_orderkey) as T5
              JOIN EACH TPCH10.customer as T6
              ON T5.o_custkey = T6.c_custkey) as T7
           JOIN EACH TPCH10.nation as T8
           ON T7.s_nationkey = T8.n_nationkey) as T9
         JOIN EACH TPCH10.nation as T10
         ON T9.c_nationkey = T10.n_nationkey) as T11
   where (
         (n1_name = 'EGYPT' and n2_name = 'ETHIOPIA')
         or (n1_name = 'ETHIOPIA' and n2_name = 'EGYPT')
         )
         and (
         l_shipdate >= 788947200 and l_shipdate <= 852019200
         )
  ) as shipping
group by
  supp_nation,
  cust_nation,
  l_year
order by
  supp_nation,
  cust_nation,
  l_year;
```

Q9

SQL

```
select
  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)
       - ps_supplycost * l_quantity as amount
   from
     part,
     supplier,
     lineitem,
     partsupp,
     orders,
     nation
   where
     s_suppkey = l_suppkey
     and ps_suppkey = l_suppkey
     and ps_partkey = l_partkey
     and p_partkey = l_partkey
     and o_orderkey = l_orderkey
     and s_nationkey = n_nationkey
     and p_name like '%papaya%'
  ) as profit
group by
  nation,
  o_year
order by
  nation,
  o_year desc;
```

BigQuery

```
select
  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)
       - ps_supplycost * l_quantity as amount
   from
     (select
        l_extendedprice, l_quantity,
        o_orderdate, ps_supplycost,
        l_discount, p_name, n_name
      from
        (select
           l_extendedprice, l_quantity,
           s_nationkey, o_orderdate,
           ps_supplycost, p_name, l_discount
         from
           (select
              l_extendedprice, l_quantity,
              l_orderkey, s_nationkey,
              ps_supplycost, p_name, l_discount
            from
              (select
                 l_suppkey, l_partkey,
                 l_extendedprice, l_quantity,
                 l_orderkey, s_nationkey,
                 p_name, l_discount
               from
                 (select
                    l_suppkey, l_partkey,
                    l_extendedprice, l_quantity,
                    l_orderkey, s_nationkey, l_discount
                  from TPCH10.supplier as T1
                       JOIN EACH TPCH10.lineitem as T2
                       ON T1.s_suppkey = T2.l_suppkey) as T3
                 JOIN EACH TPCH10.part as T4
                 ON T3.l_partkey = T4.p_partkey) as T5
               JOIN EACH TPCH10.partsupp as T6
               ON T5.l_partkey = T6.ps_partkey
                  and T5.l_suppkey = T6.ps_suppkey) as T7
            JOIN EACH TPCH10.orders as T8
            ON T7.l_orderkey = T8.o_orderkey) as T9
         JOIN EACH TPCH10.nation as T10
         ON T9.s_nationkey = T10.n_nationkey
        where
          p_name contains 'papaya'
     ) as T11
  ) as profit
group by
        nation,
        o_year
order by
        nation,
        o_year desc;
```

Q10

SQL

```
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 l_orderkey = o_orderkey
  and o_orderdate >= 754732800
  and o_orderdate < 754732800 + 7889229
  and l_returnflag = 'R'
  and c_nationkey = n_nationkey
group by
  c_custkey,
  c_name,
  c_acctbal,
  c_phone,
  n_name,
  c_address,
  c_comment
order by
  revenue desc;
```

BigQuery

```
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, l_extendedprice, l_discount
   from
     (select
        c_custkey, c_name, c_acctbal,
        c_address, c_phone, c_comment,
        c_nationkey, l_extendedprice,
        l_discount
      from
        (select
           c_custkey, c_name, c_acctbal,
           c_address, c_phone, c_comment,
           c_nationkey, o_orderkey
         from TPCH10.customer as T1
              JOIN EACH TPCH10.orders as T2
              ON T1.c_custkey = T2.o_custkey
         where o_orderdate >= 754732800
               and o_orderdate < 754732800 + 7889229
        ) as T3
        JOIN EACH TPCH10.lineitem as T4
        ON T3.o_orderkey = T4.l_orderkey
        where T4.l_returnflag = 'R') as T5
     JOIN EACH TPCH10.nation as T6
     ON T5.c_nationkey = T6.n_nationkey) as T7
group by
        c_custkey,
        c_name,
        c_acctbal,
        c_phone,
        n_name,
        c_address,
        c_comment
order by
        revenue desc;
```

Q13

SQL

```
select
  c_count,
  count(*) as custdist
from
  (select
     c_custkey,
     count(o_orderkey)
   from
     customer left outer join orders on
     c_custkey = o_custkey
     and o_comment not like '%express%accounts%'
   group by
     c_custkey
  ) as c_orders (c_custkey, c_count)
group by
  c_count
order by
  custdist desc,
  c_count desc;
```

BigQuery

```
select
  f2,
  count(*) as custdist
from
  (select
     t1.c_custkey as f1,
     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
     f1
  )
group by
  f2
order by
  custdist desc,
  f2 desc;
```

Q16

SQL

```
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 <> '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
       s_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;
```

BigQuery

```
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.ps_partkey
where
  p_brand <> 'Brand#41'
  and NOT REGEXP_MATCH(p_type, 'PROMO_BRUSHED.*')
  and p_size in (15, 46, 47, 34, 9, 22, 17, 43)
  and ps_suppkey not in
    (select
       s_suppkey
     from
       TPCH10.supplier
     where
       REGEXP_MATCH(s_comment, '.*Customer.*Complaints.*')
    )
group by
  p_brand,
  p_type,
  p_size
order by
  supplier_cnt desc,
  p_brand,
  p_type,
  p_size;
```

Q17

SQL

```
select
  sum(l_extendedprice) / 7.0 as avg_yearly
from
  lineitem,
  part
where
  p_partkey = l_partkey
  and p_brand = 'Brand#42'
  and p_container = 'MED_DRUM'
  and l_quantity <
    (select
       0.2 * avg(l_quantity)
     from
       lineitem
     where
       l_partkey = p_partkey
    );
```

BigQuery

```
select
  sum(j1.t1.l_extendedprice) / 7.0 as avg_yearly
from
  (select
     t1.l_quantity, t1.l_extendedprice,
     t2.p_partkey, t2.p_brand,
     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.t1.l_quantity < j2.average;
```

Q18

SQL

```
select
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice,
  sum(l_quantity)
from
  customer,
  orders,
  lineitem
where
  o_orderkey in
    (select
       l_orderkey
     from
       lineitem
     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;
```

BigQuery

```
select
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice,
  sum(l_quantity)
from
  (select
     c_name, c_custkey,
     o_orderkey, o_orderdate,
     o_totalprice, l_quantity
   from
     (select
        c_name, c_custkey,
        o_orderkey, o_orderdate,
        o_totalprice
      from TPCH10.customer as T1
           JOIN EACH TPCH10.orders as T2
           ON T1.c_custkey = T2.o_custkey) as T3
     JOIN EACH TPCH10.lineitem as T4
     ON T3.o_orderkey = T4.l_orderkey) as T5
where
  o_orderkey in
    (select
       l_orderkey
     from
       (select
          l_orderkey, sum(l_quantity)
        from
          TPCH10.lineitem
        group each by
          l_orderkey
        having
          sum(l_quantity) > 315
       )
    )
group each by
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice
order by
  o_totalprice desc,
  o_orderdate;
```