# Apache Druid and Google BigQuery Performance Evaluation

Evaluating enterprise data warehouse performance using the Star Schema Benchmark

WHITE PAPER

**June 2020**

## Executive summary

Imply Data evaluated the performance of Apache Druid and Google BigQuery to determine the suitability of each as an enterprise data warehouse (EDW) solution. Each solution was evaluated for query performance using the Star Schema Benchmark. Further, a price-performance comparison was conducted between Apache Druid and Google BigQuery. Tests were designed to conduct a fair and repeatable comparison between EDW solutions. All configurations, schema, queries and test scripts are available via a GitHub repository.

| **Key Findings:** |
|---|
| In Star Schema Benchmark query performance tests:<br>❑ Apache Druid outperforms Google BigQuery by 321 percent in our testing. Total average response time for the query flight in Druid was 6043 ms, compared to 19409 ms in BigQuery.<br>❑ Apache Druid exhibits a 12x price-performance advantage over Google BigQuery. |

An EDW is a database, or a collection of databases, that centralizes information from multiple sources and applications to make it available for analytics use across an entire organization. EDWs have traditionally been housed on-premises, although recent years have seen a cloud-adoption surge. The data stored in an EDW can be one of a business's most valuable assets as it represents everything an organization knows and tracks across business units related to the functioning of the business, and interactions with partners, employees and customers. However, tremendous challenges exist related to the organization and structure of data across the enterprise, and the ETL process required prior to ingestion by an EDW. Many EDW rely on a star schema, one or more fact tables referencing a multitude of dimension tables, to accomplish this feat.

The Star Schema Benchmark (SSB) is designed to evaluate database system performance of star schema data warehouse queries. The schema for SSB is based on the TPC-H benchmark, but in a highly modified form. The SSB has been used to measure query performance of commercial and open-source database products on Linux and Windows since January 2007. Testing using the performance results of 13 standard SQL queries allows for comparison between products and configurations.

This testing evaluates the suitability of Apache Druid and Google BigQuery for EDW workloads in terms of performance and price-performance using SSB. EDW workloads are shifting to the cloud and, as a result, a new class of technologies is emerging that can provide fast query response times at scale. These solutions load, store and analyze large amounts of data at high speed to prove timely business insights. New columnar architectures provide microsecond response time at high levels of concurrency where traditional EDW struggle. When deployed elastically as a service, they enable enterprises to innovate BI and OLAP apps at a more rapid pace.

Drawing upon work previously conducted in academia , we ran the SSB queries in their standard form and, whenever possible, in an optimized form, and recorded results using Apache JMeter to compare performance of Imply Cloud (Apache Druid) and Google BigQuery. We relied on standard on-demand BigQuery access to process queries via the HTTP API. Please refer to the Testing Methodology section of this report for complete details on how we conducted these tests.

Figure 1 shows the overall Star Schema Benchmark test results for Apache Druid and Google BigQuery. In our test configurations, Apache Druid outperforms Google BigQuery by 321% on the aggregate SSB query flight. Total average response time for the query flight in Druid was 6043 ms, compared to 19409 ms in BigQuery. For these tests, lower average response times are better.

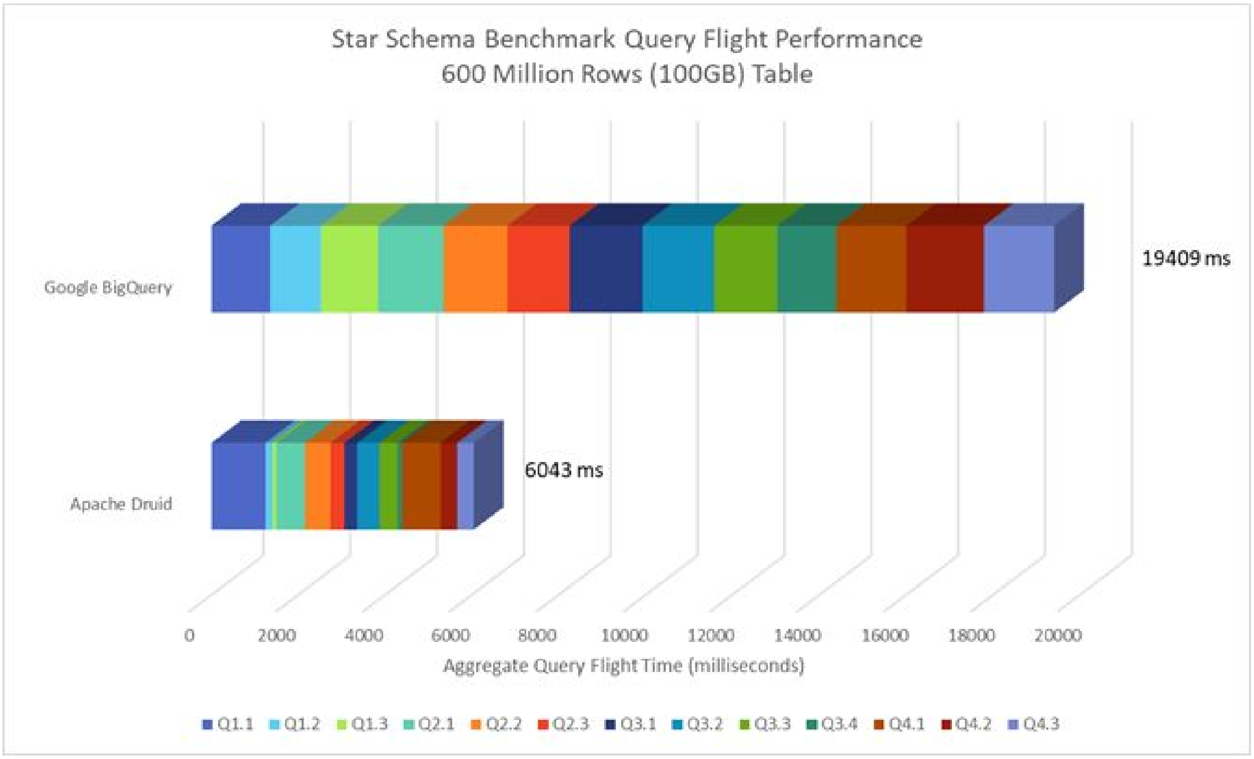


Figure 1. Star Schema Benchmark Test Results (average query response time) for Apache Druid and Google BigQuery. For these tests, lower is better.

We conducted a price performance comparison for Apache Druid and Google BigQuery. In our testing, we found that Druid outperformed BigQuery by 3.2 times (Total Query Time) at a much lower cost. We modeled price-performance of Druid to BigQuery using the SSB workload run on an enterprise scale over a month and found that Druid has a 12x (11.858x) price-performance advantage over Google BigQuery.

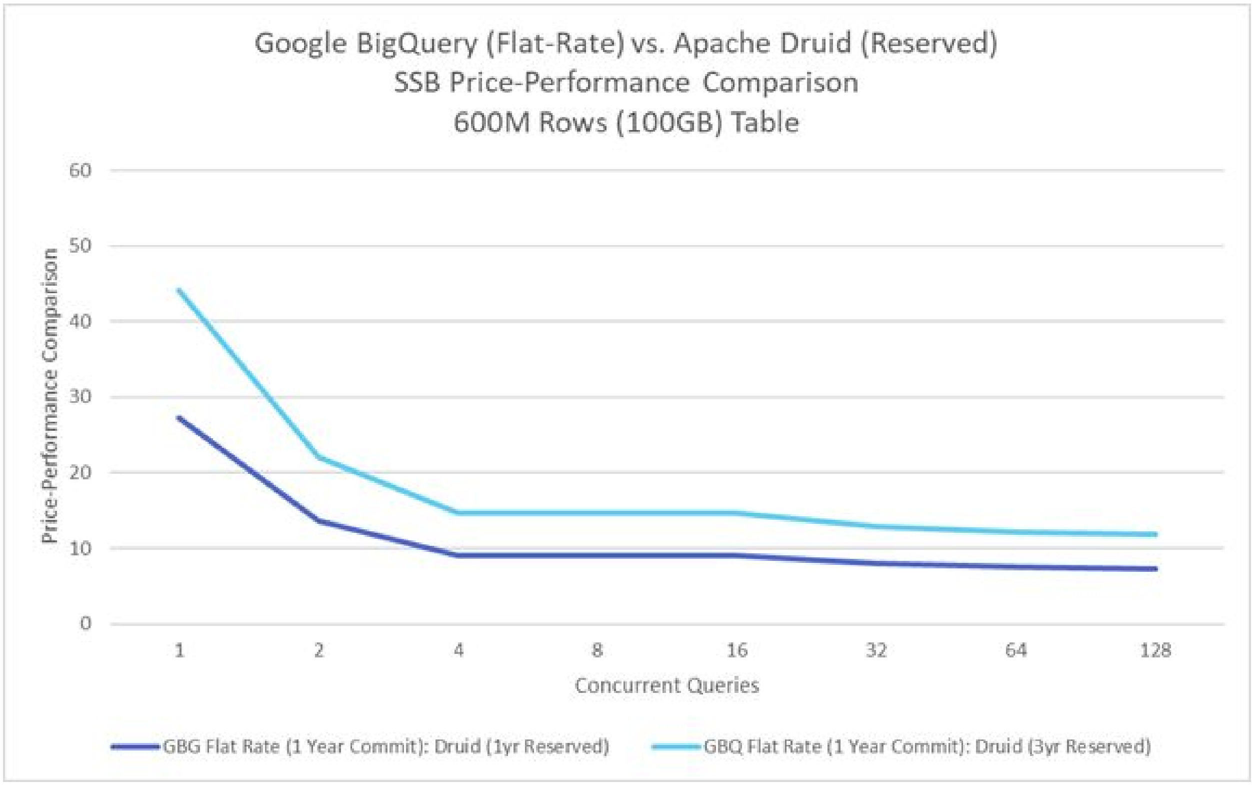


Figure 2. SSB Price-Performance Comparison: Google BigQuery (Flat-Rate) to Apache Druid (Reserved). Google BigQuery Flat-Rate pricing with one year commit is shown against Apache Druid pricing on reserved AWS instances with one year and three year commit. Price-performance ratio is shown for increasing levels of concurrent queries.

Concurrent queries can be used as an indication of flat-rate monthly cost, while queries per month indicates how BigQuery on-demand cost scales with respect to the cost of an equivalently performing Druid cluster.

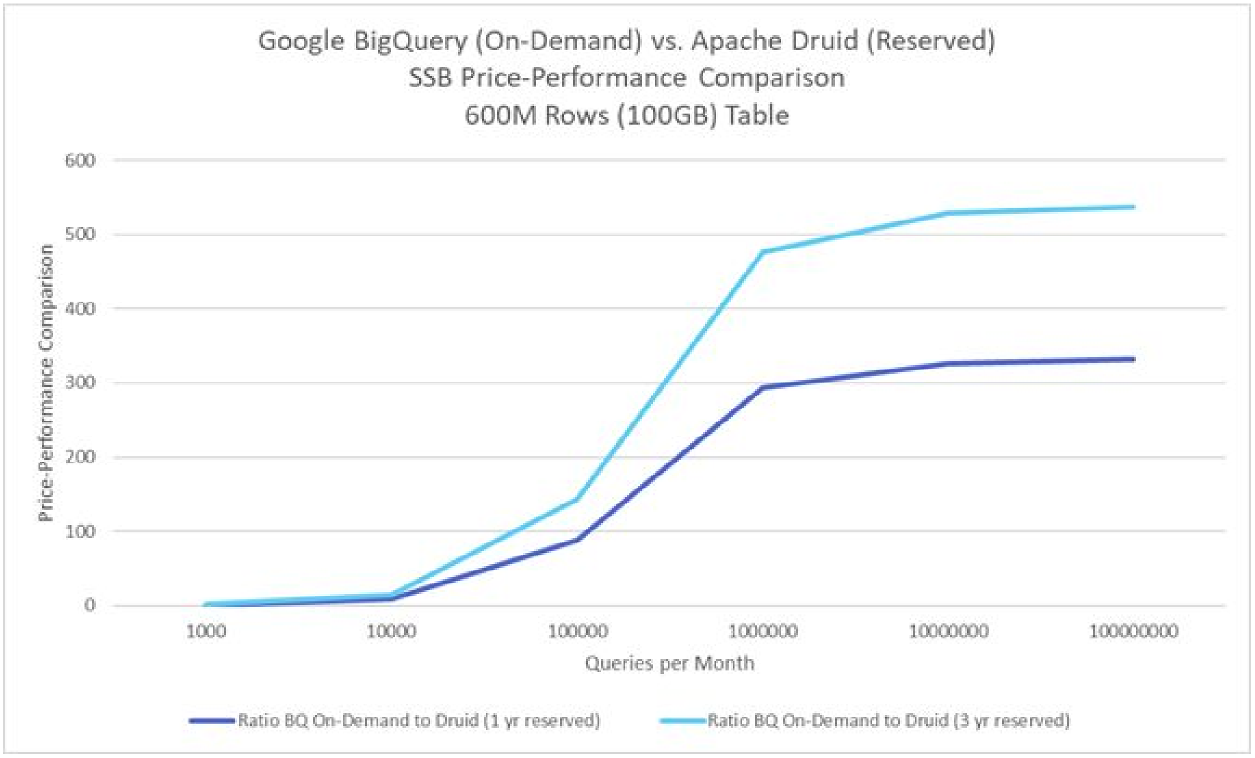


Figure 3. SSB Price-Performance Comparison: Google BigQuery (On-Demand) to Apache Druid (Reserved). Price-performance ratio is shown for increasing levels of queries per month.

## Testing methodology

We evaluated the performance of Imply Cloud (Apache Druid) and Google BigQuery to determine the suitability of each as an enterprise data warehouse (EDW) solution. Testing involved five major steps for each solution:

1. Provision each solution
2. Generate and prepare SSB test data
3. Ingest SSB test data with optimal schema
4. Optimize SSB test queries
5. Performance test using Apache JMeter

We optimized schema and queries following documented best practices for each platform. The SSB papers are clear in their vagueness regarding optimization ground rules.

1. Columns in SSB tables can be compressed by whatever means available in the database system used, as long as reported data retrieved by queries has the values specified in the original schema.
2. Any product capability used in one product database design to improve performance must be matched in the database design for other products by an attempt to use the same type of capability, assuming such a capability exists and improves performance.
3. Materialized Views that pre-join some useful dimension columns with the lineorder table are permitted.
4. Denormalization is not only acceptable, it is considered a standard practice in data warehousing.
5. Queries are designed to select from the lineorder table exactly once (no self-joins or subqueries or table queries) to represent “classic” data warehouse queries of select from the fact table with restrictions on the data table attributes.
6. Queries are chosen as much as possible to span the tasks performed by an analyst, starting broadly and the increasing restrictions through the query group.
7. Variant query forms are allowed. Any alternative SQL form that modifies predicate restrictions but retains the same effect on retrieval is fine.
8. Query cache must be disabled, although OS and hardware level caching may not. Caching rules must be documented.
9. SSB reports must include a scale factor, database product name, version number, processor model and number of processors, memory space, disk setup, and all other parameters that may impact performance.
10. Any tuning capability habitually used to improve performance in a database product should be adopted for that product.
11. Published SSB reports can anonymize the products tested, removing product-specific tuning details and query plans

A logical testbed diagram follows. Apache Druid (Imply Cloud) and the JMeter server ran on AWS instances, while Google BigQuery was tested with JMeter running on GCP. Wherever possible, instances used for JMeter load generation were deployed into the same cloud provider region or zone to minimize network latency.

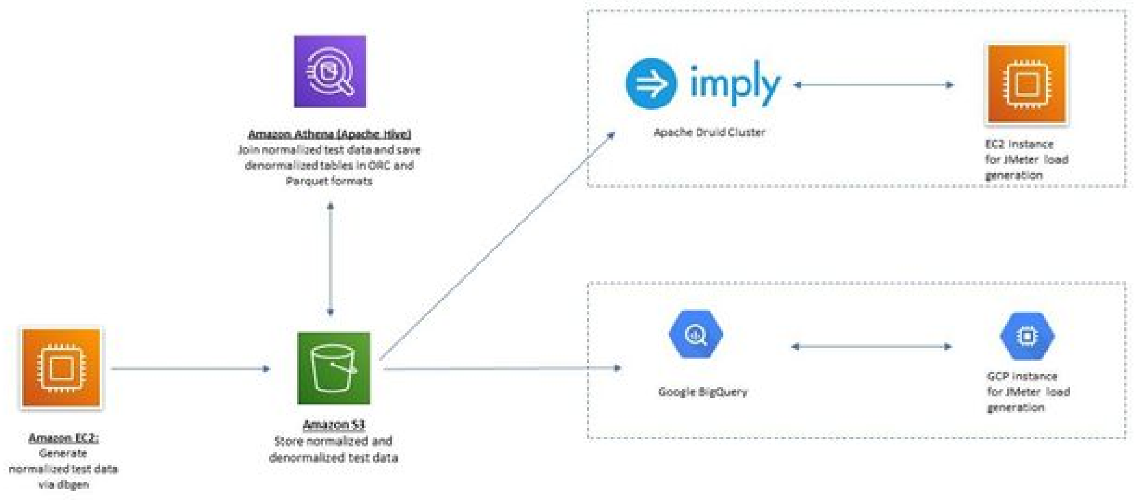

Amazon Athena (Apache Hive)
Join normalized test data and save denormalized tables in ORC and Parquet formats
imply
Apache Druid Cluster
EC2 Instance for JMeter load generation
Amazon EC2:
Generate normalized test data via dbgen
Amazon S3
Store normalized and denormalized test data
Google BigQuery
GCP Instance for JMeter load generation

Figure 4. Logical testbed diagram for SSB performance testing. Apache Druid was deployed in AWS.

We tested on Apache Druid unmodified from Imply Cloud. Google BigQuery is available as a service, making provisioning and configuration unnecessary. We followed published best practices to optimize for query response time throughout each stage of the test process.

## Apache Druid

Apache Druid is an open source distributed data store. Druid's core design combines ideas from data warehouses, time series databases, and search systems to create a unified system for real-time analytics for a broad range of use cases. Druid merges key characteristics of each of the 3 systems into its ingestion layer, storage format, querying layer, and core architecture.

- Testing was conducted on Imply Cloud using Imply version 3.3.0.1 (Apache Druid 0.18.1)
- The following hardware configurations were used:
    - Data Servers: 3 @ i3.2xlarge (8 vCPU, 61 GB memory, 1.9 TB NVMe SSD storage each)
    - Query Servers: 2 @ m5d.large (2 vCPU, 8 GB memory each)
    - Master Server: 1 @ m5.large (2 vCPU, 8 GB memory)

## Google BigQuery

Google BigQuery is a cloud-based, fully managed, serverless enterprise data warehouse that supports analytics over petabyte-scale data. It delivers high-speed analysis of large data sets as a service. BigQuery scales its use of hardware up or down to maximize performance of each query, adding and removing compute and storage resources as needed.

BigQuery is very easy to use via a simple browser-based GUI, although it lacks indexes or column constraints. No system tuning is possible. Google dynamically allocates storage and compute resources. Customers pay for the amount of data they query and store. Customers can pre-purchase flat-rate computation “slots” in increments of $10,000 per month per 500 computer units, or commit to a year for $8,500 per month.

Ingestion was not as smooth as we were led to believe it would be, although once configured, it was fast. Our data transfer job for the ORC test file required entering location on our S3 bucket as a path containing wildcards in a box that stated that wildcards are not valid input. Google claims that data with an unknown schema can be loaded automatically, but we could not get this feature to work.

## Apache Jmeter and instances

The Apache JMeter™ application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing web applications but has since expanded to other test functions. JMeter Version 5.2.1 was used for this testing.

JMeter was deployed as follows:

- To evaluate Druid performance, JMeter was installed and run from the command line of the primary Query server.
- To evaluate BigQuery performance, JMeter was deployed onto a e2-standard-4 (4 vCPU, 16GB memory) in the us-east1 zone.

# Data generation and preparation

We generated 600 million rows (approximately 100GB) of test data using SSB’s dbgen downloaded from https://github.com/lemire/StarSchemaBenchmark and executed locally. The test files generated included the fact table lineorder.tbl, and the dimension tables customer.tbl, part.tbl, supplier.tbl, and date.tbl. We executed dbgen with a Scale Factor of 1 (SF=1) to generate a lineorder table with 6,000,000 rows.

The schema of the SSB is based on TPC-H, as are the queries. Denormalization is standard in data warehousing and makes many joins unnecessary in common queries. Columns are classified as identifiers (any data type, but unique values for what it is identifying), text (fixed or variable length), and numeric (whole numbers, not floating point.)  Numeric identifiers must have unique values and have numeric interpretations which provide unique numbers. Text is in 8-bit ASCII.

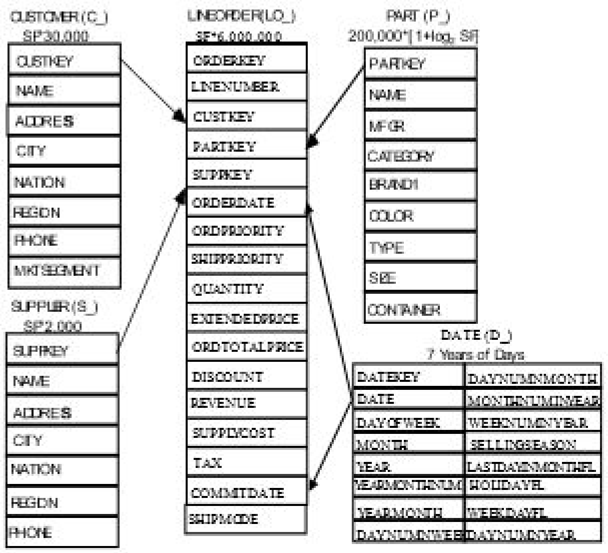


Figure 5. SSB Schema

We denormalized the SSB data to create a single flat file using Amazon Athena (Hive). We partitioned the denormalized data by month. Athena DML is contained in a Github repository.

We saved the denormalized data to an S3 bucket in ORC and parquet formats.

## Data ingestion

Prior to querying data, we had to create tables and ingest the denormalized SSB data. We created tables with the foreknowledge that dbgen generates patterns of data, especially so with respect to dates. Data falls roughly into equal amounts of rows per day and month. This worked out conveniently when planning for optimal table structure and queries. We partitioned on month in the source data and saved it in ORC for Druid and BigQuery.

In Druid's case, we installed the druid-ORC extension and then ingested partitioned source data in ORC format. Data was partitioned on month and ingested by parsing lo_orderdate into 2,607 segments with an average segment size of 75.07 MB and a total size of 195.70 GB. See this Github repository for Druid ingestion and tuning specs.

In BigQuery we performed the following steps:

1. Create a table
2. Run a data transfer job on our ORC test data from S3 and insert
3. Create another table with an additional column *f0_* so data can be partitioned by date

See this Github repository for BigQuery DDL.

## Query optimization

SSB's classic data warehouse queries select from the line order table exactly once (no self-joins or subqueries), with predicate restrictions on dimension table attributes. Benchmark queries are intended to span the tasks performed by common Star Schema queries used in commercial data warehouse systems.

Both platforms tested support SQL. We followed the same general procedure to optimize queries for each platform, starting with the published generic SSB SQL queries and optimizing them to make use of platform-specific optimizations.

Apache Druid supports queries via Druid SQL and native queries. Druid SQL is a built-in SQL layer that is powered by a parser and planner based on Apache Calcite. Druid SQL translates SQL into native Druid queries on the Query server, which are then executed. While there is a theoretical overhead involved in translating SQL, it was negligible in our testing. We further found that optimized SQL queries execute faster than unoptimized native queries.

Google BigQuery leverages standard SQL that complies with the SQL 2011 standard. Prior to version 2.0, BigQuery ran "BigQuery SQL", a non-standard SQL dialect that was renamed to legacy SQL. Standard SQL allowed us to utilize subqueries in the SELECT and WHERE clauses.

We leveraged platform-specific syntax for date expressions in SSB queries. This enabled us to limit queries across partitions and to filter on time using the platform's strengths to reduce the number of columns and rows that needed to be scanned. We also applied general SQL query optimization tactics such as simplifying expressions to aid in query planning.

For example, here is how this procedure played out in optimizing Query 4.3.

| **Query Optimization Stage** | **Query 4.3** |
|---|---|
| SSB (Original) | select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit from denormalized where s_nation = 'UNITED STATES' and (d_year = 1997 or d_year = 1998) and p_category = 'MFGR#14' group by d_year, s_city, p_brand1 order by d_year, s_city, p_brand1 |
| Apache Druid | select d_year, s_nation, p_category, sum(lo_revenue) - sum(lo_supplycost) as profit from ${jmDataSource} where c_region = 'AMERICA' and s_region = 'AMERICA' and (FLOOR(\"__time\" to YEAR) = TIME_PARSE('1997-01-01T00:00:00.000Z') or FLOOR(\"__time\" to YEAR) = TIME_PARSE('1998-01-01T00:00:00.000Z')) and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, s_nation, p_category order by d_year, s_nation, p_category |

| | |
|---|---|
| Google BigQuery | select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit<br><br>from \`community-benchmark.SSBData.ssb_data_small_part\`<br><br>where s_nation = 'UNITED STATES'<br><br>and (DATE(f0_) BETWEEN "1997-01-01" AND "1998-12-31")<br><br>and p_category = 'MFGR#14'<br><br>group by d_year, s_city, p_brand1<br><br>order by d_year, s_city, p_brand1 |

Figure 6. Query Optimization Process

A detailed discussion of SSB queries is available in Appendix A and a Github repository.

## Performance testing

We used JMeter to assess single-user query performance. No multi-user testing was performed. A JMeter script can be found in a Github repository.

We ran JMeter against each platform's HTTP API under the following conditions:

- Query cache off
- Each SSB query was run 10 times (10 samples per query)
- Each query flight consisted of all 13 SSB queries run in succession
- For each test, Average Response Time, Lowest Response Time, Highest Response Time, and Average Response Time Standard Deviation per query were calculated
- Each test was repeated five times
- The lowest and highest test results were discarded, a standard practice to remove outliers from performance testing results, leaving results from 3 test runs

- The remaining 3 results for each query were averaged to provide results for Average Response Time, Lowest Response Time, Highest Response Time, and Average Response Time Standard Deviation per query were calculated

## Personnel

The following people contributed to this research:

| Community Team | Professional Services Team |
|---|---|
| Matt Sarrel<br><br>Surekha Saharan<br><br>Gian Merlino<br><br>Rachel Pedreschi | Ben Hopp<br><br>Rommel Garcia |

Figure 7. Contributors

# Test results

This section provides detailed results of performance testing and price-performance comparison of Apache Druid and Google BigQuery using the Star Schema Benchmark. Please refer to the Testing Methodology section (above) for complete details of how we conducted testing, and please see SSB Performance Test Results (below) for results.

## Computation and analytical parameters

To obtain the average query response results presented, we used JMeter to measure query response time via HTTP API.

### Price-performance comparison

The benchmark results tell us about performance of a specific workload, yet we also want to account for price, and we need to compare the cost of Apache Druid running on AWS instances to serverless BigQuery. Our overall methodology was to calculate the normalized cost of running the full SSB query flight for each platform. We summed the individual query times to determine an aggregate query flight time. We then set a performance threshold that stipulated that the total query flight must complete in less than 25 seconds. We relied on actual costs presented by the Google billing API as well as publicly available pricing for Google BigQuery and AWS in our calculations.

Cost for Apache Druid is based on AWS on-demand and reserved instance costs. We obtained query response time via JMeter and query CPU time via the platform interface. Cost for BigQuery is modeled for their on-demand and flat-rate monthly price plans. We used the query cost and slot time provided by the console to calculate the number of slots used and the slot cost for both on-demand and flat-rate billing. Pricing assumes the SSB workload runs 24/7.

We used our SLA-type requirement (query flight must complete in 25 seconds or less, selected to ensure BigQuery can complete the test flight in the given time period given that BigQuery took 19.4 seconds in our testing) to establish two pricing models, one based on flat-rate pricing and concurrency (number of users) per month and one based on on-demand pricing and number of queries per month. Our flat-rate comparisons will be helpful in estimating recurring monthly cost based on the number of concurrent users, while our on-demand comparisons will be helpful in estimating monthly cost based on the number of queries. We used both average query time and total query flight time as performance metrics. We then compared results from the two models to determine price-performance as a function of queries per month, concurrency, and price.

We relied on the following assumptions to begin calculations:

| **Minutes per year** | **Flight query count** | **Required flight time (seconds)** | **GBQ flat slot batch size** | **GBQ flat slot batch cost (1 yr commit)** | **Druid server on-demand cost** | **Druid server reserved cost (1yr not prepaid)** | **Druid server reserved cost (3yr prepaid)** | **Peak/avg ratio** |
|---|---|---|---|---|---|---|---|---|
| 525600 | 13 | 25 | 500 | $8,000 | $457 | $312 | $193 | 5 |

Figure 8. Parameters and assumptions used to model price-performance.

We performed the following calculations in order to model price-performance:

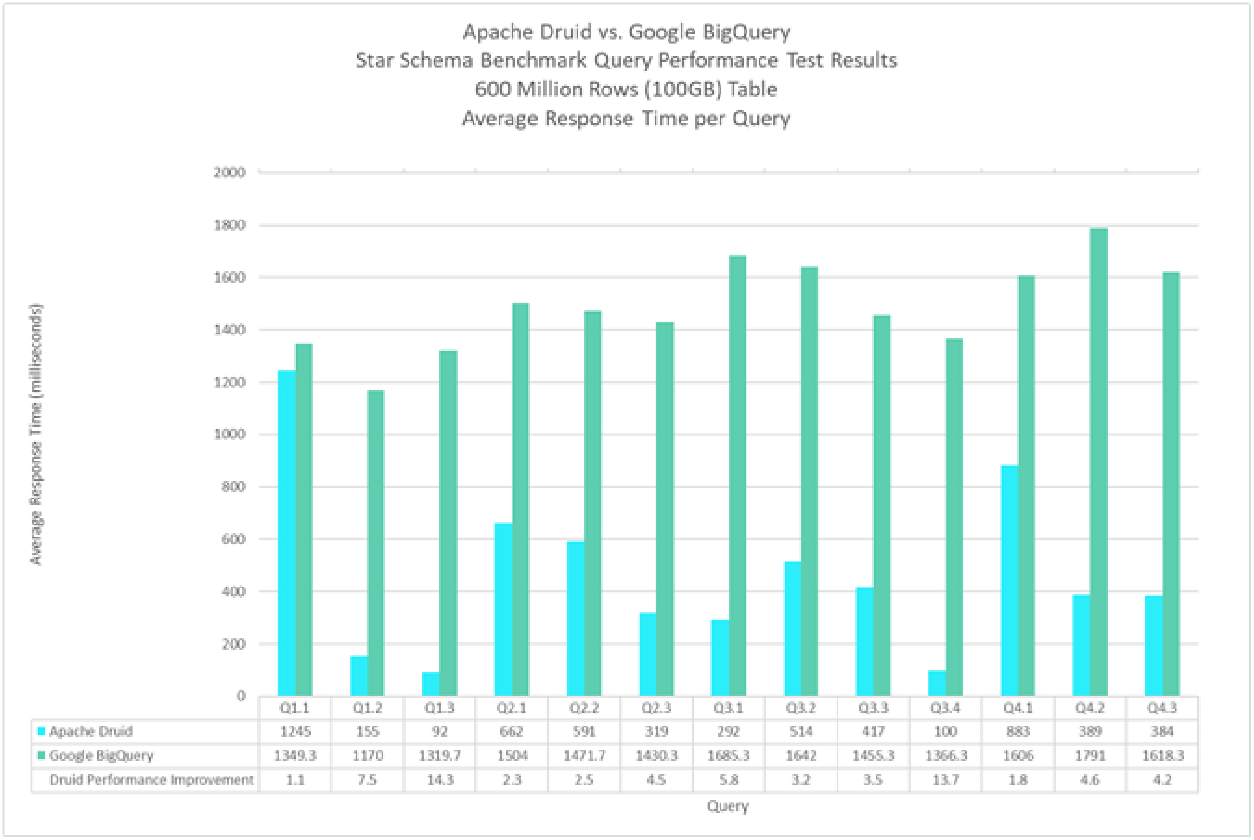
1. Start with average response time for Google BigQuery and Apache Druid for partitioned data with time filtered queries
2. Sum average response time for all SSB queries in the query flight

3. Cost calculated is based on bytes transferred for BigQuery and AWS instance time for Druid
4. Cost per query calculated as total cost / 13 (number of SSB queries)
5. BigQuery flat slot cost assumes 500 slots purchased for $8500K/month. On-demand and flat rate comparison allowed us to:
    1. Calculate time taken per slot
    2. Calculate slot cost
    3. Calculate cost per query
6. Assuming that performance scales linearly in column-oriented databases, we extrapolated performance and calculated price across a range of usage scenarios
7. We then calculated ratios of Druid to BigQuery for price to performance

## SSB performance test results

We ran the optimized SSB queries and recorded results using Apache JMeter to compare performance of Apache Druid and Google BigQuery. We submitted queries to both via the HTTP API to make it easier to compare performance results. We used on-demand BigQuery access to process queries via the HTTP API. **Please refer to the Testing Methodology section of this report for complete details on how we conducted these tests**.

The overall Star Schema Benchmark test results for Apache Druid and Google BigQuery, shown below, indicate that Apache Druid is the overall performance leader. In our test configurations, Apache Druid outperforms Google BigQuery by 321% on the Star Schema Benchmark. Total average response time for the query flight in Druid was 6043 ms, compared to 19409 ms in BigQuery. For these tests, lower average response times are better.

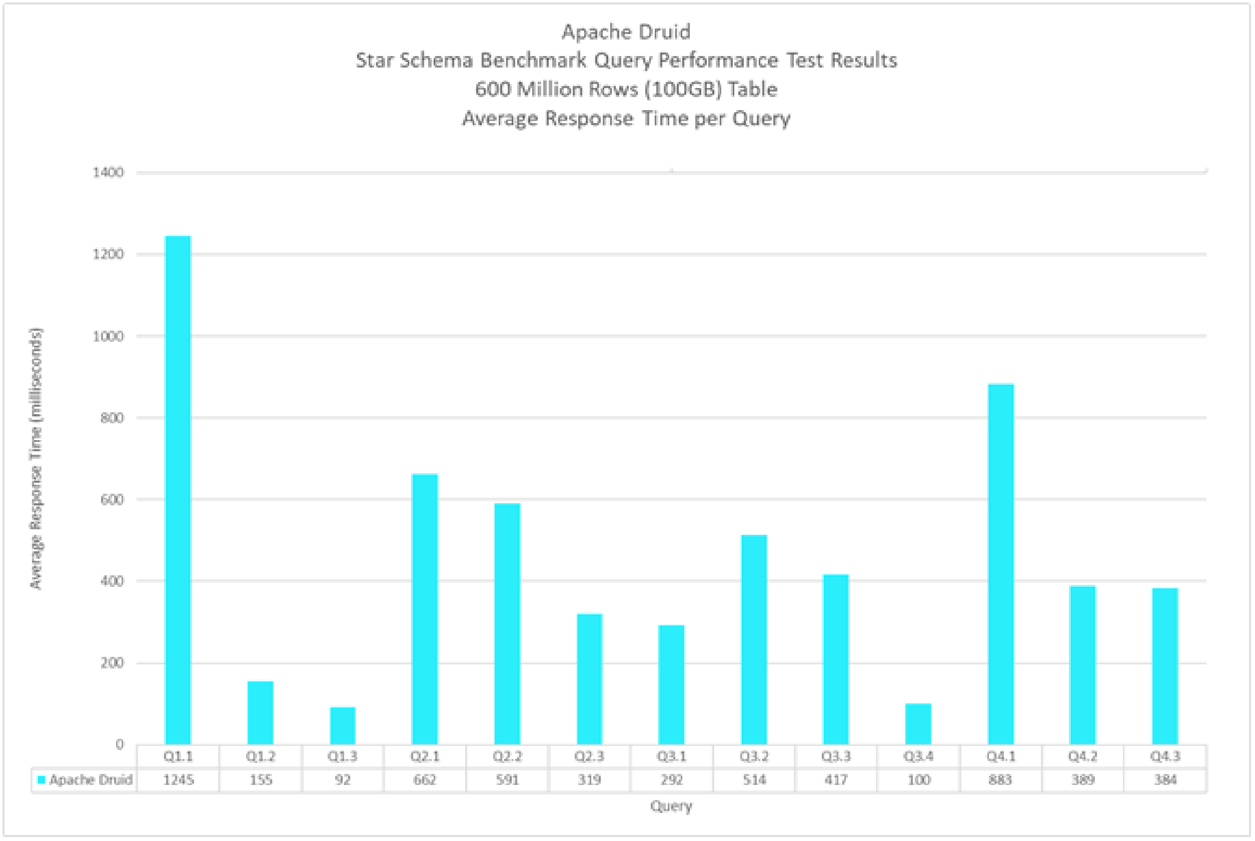


| | Q1.1 | Q1.2 | Q1.3 | Q2.1 | Q2.2 | Q2.3 | Q3.1 | Q3.2 | Q3.3 | Q3.4 | Q4.1 | Q4.2 | Q4.3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ■ Apache Druid | 1245 | 155 | 92 | 662 | 591 | 319 | 292 | 514 | 417 | 100 | 883 | 389 | 384 |
| ■ Google BigQuery | 1349.3 | 1170 | 1319.7 | 1504 | 1471.7 | 1430.3 | 1685.3 | 1642 | 1455.3 | 1366.3 | 1606 | 1791 | 1618.3 |
| Druid Performance Improvement | 1.1 | 7.5 | 14.3 | 2.3 | 2.5 | 4.5 | 5.8 | 3.2 | 3.5 | 13.7 | 1.8 | 4.6 | 4.2 |

Figure 9. Star Schema Benchmark (SSB) Query Performance for Apache Druid and Google BigQuery. Average response time (milliseconds) per query. Total query flight time: Apache Druid (6043 ms), Google BigQuery (19409 ms). Lower is better.

## Test results by solution

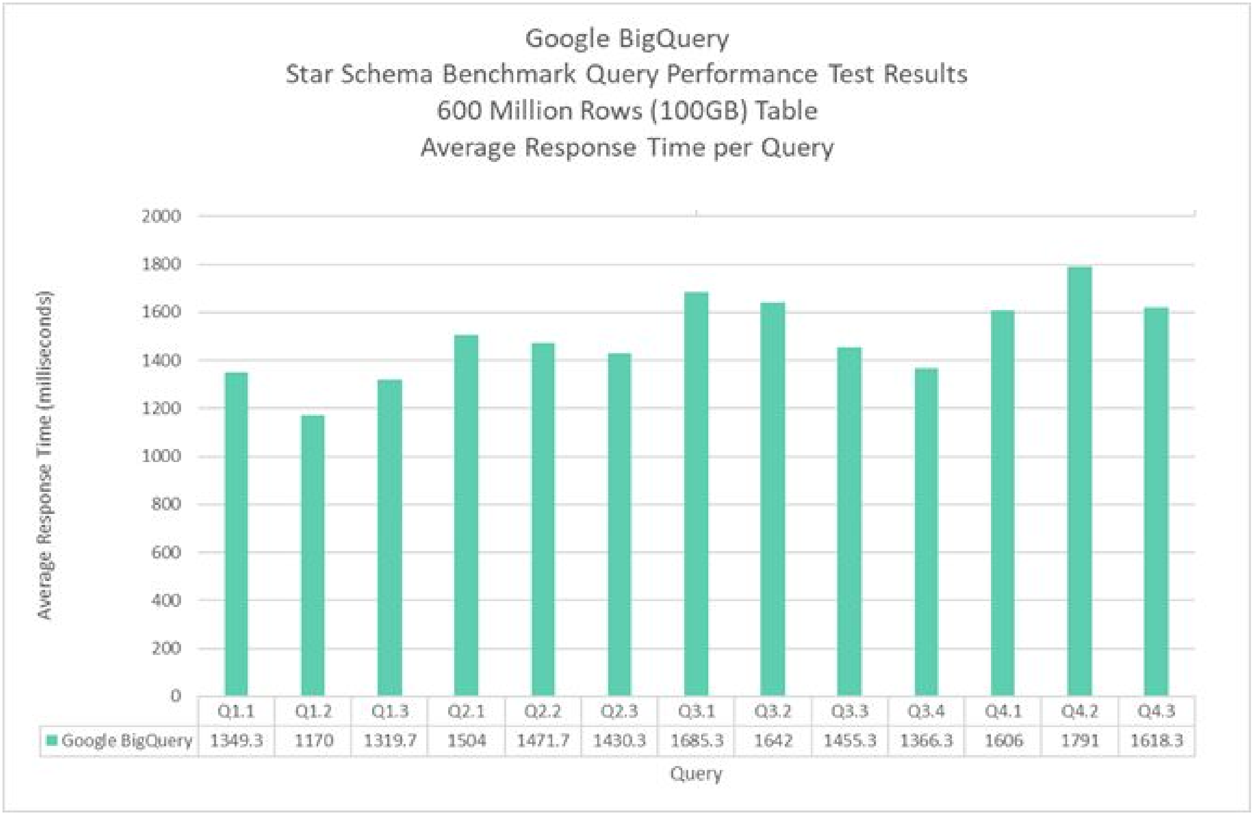
Apache Druid was the overall best performer on the 13 SSB queries, beating BigQuery in every query, in our testing. We can see that vectorization dramatically improves query response time.



| | Q1.1 | Q1.2 | Q1.3 | Q2.1 | Q2.2 | Q2.3 | Q3.1 | Q3.2 | Q3.3 | Q3.4 | Q4.1 | Q4.2 | Q4.3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ■ Apache Druid | 1245 | 155 | 92 | 662 | 591 | 319 | 292 | 514 | 417 | 100 | 883 | 389 | 384 |

Figure 10. SSB Query Average, Highest, Lowest Response Time and Vectorization: Apache Druid. Total query flight time for Apache Druid is 6043 ms. Lower is better.

Google BigQuery, outperformed by Druid on each query, exhibited consistent, although slow, query response times.



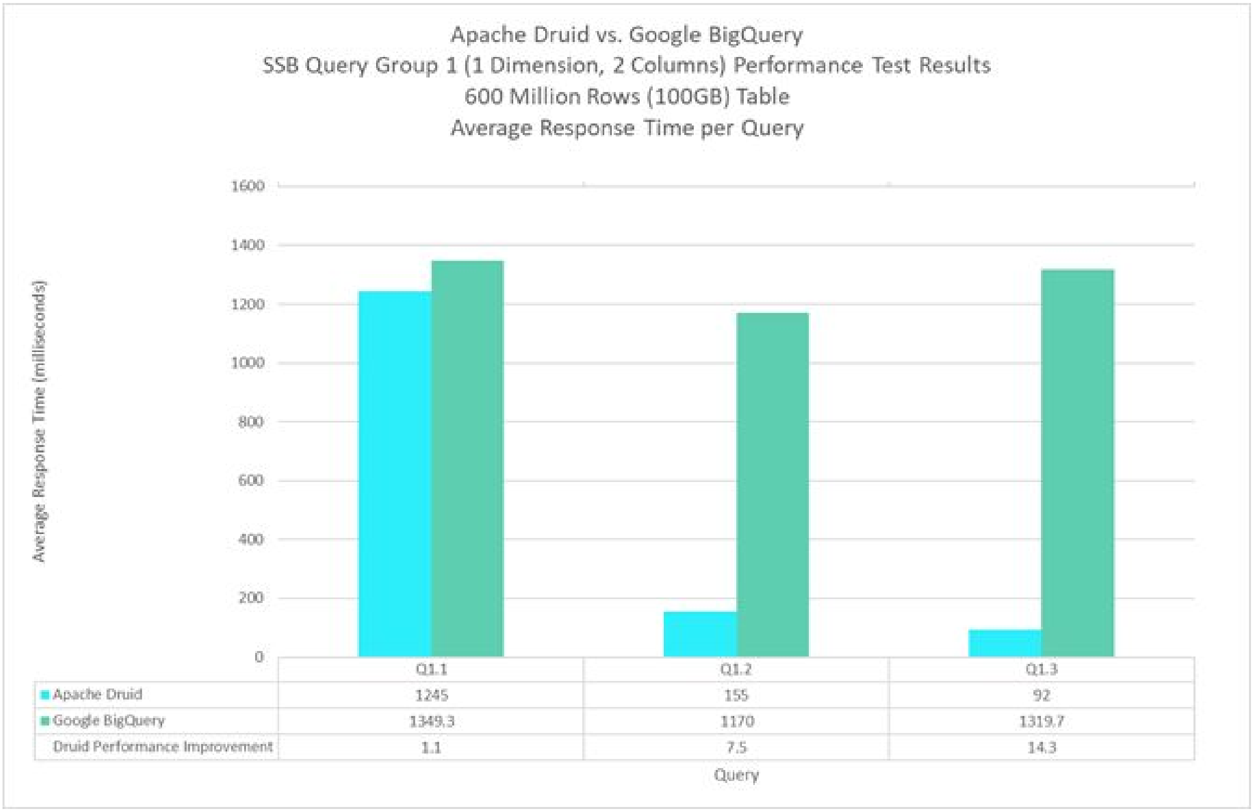
| | Q1.1 | Q1.2 | Q1.3 | Q2.1 | Q2.2 | Q2.3 | Q3.1 | Q3.2 | Q3.3 | Q3.4 | Q4.1 | Q4.2 | Q4.3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 🟩 Google BigQuery | 1349.3 | 1170 | 1319.7 | 1504 | 1471.7 | 1430.3 | 1685.3 | 1642 | 1455.3 | 1366.3 | 1606 | 1791 | 1618.3 |

Figure 11. SSB Query Average, Lowest, Highest Response Time: Google BigQuery. Total query flight time for Google BigQuery is 19409 ms. Lower is better.

## Results by query group

SSB contains a group of 13 queries organized into 4 groups. Each Query Group is meant to provide functional coverage so as to allow prospective users to derive a performance rating to match the query workload they expect to use in practice. Also, in general, the total number of fact table rows retrieved is determined by the selectivity of restrictions on dimensions. The selectivity of queries varies across each Query Group, as well as how results are grouped, aggregated and sorted.

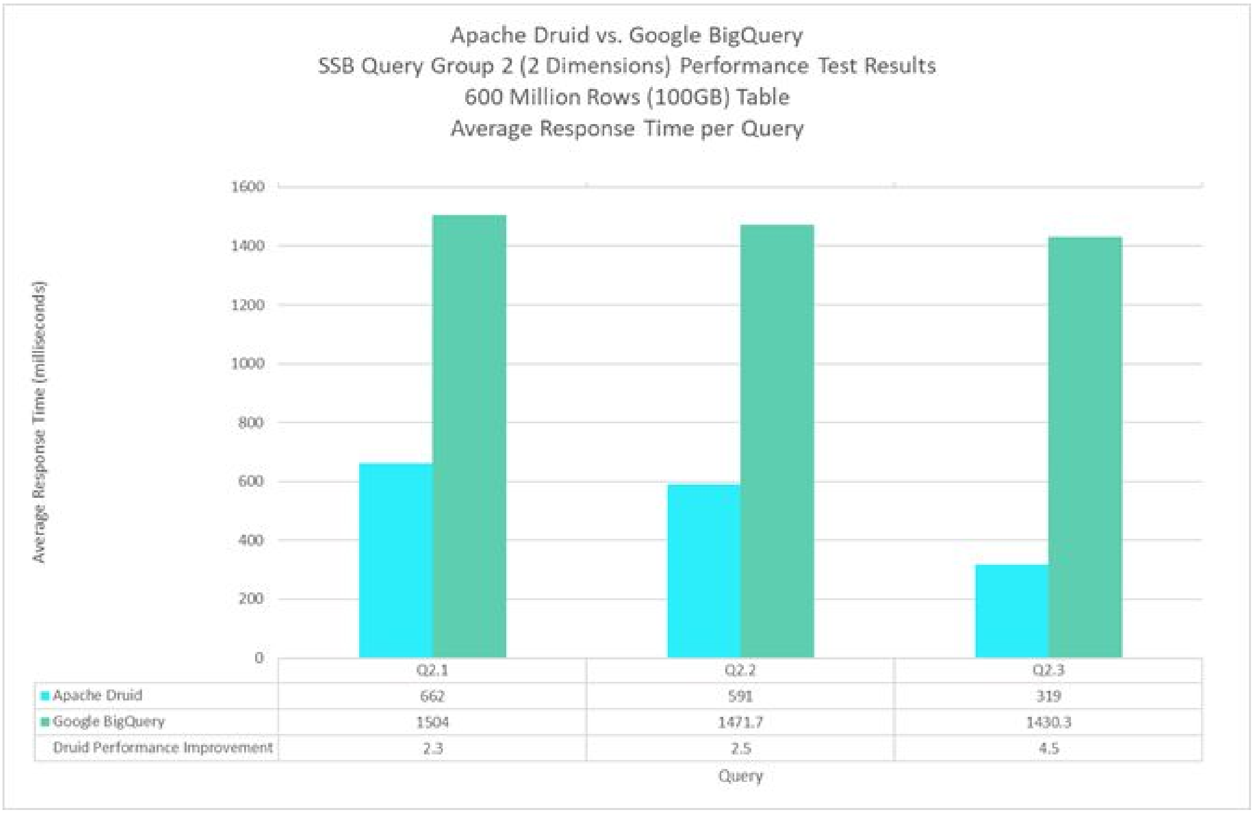
Query Group 1 (based on TPC-H TPCQ6) has a restriction on one dimension and two lineorder columns, lo_discount and lo_quantity. The query measures the revenue increase from eliminating various ranges of discounts in given product order quantity intervals in a given year. Druid outperformed BigQuery in this query group.



| | Q1.1 | Q1.2 | Q1.3 |
|---|---|---|---|
| 🟦 Apache Druid | 1245 | 155 | 92 |
| 🟩 Google BigQuery | 1349.3 | 1170 | 1319.7 |
| Druid Performance Improvement | 1.1 | 7.5 | 14.3 |

Figure 12. SSB Query Group 1 Average, Lowest, Highest Response Time: Apache Druid, Google BigQuery. Lower is better.

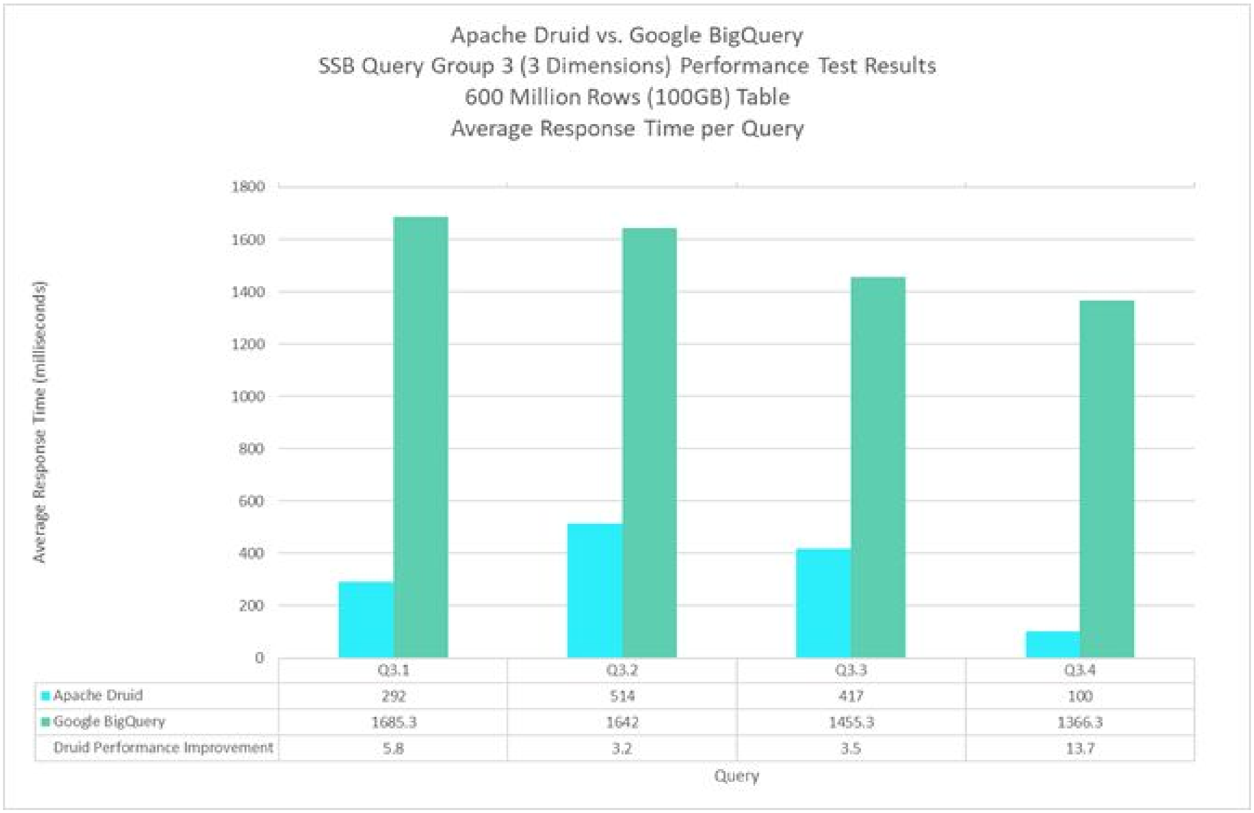
Query Group 2 has restrictions on two dimensions. The query compares revenues for certain product classes and suppliers in a given region, grouped by more restrictive product classes and all years of orders. Druid outperformed BigQuery.



| | Q2.1 | Q2.2 | Q2.3 |
|---|---|---|---|
| Apache Druid | 662 | 591 | 319 |
| Google BigQuery | 1504 | 1471.7 | 1430.3 |
| Druid Performance Improvement | 2.3 | 2.5 | 4.5 |

Figure 13. SSB Query Group 2 Average, Lowest, Highest Response Time: Apache Druid, Google BigQuery. Lower is better.

Query Group 3, based on TPC-H query TPCQ5, has restrictions on three dimensions. The query retrieves total revenue for lineorder transactions within a given region in a certain time period, grouped by customer nation, supplier nation and year. These queries progressively restrict to smaller numbers of results, therefore the expectation is that Q3.4 will execute fastest for a given product.  Druid dramatically outperformed BigQuery when faced with growing query complexity.

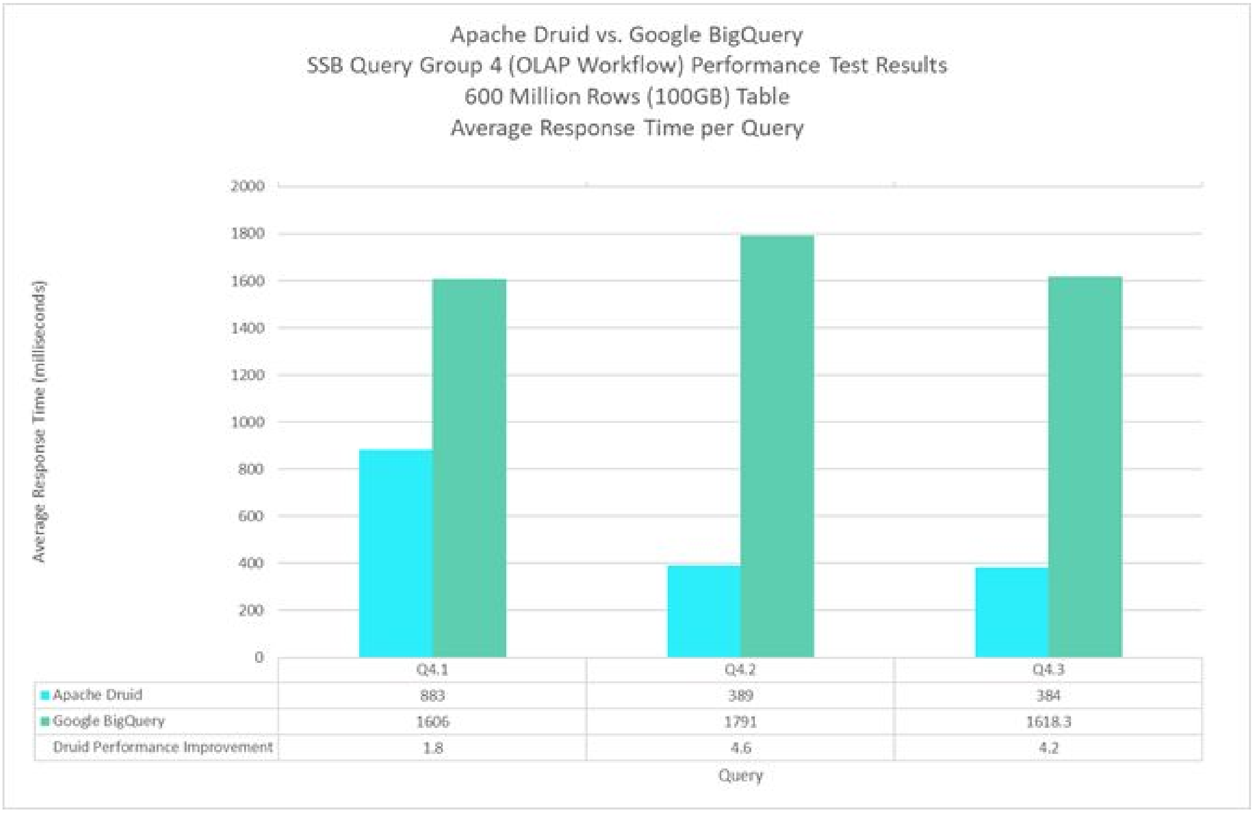


| | Q3.1 | Q3.2 | Q3.3 | Q3.4 |
|---|---|---|---|---|
| ■ Apache Druid | 292 | 514 | 417 | 100 |
| ■ Google BigQuery | 1685.3 | 1642 | 1455.3 | 1366.3 |
| Druid Performance Improvement | 5.8 | 3.2 | 3.5 | 13.7 |

Figure 14. SSB Query Group 3 Average Response Time: Apache Druid and Google BigQuery. Lower is better.

SSB Query Group 4 provides a “What-If” sequence of queries that are representative of an OLAP style of drill down exploration. It starts with a query with weak constraints on three-dimensional columns, retrieves aggregate profit, sum(lo_revenue - lo_supplycost), groups by d_year and c_nation. Following queries modify predicate constraints by drilling down to locate the origin of an anomaly. Q4.1 shows a growth in profit. Q4.2 drills down to group by p_category to see where the profit change came from. Q4.3 restricts s_nation to ‘UNITED STATES’ and p_category = ‘MFGR#14’, and drills down to group by s_city and p_brand.

Druid outperforms BigQuery on every query in Query Group 4, demonstrating superior performance on an OLAP-style EDW workload. BigQuery lags significantly throughout this query group, suggesting that the new class of cloud EDW typified by Druid excels at OLAP-style workloads.



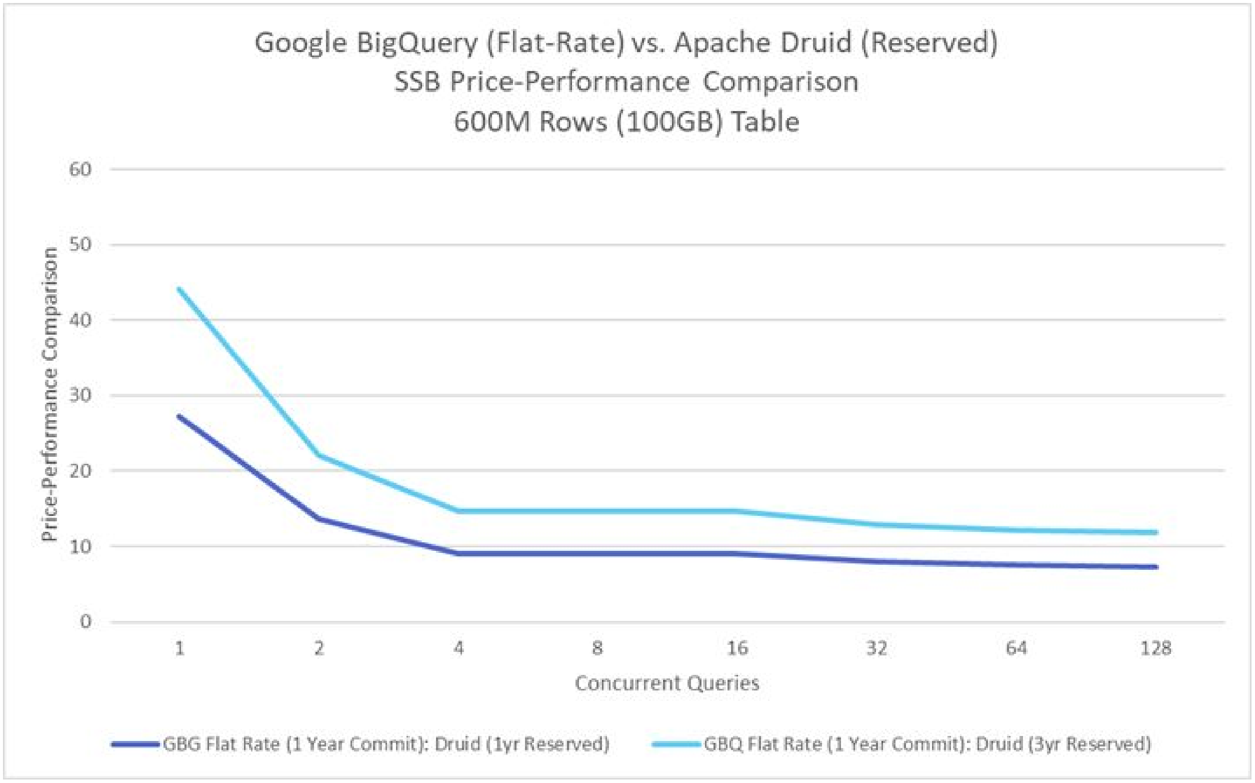
| | Q4.1 | Q4.2 | Q4.3 |
|---|---|---|---|
| ■ Apache Druid | 883 | 389 | 384 |
| ■ Google BigQuery | 1606 | 1791 | 1618.3 |
| Druid Performance Improvement | 1.8 | 4.6 | 4.2 |

Figure 14. SSB Query Group 4 Average Response Time: Apache Druid, Google BigQuery. Lower is better.

## Price performance comparison

Based on the detailed analysis below, we provide the conservative estimate that Druid exhibits a 12x price performance advantage over BigQuery. In our testing with the SSB workload, we found that Druid outperformed BigQuery by three times (Total Query Flight Time) at a much lower cost. Calculations below demonstrate a conservative estimation that Druid has a twelve times price performance advantage over BigQuery for the SSB workload run on an enterprise scale, as shown in concurrent queries, over a month. Concurrent queries are defined as those actively running at one time against an EDW. Concurrent queries come from many sources and have different profiles, so we modelled across a range of results. For example, dashboards can launch and refresh eight or more SQL queries that execute concurrently, or multiple users walking through multiple what-if query paths would represent concurrent queries.

Most enterprises will focus on BigQuery's flat-rate pricing with a one-year commit for budgeting, so we emphasize that comparison to Druid. When normalizing performance, BigQuery costs between seven and 44 times as much for the same performance as Druid on a monthly basis depending on AWS reserved instance cost.



Apache Druid (Reserved). Google BigQuery Flat-Rate pricing with one year commit is shown against Apache Druid pricing on reserved AWS instances with one year and three year commit. Price-performance ratio is shown for increasing levels of concurrent queries.

The above model is based on concurrent queries, while the below model is based on total number of queries per month. Concurrent queries can be used as an indication of flat-rate monthly cost, while queries per month indicates how BigQuery on-demand cost scales with respect to the cost of an equivalently performing Druid cluster. Google charges by bytes transferred in the query response, so cost goes up linearly with the number of queries. Druid requires servers added to the cluster to handle the load from additional queries, so cost increases with the number of AWS instances required to meet that load.

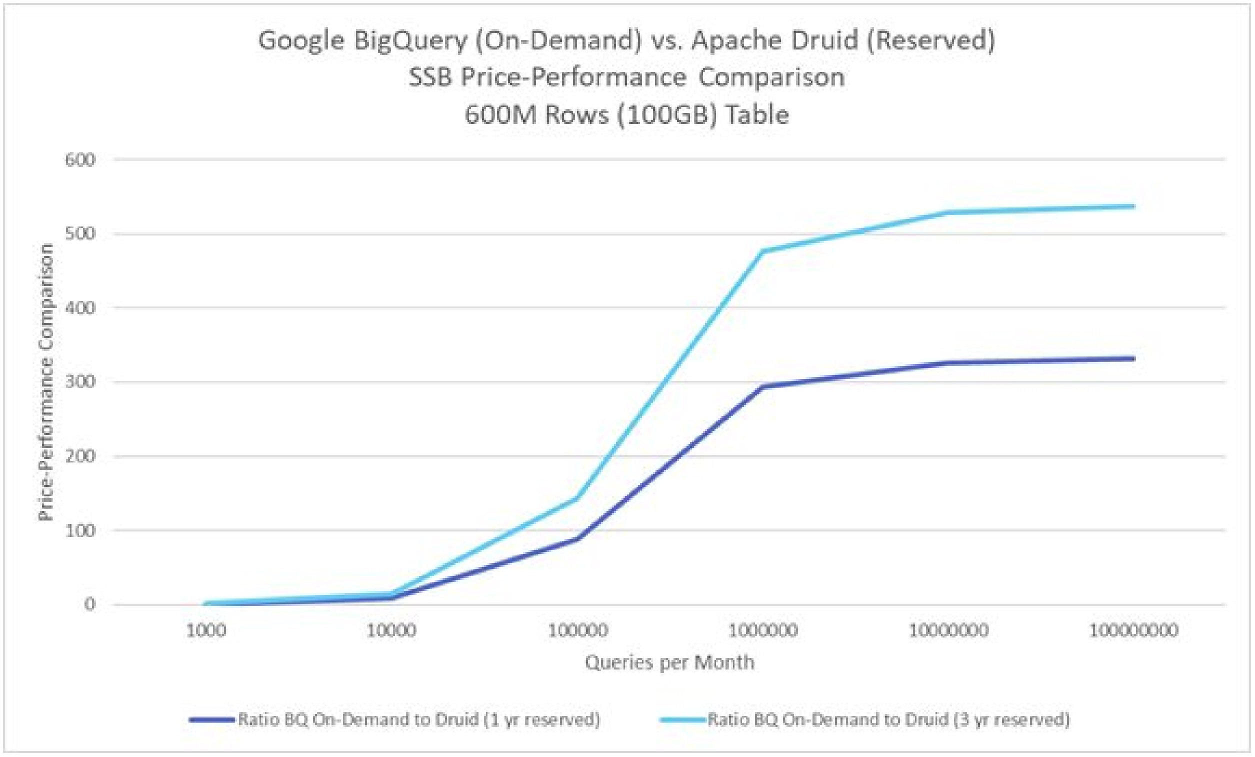


Figure 16. SSB Price-Performance Comparison: Google BigQuery (On-Demand) to Apache Druid (Reserved). Price-performance ratio is shown for increasing levels of queries per month.

When modelling based on concurrency, we used BigQuery slot time as the basis for flat-rate pricing. We assumed a linear increase in slot time required to meet additional concurrency. For Druid, we used the price of AWS instances that we projected would be required to meet the same query response time threshold. Druid's Price Performance is 7 times better when using 1 year reserved pricing, 12 times better when using 3 year reserved pricing, than BigQuery Flat Rate. Druid’s cost can be minimized with 3 year reserved and prepaid AWS instances.

| Conc. Queries | BigQuery slot time | BigQuery slots | BigQuery monthly cost | BigQuery Flat Rate cost, rounded, 1yr commit | Druid server time | Druid servers, rounded | Druid monthly cost (1 yr not prepaid) | Druid monthly cost (3 yr prepaid) | Ratio: BigQuery Flat-Rate to Druid (1 yr not prepaid) | Ratio: BigQuery Flat Rate to Druid (3 yr prepaid) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2391.626 | 96 | $1,626 | $8,500 | 18.128 | 1 | $312 | $193 | 27.205 | 44.110 |
| 2 | 4783.252 | 191 | $3,253 | $8,500 | 36.256 | 2 | $625 | $385 | 13.603 | 22.055 |
| 4 | 9566.504 | 383 | $6,505 | $8,500 | 72.512 | 3 | $937 | $578 | 9.068 | 14.703 |
| 8 | 19133.008 | 765 | $13,010 | $17,000 | 145.024 | 6 | $1,875 | $1,156 | 9.068 | 14.703 |
| 16 | 38266.016 | 1531 | $26,021 | $34,000 | 290.048 | 12 | $3,749 | $2,312 | 9.068 | 14.703 |
| 32 | 76532.032 | 3061 | $52,042 | $59,500 | 580.096 | 24 | $7,499 | $4,625 | 7.935 | 12.865 |
| 64 | 153064.064 | 6123 | $104,084 | $110,500 | 1160.192 | 47 | $14,685 | $9,057 | 7.525 | 12.201 |
| 128 | 306128.128 | 12245 | $208,167 | $212,500 | 2320.384 | 93 | $29,057 | $17,921 | 7.313 | 11.858 |

Figure 17: Druid vs. BigQuery Price Performance: Flat-Rate Pricing with 1 Year Commit Per Month, Varying Concurrency. Druid's Price Performance is 7 times better when using 1 year reserved pricing, 12 times better when using 3 year reserved pricing, than BigQuery Flat Rate

Google limits the maximum concurrent number of slots and queries that are available concurrently on-demand. Results are shown to provide comparison to Druid.

As mentioned previously, the above model is based on concurrent queries, while the below model is based on total number of queries per month. Concurrent queries can be used as an indication of flat-rate monthly cost, while queries per month indicates how BigQuery on-demand cost scales with respect to the cost of an equivalently performing Druid cluster. Google charges by bytes transferred in the query response, so cost goes up linearly with the number of queries. Druid requires servers added to the cluster to handle the load from additional queries, so cost increases with the number of AWS instances required to meet that load.

| Queries / month | BigQuery On-Demand cost / month | Queries / peak time | Flights / peak time | Druid server time / peak time | Druid servers | Druid servers, rounded | Druid monthly cost (1 yr not prepaid) | Druid monthly cost (3 yr prepaid) | Ratio BigQuery On-Demand to Druid (1 yr not prepaid) | Ratio BigQuery On-Demand to Druid (3 yr prepaid) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1,000 | $275 | 0.0476 | 0.0037 | 0.0663 | 0.0027 | 1 | $312 | $193 | 0.8814 | 1.4291 |
| 10,000 | $2,754 | 0.4756 | 0.0366 | 0.6633 | 0.0265 | 1 | $312 | $193 | 8.8140 | 14.2908 |
| 100,000 | $27,538 | 4.7565 | 0.3659 | 6.6327 | 0.2653 | 1 | $312 | $193 | 88.1400 | 142.9085 |
| 1,000,000 | $275,385 | 47.5647 | 3.6588 | 66.3271 | 2.6531 | 3 | $937 | $578 | 293.8000 | 476.3616 |
| 10,000,000 | $2,753,846 | 475.6469 | 36.5882 | 663.2713 | 26.5309 | 27 | $8,436 | $5,203 | 326.4444 | 529.2906 |
| 100,000,000 | $27,538,462 | 4756.4688 | 365.8822 | 6632.7128 | 265.3085 | 266 | $83,109 | $51,258 | 331.3534 | 537.2499 |

Figure 18: Druid vs. BigQuery Price Performance: On-Demand Pricing Per Month, Varying Number Queries. Druid's Price Performance is 0.9 to 537 times better than BigQuery On-Demand.

# Appendix a

Below are the queries that were used in testing. The SQL queries as published in the Star Schema Benchmark are first, followed by the plain-English versions, then followed by denormalized versions, and then followed by queries optimized for Apache Druid and Google BigQuery.

## Star schema benchmark queries (original)

Query 1.1

```
select sum(lo_extendedprice*lo_discount) as revenue

from ssb.lineorder, ssb.dwdate

where lo_orderdate = d_datekey

and d_year = 1993

and lo_discount between 1 and 3

and lo_quantity < 25;
```

Query 1.2

```
select sum(lo_extendedprice*lo_discount) as revenue

from ssb.lineorder, ssb.dwdate

where lo_orderdate = d_datekey

and d_yearmonthnum = 199401

and lo_discount between 4 and 6

and lo_quantity between 26 and 35;
```

Query 1.3

```
select sum(lo_extendedprice*lo_discount) as revenue

from ssb.lineorder, ssb.dwdate

where lo_orderdate = d_datekey

and d_weeknuminyear = 6

and d_year = 1994

and lo_discount between 5 and 7

and lo_quantity between 26 and 35;
```

Query 2.1

```
select sum(lo_revenue), d_year, p_brand1

from ssb.lineorder, ssb.dwdate, ssb.part, ssb.supplier

where lo_orderdate = d_datekey

and lo_partkey = p_partkey

and lo_suppkey = s_suppkey

and p_category = 'MFGR#12'

and s_region = 'AMERICA'

group by d_year, p_brand1

order by d_year, p_brand1;
```

Query 2.2

```
select sum(lo_revenue), d_year, p_brand1

from ssb.lineorder, ssb.dwdate, ssb.part, ssb.supplier
```

where lo_orderdate = d_datekey

and lo_partkey = p_partkey

and lo_suppkey = s_suppkey

and p_brand1 between 'MFGR#2221' and 'MFGR#2228'

and s_region = 'ASIA'

group by d_year, p_brand1

order by d_year, p_brand1;

Query 2.3

select sum(lo_revenue), d_year, p_brand1

from ssb.lineorder, ssb.dwdate, ssb.part, ssb.supplier

where lo_orderdate = d_datekey

and lo_partkey = p_partkey

and lo_suppkey = s_suppkey

and p_brand1 = 'MFGR#2221'

and s_region = 'EUROPE'

group by d_year, p_brand1

order by d_year, p_brand1;

Query 3.1

select c_nation, s_nation, d_year, sum(lo_revenue) as revenue

from ssb.customer, ssb.lineorder, ssb.supplier, ssb.dwdate

where lo_custkey = c_custkey

and lo_suppkey = s_suppkey

```
and lo_orderdate = d_datekey

and c_region = 'ASIA' and s_region = 'ASIA'

and d_year >= 1992 and d_year <= 1997

group by c_nation, s_nation, d_year

order by d_year asc, revenue desc;
```

Query 3.2

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue

from ssb.customer, ssb.lineorder, ssb.supplier, ssb.dwdate

where lo_custkey = c_custkey

and lo_suppkey = s_suppkey

and lo_orderdate = d_datekey

and c_nation = 'UNITED STATES'

and s_nation = 'UNITED STATES'

and d_year >= 1992 and d_year <= 1997

group by c_city, s_city, d_year

order by d_year asc, revenue desc;
```

Query 3.3

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue

from ssb.customer, ssb.lineorder, ssb.supplier, ssb.dwdate

where lo_custkey = c_custkey

and lo_suppkey = s_suppkey

and lo_orderdate = d_datekey
and (c_city='UNITED KI1' or c_city='UNITED KI5')

and (s_city='UNITED KI1' or s_city='UNITED KI5')

and d_year >= 1992 and d_year <= 1997

group by c_city, s_city, d_year

order by d_year asc, revenue desc;
```

Query 3.4

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue

from ssb.customer, ssb.lineorder, ssb.supplier, ssb.dwdate

where lo_custkey = c_custkey

and lo_suppkey = s_suppkey

and lo_orderdate = d_datekey

and (c_city='UNITED KI1' or c_city='UNITED KI5')

and (s_city='UNITED KI1' or s_city='UNITED KI5')

and d_yearmonth = 'Dec1997'

group by c_city, s_city, d_year

order by d_year asc, revenue desc;
```

Query 4.1

```
select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit

from ssb.dwdate, ssb.customer, ssb.supplier, ssb.part, ssb.lineorder

where lo_custkey = c_custkey

 and lo_suppkey = s_suppkey

 and lo_partkey = p_partkey
 and lo_orderdate = d_datekey

 and c_region = 'AMERICA'

 and s_region = 'AMERICA'

 and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')

group by d_year, c_nation

order by d_year, c_nation;
```

Query 4.2

```
select d_year, s_nation, p_category, sum(lo_revenue - lo_supplycost) as profit

from ssb.dwdate, ssb.customer, ssb.supplier, ssb.part, ssb.lineorder

where lo_custkey = c_custkey

and lo_suppkey = s_suppkey

and lo_partkey = p_partkey

and lo_orderdate = d_datekey

and c_region = 'AMERICA'

and s_region = 'AMERICA'

and (d_year = 1997 or d_year = 1998)

and (p_mfgr = 'MFGR#1'

or p_mfgr = 'MFGR#2')

group by d_year, s_nation, p_category order by d_year, s_nation, p_category;
```

Query 4.3

```
select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit

from ssb.dwdate, ssb.customer, ssb.supplier, ssb.part, ssb.lineorder
```

```
where lo_custkey = c_custkey

and lo_suppkey = s_suppkey

and lo_partkey = p_partkey

and lo_orderdate = d_datekey

and c_region = 'AMERICA'

and s_nation = 'UNITED STATES'

and (d_year = 1997 or d_year = 1998)

and p_category = 'MFGR#14'

group by d_year, s_city, p_brand1 order by d_year, s_city, p_brand1;
```

## Star schema benchmark queries (plain-English)

Query Flight 1 has restrictions on 1 dimension and measures revenue increase from eliminating ranges of discounts in given product order quantity intervals shipped in a given year.

- Q1.1 has restrictions d_year = 1993, lo_quantity < 25, and lo_discount between 1 and 3.
- Q1.2 changes restrictions of Q1.1 to d_yearmonthnum = 199401, lo_quantity between 26 and 35, lo_discount between 4 and 6.
- Q1.3 changes the restrictions to d_weeknuminyear = 6 and d_year= 1994, lo_quantity between 36 and 40, and lo_discount between 5 and 7

Query flight 2 has restictions on 2 dimensions. The query compares revenues for certain product classes and suppliers in a certain region, grouped by more restrictive product classes and all years of orders.

- 2.1 has restrictions on p_category and s_region.
- 2.2 changes restrictions of Q2.1 to p_brand1  between 'MFGR#2221' and 'MFGR#2228' and s_regrion to 'ASIA'
- 2.3 changes restriction to p_brand1='MFGR#2339' and s_region='EUROPE'

Query flight 3, has restrictions on 3 dimensions. The query is intended to retrieve total revenue for lineorder transactions within and given region in a certain time period, grouped by customer nation, supplier nation and year.

- Q3.1 has restriction c_region = 'ASIA', s_region='ASIA', and restricts d_year to a 6-year period, grouped by c_nation, s_nation and d_year

- 3.2 changes region restrictions to c_nation = ""UNITED STATES' and s_nation = 'UNITED STATES', grouping revenue by customer city, supplier city and year.
- 3.3 changes restrictions to c_city and s_city to two cities in 'UNITED KINGDOM' and retrieves revenue grouped by c_city, s_city, d_year.
- 3.4 changes date restriction to a single month. After partitioning the 12 billion row dataset on d_yearmonth, we needed to rewrite the query for d_yearmonthnum

Query flight 4 provides a ""what-if"" sequence of queries that might be generated in an OLAP-style of exploration. Starting with a query with rather weak constraints on three dimensional columns, we retreive aggregate profit, sum(lo_revenue-lo_supplycost), grouped by d_year and c_nation. Successive queries modify predicate constraints by drilling down to find the source of an anomaly.

- Q4.1 restricts c_region and s_region both to 'AMERICA', and p_mfgr to one of two possilities.
- Q4.2 utilizes a typical workflow to dig deeper into the results. We pivot away from grouping by s_nation, restrict d_year to 1997 and 1998, and drill down to group by p_category to see where the profit change arises.
- Q4.3 digs deeper, restricting s_nation to 'UNITED STATES' and p_category = 'MFGR#14', drilling down to group by s_city (in the USA) and p_brand1 (within p_category 'MFGR#14').

## Star schema benchmark queries (denormalized)

Query 1.1

select sum(lo_extendedprice*lo_discount) as revenue from denormalized where d_year = 1993 and lo_discount between 1 and 3 and lo_quantity < 25

Query 1.2

select sum(lo_extendedprice*lo_discount) as lo_revenue from denormalized where d_yearmonthnum = 199401 and lo_discount between 4 and 6 and lo_quantity between 26 and 35

Query 1.3

select sum(lo_extendedprice*lo_discount) as lo_revenue from denormalized where d_weeknuminyear = 6 and d_year = 1994 and lo_discount between 5 and 7 and lo_quantity between 26 and 35

Query 2.1

select sum(lo_revenue), d_year, p_brand1 from denormalized where p_category = 'MFGR#12' and s_region = 'AMERICA' group by d_year, p_brand1 order by d_year, p_brand1

Query 2.2

select sum(lo_revenue), d_year, p_brand1 from denormalized where p_brand1 between 'MFGR#2221' and 'MFGR#2228' and s_region = 'ASIA' group by d_year, p_brand1 order by d_year, p_brand1

Query 2.3

select sum(lo_revenue), d_year, p_brand1 from denormalized where p_brand1= 'MFGR#2239' and s_region = 'EUROPE' group by d_year, p_brand1 order by d_year, p_brand1

Query 3.1

select c_nation, s_nation, d_year, sum(lo_revenue) as lo_revenue from denormalized where c_region = 'ASIA' and s_region = 'ASIA' and d_year >= 1992 and d_year <= 1997 group by c_nation, s_nation, d_year order by d_year asc, lo_revenue desc

Query 3.2

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from denormalized where c_nation = 'UNITED STATES' and s_nation = 'UNITED STATES' and d_year >= 1992 and d_year <= 1997 group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 3.3

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from denormalized where (c_city='UNITED KI1' or c_city='UNITED KI5') and (s_city='UNITED KI1' or s_city='UNITED KI5') and d_year >= 1992 and d_year <= 1997 group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 3.4

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from denormalized where (c_city='UNITED KI1' or c_city='UNITED KI5') and (s_city='UNITED KI1' or s_city='UNITED KI5') and d_yearmonth = 'Dec1997' group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 4.1

select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit from denormalized where c_region = 'AMERICA' and s_region = 'AMERICA' and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, c_nation order by d_year, c_nation

Query 4.2

select d_year, s_nation, p_category, sum(lo_revenue - lo_supplycost) as profit from denormalized where c_region = 'AMERICA' and s_region = 'AMERICA' and (d_year = 1997 or d_year = 1998) and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, s_nation, p_category order by d_year, s_nation, p_category

Query 4.3

select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit from denormalized where c_region = 'AMERICA' and s_nation = 'UNITED STATES' and (d_year = 1997 or d_year = 1998) and p_category = 'MFGR#14' group by d_year, s_city, p_brand1 order by d_year, s_city, p_brand1

## Optimized Apache Druid queries

Query 1.1

select sum(lo_extendedprice*lo_discount) as revenue from ssb_data where floor(__time to YEAR) = TIMESTAMP '1993-01-01' and lo_discount between 1 and 3 and lo_quantity < 25

Query 1.2

select sum(lo_extendedprice*lo_discount) as lo_revenue from  ssb_data where TIME_FLOOR(\"__time\",'P1M') = TIME_PARSE('1994-01-01T00:00:00.000Z') and lo_discount between 4 and 6 and lo_quantity between 26 and 35

Query 1.3

select sum(lo_extendedprice*lo_discount) as lo_revenue from ssb_data where

TIME_FLOOR(__time,'P1W')=TIME_PARSE('1994-02-07T00:00:00.000Z') and lo_discount between 5 and 7 and lo_quantity between 26 and 35

Query 2.1

select sum(lo_revenue), d_year, p_brand1 from ssb_data  where p_category = 'MFGR#12' and s_region = 'AMERICA' group by d_year, p_brand1 order by d_year, p_brand1

Query 2.2

select sum(lo_revenue), d_year, p_brand1 from ssb_data  where p_brand1 between 'MFGR#2221' and 'MFGR#2228' and s_region = 'ASIA' group by d_year, p_brand1 order by d_year, p_brand1

Query 2.3

select sum(lo_revenue), d_year, p_brand1 from ssb_data  where p_brand1= 'MFGR#2239' and s_region = 'EUROPE' group by d_year, p_brand1 order by d_year, p_brand1

Query 3.1

select c_nation, s_nation, d_year, sum(lo_revenue) as lo_revenue from ssb_data  where c_region = 'ASIA' and s_region = 'ASIA' and TIME_EXTRACT(\"__time\",'YEAR') >= 1992 and TIME_EXTRACT(\"__time\",'YEAR') <= 1997 group by c_nation, s_nation, d_year order by d_year asc, lo_revenue desc

Query 3.2

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from ssb_data  where c_nation = 'UNITED STATES' and s_nation = 'UNITED STATES' and TIME_EXTRACT(\"__time\",'YEAR') >= 1992 and TIME_EXTRACT(\"__time\",'YEAR')  <= 1997 group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 3.3

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from ssb_data  where (c_city='UNITED KI1' or c_city='UNITED KI5') and (s_city='UNITED KI1' or s_city='UNITED KI5') and FLOOR(\"__time\" to YEAR) >= TIME_PARSE('1992-01-01T00:00:00.000Z') and FLOOR(\"__time\" to YEAR) <= TIME_PARSE('1997-01-01T00:00:00.000Z') group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 3.4

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from ssb_data where (c_city='UNITED KI1' or c_city='UNITED KI5') and (s_city='UNITED KI1' or s_city='UNITED KI5') and TIME_FLOOR(""__time"",'P1M') = TIME_PARSE('Dec1997','MMMyyyy') group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 4.1

select d_year, c_nation, sum(lo_revenue) - sum(lo_supplycost) as profit from ssb_data  where c_region = 'AMERICA' and s_region = 'AMERICA' and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, c_nation order by d_year, c_nation

Query 4.2

select d_year, s_nation, p_category, sum(lo_revenue) - sum(lo_supplycost) as profit from ssb_data  where c_region = 'AMERICA' and s_region = 'AMERICA' and (FLOOR(\"__time\" to YEAR) = TIME_PARSE('1997-01-01T00:00:00.000Z') or FLOOR(\"__time\" to YEAR) = TIME_PARSE('1998-01-01T00:00:00.000Z')) and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, s_nation, p_category order by d_year, s_nation, p_category

Query 4.3

select d_year, s_nation, p_category, sum(lo_revenue) - sum(lo_supplycost) as profit from ssb_data where c_region = 'AMERICA' and s_region = 'AMERICA' and (FLOOR(\"__time\" to YEAR) = TIME_PARSE('1997-01-01T00:00:00.000Z') or FLOOR(\"__time\" to YEAR) = TIME_PARSE('1998-01-01T00:00:00.000Z')) and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, s_nation, p_category order by d_year, s_nation, p_category

## Google BigQuery optimized queries

Query 1.1

select sum(lo_extendedprice*lo_discount) as revenue from `community-benchmark.SSBData.ssb_data_small_part` where (date(f0_) between ""1993-01-01"" and ""1993-12-31"") and lo_discount between 1 and 3 and lo_quantity < 25

Query 1.2

select sum(lo_extendedprice*lo_discount) as lo_revenue from `community-benchmark.SSBData.ssb_data_small_part` where (date(f0_) between ""1994-01-01"" and ""1994-01-31"") and lo_discount between 4 and 6 and lo_quantity between 26 and 35

Query 1.3

select sum(lo_extendedprice*lo_discount) as lo_revenue from `community-benchmark.SSBData.ssb_data_small_part` where d_weeknuminyear = 6 and

DATE (f0_) BETWEEN ""1994-01-01"" and ""1994-12-31"" and lo_discount between 5 and 7 and lo_quantity between 26 and 35

Query 2.1

select sum(lo_revenue), d_year, p_brand1 from `community-benchmark.SSBData.ssb_data_small_part`  where p_category = 'MFGR#12' and s_region = 'AMERICA' group by d_year, p_brand1 order by d_year, p_brand1

Query 2.2

select sum(lo_revenue), d_year, p_brand1 from `community-benchmark.SSBData.ssb_data_small_part`  where p_brand1 between 'MFGR#2221' and 'MFGR#2228' and s_region = 'ASIA' group by d_year, p_brand1 order by d_year, p_brand1

Query 2.3

select sum(lo_revenue), d_year, p_brand1 from `community-benchmark.SSBData.ssb_data_small_part`  where p_brand1= 'MFGR#2239' and s_region = 'EUROPE' group by d_year, p_brand1 order by d_year, p_brand1

Query 3.1

select c_nation, s_nation, d_year, sum(lo_revenue) as lo_revenue from `community-benchmark.SSBData.ssb_data_small_part`where c_region = 'ASIA' and s_region = 'ASIA' and (DATE(f0_) BETWEEN ""1992-01-01"" AND ""1997-12-31"") group by c_nation, s_nation, d_year order by d_year asc, lo_revenue desc

Query 3.2

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from `community-benchmark.SSBData.ssb_data_small_part` where c_nation = 'UNITED STATES' and s_nation = 'UNITED STATES' and (DATE(f0_) BETWEEN ""1992-01-01"" AND ""1997-12-31"") group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 3.3

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from `community-benchmark.SSBData.ssb_data_small_part` where (c_city='UNITED KI1' or c_city='UNITED KI5') and (s_city='UNITED KI1' or s_city='UNITED KI5') and (DATE(f0_) BETWEEN ""1992-01-01"" AND ""1997-12-31"") group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 3.4

select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from `community-benchmark.SSBData.ssb_data_small_part` where (c_city='UNITED KI1' or c_city='UNITED KI5') and (s_city='UNITED KI1' or s_city='UNITED KI5') and (DATE(f0_) BETWEEN ""1997-12-01"" AND ""1997-12-31"") group by c_city, s_city, d_year order by d_year asc, lo_revenue desc

Query 4.1

select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit from `community-benchmark.SSBData.ssb_data_small_part` where c_region = 'AMERICA' and s_region = 'AMERICA' and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, c_nation order by d_year, c_nation

Query 4.2

select d_year, s_nation, p_category, sum(lo_revenue - lo_supplycost) as profit from `community-benchmark.SSBData.ssb_data_small_part` where c_region = 'AMERICA' and s_region = 'AMERICA'

and (DATE(f0_) BETWEEN ""1997-01-01"" AND ""1998-12-31"") and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, s_nation, p_category order by d_year, s_nation, p_category

Query 4.3

```
select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit from
`community-benchmark.SSBData.ssb_data_small_part` where s_nation = 'UNITED STATES'
and (DATE(f0_) BETWEEN ""1997-01-01"" AND ""1998-12-31"") and p_category = 'MFGR#14'
group by d_year, s_city, p_brand1 order by d_year, s_city, p_brand1
```

## About Apache Druid

Apache Druid is an open source distributed data store. Druid's core design combines ideas from data warehouses, time series databases, and search systems to create a unified system for real-time analytics for a broad range of use cases. Druid merges key characteristics of each of these three architectures into its ingestion, storage and querying layers.

## About Imply

Imply transforms how businesses run by integrating real-time analytics into their operations. Founded by the authors of the Apache Druid database, Imply provides a cloud-native solution that delivers real-time ingestion, interactive ad-hoc queries, and intuitive visualizations for many types of event-driven and streaming data flows. Imply has operations in North America, Europe, and Asia Pacific and is backed by Andreesen Horowitz, Khosla Ventures, and Geodesic Capital. For more information visit, please visit imply.io.

If you are interested in trying out Druid or Imply, you can download Imply or request an Imply Cloud Trial Account.