ROCKSET

Rockset Performance Evaluation on the Star Schema Benchmark

April 2022

Abstract: Low query latency is an important requirement for real-time analytics. We evaluate Rockset's ability to support real-time analytics by measuring its query performance on the Star Schema Benchmark, an industry-standard benchmark for analytical applications. This paper provides an overview of the benchmark data and queries, describes the configuration for running the benchmark, and discusses the results from the evaluation.

Authors: Ben Hannel & Kevin Leong

Table of Contents

Abstract **1**

Table of Contents **2**

Executive Summary **3**

Star Schema Benchmark Overview **4**

Benchmark Data 4

Benchmark Queries 5

Performance Evaluation **6**

Rockset Configuration 7

Data Generation and Ingestion 7

Query Modification 8

Performance Measurement 8

Benchmark Results **9**

Analysis of Results 10

Column-Based Index Format 11

SIMD Vectorized Query Execution 12

RocksDB Block Size 12

Column-Based Cluster Splitting 13

More Efficient Check for Set Containment 14

Cache for Column-Based Cluster Metadata 14

Conclusion 15

Appendix **16**

Original SSB Queries 16

SSB Queries on Denormalized Data 21

Optimized SSB Queries 25

SSB Queries Explained 31

List of Query Optimizations 33

Executive Summary

A database that supports real-time analytics must be capable of both high write rates and low-latency queries, enabling applications to operate on data within limited time windows. Both data latency—the lag between when data is produced and when it is available to query—and query latency—the time taken for queries to be executed and results returned—are important measures of performance for real-time analytics.

Minimizing data latency is critical to ensuring that applications are querying up-to-date data. We designed a benchmark to measure data latency for databases and published our methodology and results in a separate document.

This paper focuses on the query latency aspect of real-time analytics. Once incoming data is made queryable, it is essential that queries execute quickly so the applications they drive can take actions within tight time constraints or deliver immediate insights to users. An inability to meet query latency requirements can result in missed opportunities, failure to detect threats or a poor user experience.

We evaluated the query latency characteristics of Rockset, a real-time indexing database used to back analytical applications, using the Star Schema Benchmark (SSB). The SSB is designed to measure database performance for analytical applications. It is a widely recognized industry-standard benchmark, with queries that have both good functional and good selectivity coverage. While the SSB is oriented towards batch analytics scenarios, and not the real-time analytics Rockset is designed for, the results from a SSB evaluation can yield valuable insight into Rockset's query performance on a range of common analytical queries.

Running the SSB on Rockset, using a single m5.8xlarge server, we found that Rockset executed every query in the SSB suite in 88 milliseconds or less. The aggregate time to execute all 13 queries in the SSB was 664 milliseconds.

Star Schema Benchmark Overview

The SSB derives from TPC-H, a popular decision support benchmark, and modifies the schema and queries from TPC-H to represent popular star-schema-based business analytics workloads.

Benchmark Data

SSB data is organized into a star schema, consisting of the following tables:

- **Fact table**: *lineorder*
- **Dimension tables**: *customer, supplier, part, date*

The diagram below depicts the SSB schema and the relationships between tables and fields.

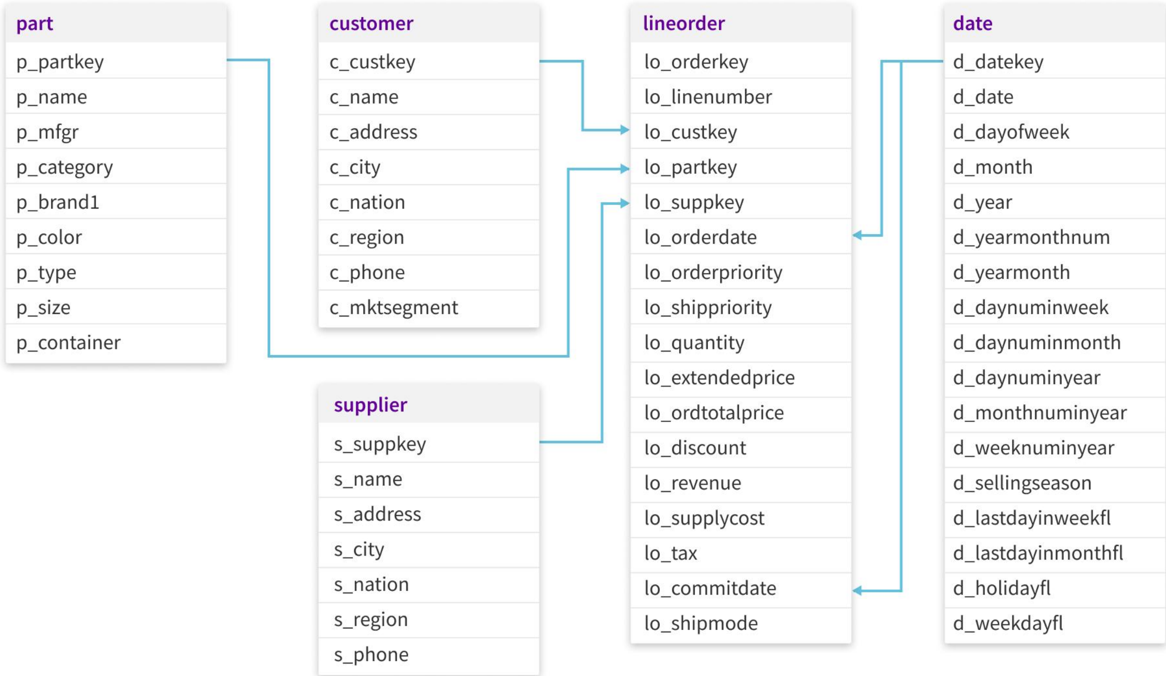

Figure 1: Graphical representation of the SSB schema

The size of the benchmark data set can be varied through the use of a Scale Factor. For instance, the **lineorder** fact table will have 6M rows at Scale Factor 1 and 600M rows at Scale Factor 100. The dimension tables also grow in size—though not at the same rate as the fact table—as the Scale Factor increases.

Benchmark Queries

The SSB suite has 4 query flights, each having 3-4 queries.

Query Flight 1 (Q1.1, Q1.2 and Q1.3) performs a single join between the **lineorder** fact table and the **date** dimension table. Predicates are set on the fields **d_year**, **lo_discount** and **lo_quantity**. The predicate on **date** is primarily used to vary the data selectivity across the queries in this query flight. For instance, the date range in Q1.1 spans 1 year, which decreases to 1 month in Q1.2, which further decreases to 1 week in Q1.3

Query flight 2 joins the fact table **lineorder** with 3 dimension tables, **date**, **part**, and **supplier**. They aggregate and sort the revenue by the fields **d_year** and **p_brand1**. The predicates are not time based in this query flight. The selectivity is modified based on varying the fields **s_region**, **p_category**, and **p_brand1**. The fields **p_category** and **p_brand1** are part of the selectivity hierarchy of the same dimension table **part**.

Query Flight 3 joins the fact table **lineorder** with 3 dimension tables, **customer**, **supplier**, and **date**. It aggregates the revenue based on the customer and supplier regions and year. Across the queries in this flight, it varies the selectivities by using the region-based selectivity hierarchies in customer and supplier tables: **city**, **nation**, **region**. Q3.1, 3.2, 3.3 have a date predicate spanning 6 years from 1992 to 1997. Q3.4 additionally also narrows the date selectivity to 1 month.

Query Flight 4 joins all tables. The queries calculate the total profit by drilling down into the **region** and **manufacturer** by using the selectivity hierarchies in the **part**, **customer** and **supplier** tables. Q4.2 and Q4.3 also add a predicate on **d_year** to restrict the results to a range of 2 years only.

Query Selectivity

The SSB queries have varying levels of selectivity. Selectivity is simply the fraction of rows that satisfy the query predicate. Each dimension table has selectivity hierarchies. This allows a simple way to control the row selectivity of the queries and enables the SSB to provide a good combination of both functional and selectivity coverage.

The table below lists the SSB query id with the corresponding selectivity on **lineorder**, or fraction of rows, that needs to be accessed to answer the query.

Query	Selectivity
1.1	0.0198
1.2	0.0007
1.3	0.0002
2.1	0.0079
2.2	0.0016
2.3	0.00019
3.1	0.036
3.2	0.0014
3.3	0.000055
3.4	0.000007782
4.1	0.015
4.2	0.0038
4.3	0.000077

Figure 2: Benchmark queries and their respective selectivity ratios

Performance Evaluation

Evaluating Rockset's performance on the SSB involved several steps:

1. Selecting the Rockset configuration
2. Generating and ingesting the SSB data
3. Modifying the SSB queries
4. Measuring query latencies

Rockset Configuration

We ran the SSB on Rockset using a single m5.8xlarge server that has 32 vCPU and 128 GiB RAM.

Data Generation and Ingestion

We generated the SSB data at scale factor 100, which corresponds to 100GB and 600M rows of data. This was done using the dbgen utility that can be found here:
https://github.com/lemire/StarSchemaBenchmark

For this evaluation, we denormalized the data prior to ingestion with the goal of measuring performance without query-time joins. The denormalized data set is available in Amazon S3:
s3://rs-benchmarks/ssb/100GB/data/denormalized/

Rockset allows column-based clustering to speed up query execution. This clustering scheme is configured at collection creation time. The goal of clustering is to colocate the data required to serve a query on storage as much as possible. This helps make the query faster primarily for 2 reasons:

1. Sequential scans are faster than random seeks
2. We can take advantage of pruning by scanning less data

Based on the commonly recurring predicates across all the SSB queries, we chose to cluster the data in the denormalized data set along multiple dimensions. The clustering scheme that we used for this benchmark was as follows:

d_year, s_region, c_region, p_category, d_monthnuminyear

With this configuration, Rockset's column-based index was ordered according to this clustering schema, such that documents with similar values of the clustering fields were located near each other on storage.

Query Modification

The SSB permits variant query forms, allowing “any alternative SQL form that modifies predicate restrictions but retains the same effect on retrieval” (The Star Schema Benchmark). Given this, we removed the joins from the original queries which were no longer needed since the data set was denormalized before ingestion.

In addition, we rewrote some queries to pick the column scan access path during query execution, while the predicates were also modified, without changing their semantics, such that the query would run efficiently with the benefits of clustering. We verified that the result set with these rewrites remained unchanged.

The original, denormalized and optimized queries can be found in the **Appendix** at the end of this document.

Performance Measurement

Having generated the SSB data and loaded it into Rockset, we ran a load-generator query script using Python.

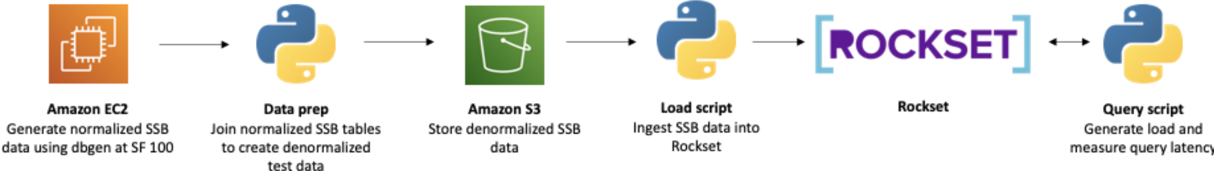

Figure 3: Performance harness used to generate and load SSB data, run queries and measure query runtimes.

We ran each query 10 times on a warmed OS cache and reported the mean of the run times. There was no form of query caching used for the evaluation. The times are reported by Rockset's API Server.

Benchmark Results

We ran 4 flights of SSB queries described previously and obtained the following results.

Query	Runtime (ms)
Q1.1	71
Q1.2	80
Q1.3	87
Q2.1	34
Q2.2	35
Q2.3	29
Q3.1	88
Q3.2	52
Q3.3	76
Q3.4	19
Q4.1	53
Q4.2	23
Q4.3	17
Total	**664**

Figure 3: Benchmark results when running SSB on Rockset at scale factor 100

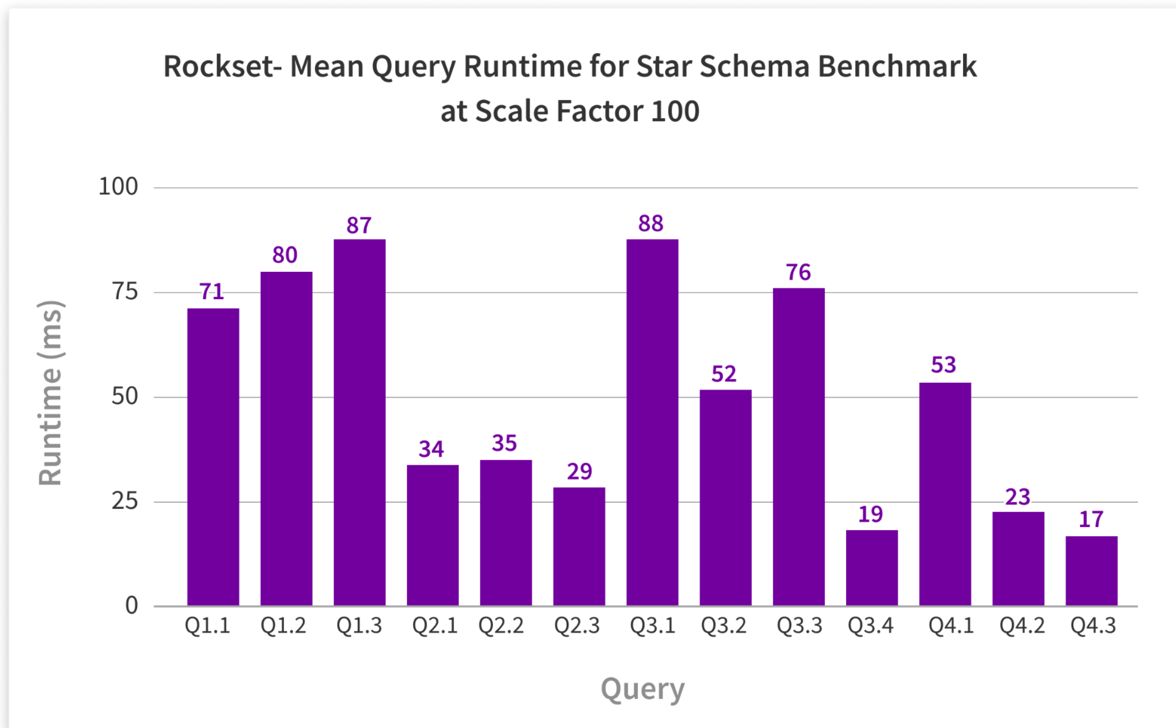

Figure 4: Graph of query runtimes when running SSB on Rockset at scale factor 100

Analysis of Results

We introduced several performance enhancements that enabled Rockset to execute every query in the SSB suite in under 88 milliseconds: an improved column group format, Single Instruction/Multiple Data (SIMD) for function evaluation, automatic column cluster splitting, a custom RocksDB block size policy, a more efficient check for set containment, and a cache for column cluster metadata. In the following sections, we discuss how each feature helps accelerate SSB queries.

Column-Based Index Format

Rockset builds a Converged Index on all ingested data. The Converged Index is a combination of:

- Inverted index (also known as a search index)
- Column-based index
- Row-based index

These various indexes help make multiple access patterns, including key-value, time-series, document, search and aggregation queries, execute efficiently.

Rockset introduced a new version of the on-disk format for the column-based index that has better compression, faster decoding and computations on compressed data.

The new format supports dictionary encoding for strings. This means that if the same string is repeated multiple times within one chunk of data in the column-based index, the string is only stored on disk once, and we just store the index of that string. This reduces space usage on disk, and since the data is more compact, it is faster to load from disk or memory. We continue to store the strings in dictionary encoded format in memory, and we can compute on that format. Say, for instance, that you need to evaluate a function on a string. SSB query 2.1 provides an example of this:

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

`s_region` contains many duplicates, because there are only 5 distinct regions in the dataset. When evaluating the predicate `s_region = 'AMERICA'` we perform string comparison once for each unique string in the dictionary, so at most 5 times. We then reuse that result for each original element of the array by cross-referencing the dictionary. The new columnar format also has other advantages, like handling null values more efficiently, and it is more extensible.

SIMD Vectorized Query Execution

The data exchange between query execution operators happens in the form of data chunks, which organize data in a columnar format. This makes it feasible for the operators to execute in a vectorized manner, where operations can be performed on a set of values instead of one value at a time, for more efficient query execution. With SIMD instructions, we leverage modern processors that can compute on 256 bits or 512 bits of data at a time with a single CPU instruction.

For example, the `_mm256_cmpeq_epi64` intrinsic can compare four 64-bit integers in a single instruction. For batch processing operations, this can substantially increase throughput. The comparison itself isn't the end of the story though. SIMD instructions typically operate within a lane - so if you use four 64-bit inputs, you get four 64-bit outputs. That means instead of getting booleans as outputs, you get four 64-bit integers at the output. Typically when operating on booleans, you either want an array of booleans as the output, or a bitmask. We took great care to optimize that conversion step to see the maximum possible performance gain from SIMD.

RocksDB Block Size

RocksDB is a high performance embedded storage engine used by modern datastores like Kafka and Flink. Rockset stores its indexes on RocksDB. As the SSB queries access data using the column-based index, larger storage blocks were configured for that index to improve throughput.

RocksDB divides data into blocks. These blocks are the unit of data lookup for various operations, like reading from disk or reading from RocksDB's in-memory block cache. The size of these blocks is configurable. Larger blocks help with throughput for large scans because you need to do fewer total lookups in the block cache and fewer random accesses to main memory. Smaller blocks help with performance for point lookups because if you only need one key you can load less surrounding data. The cost of loading a large block does not amortize well if you only need 1% of the data in it. You also waste space in the cache by storing data that was not recently accessed.

For Rockset's inverted index and row-based index, which are often used for point lookups, a small block size makes sense. For the column-based index though, which is often used for bulk scans, a much larger block size improves throughput. We created a custom block size policy under the hood to tune the block size for each index independently and increased the size of the column-based index blocks.

Column-Based Cluster Splitting

SSB queries have narrow selectivity, meaning the number of rows that need to be processed is significantly smaller in size than the total number of rows in the collection. Since sequential access is much faster than random access, we can answer these queries more quickly if data needed for a query is located closer together on storage. To be able to do this effectively, Rockset supports creating a clustering key, which comprises a subset of the collection's fields that your queries are most likely to filter on. The column-based index is then ordered by the clustering key, so that documents with the same clustering field values are automatically stored together during ingestion. This enables our execution engine to scan only a subset of documents in clusters with predicates that match some or all specified clustering fields, rather than filtering the rows after scanning the entire collection. Such cluster-based pruning reduces the amount of data retrieved from disk and shortens processing time, thus improving query performance.

For instance, a subset of SSB queries are analytical queries that filter the results by the year, or other date fields, in the selectivity hierarchy. If the goal is to speed up one such query, you might specify one of the clustering fields as **d_year** during collection creation. This will result in the values for all fields for any particular year being stored contiguously on disk for efficient retrieval. Q1.1 which filters results on **d_year = 1993** would need to only scan the cluster corresponding to year 1993, thus eliminating the need to scan all the data.

In the new version of column-based clustering, Rockset starts by putting all documents into the same cluster, regardless of their clustering key, then automatically splits clusters into smaller clusters based on their clustering key when they grow too large. This makes the column-based clustering easier to configure, and improves read throughput by making sure clusters are always properly sized.

More Efficient Check for Set Containment

Rockset has a SQL-based query execution engine that can filter, aggregate and join data. During query execution, we introduced an efficient check for set containment to reduce compute costs.

For example, in SSB query 3.3, there is a step in query execution to check if a field is in some set of constant values:

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from denormalized HINT(access_path=column_scan)
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
  and c_region = 'EUROPE'
  and s_region = 'EUROPE'
group by c_city, s_city, d_year
order by d_year asc, revenue desc
```

We have an optimization for this pattern, where we build a hash set of all of the constant values, and then for each variable input value, we hash it and check if it is contained in the set. This is a good optimization for long lists of constants, but for lists of just 2 or 3 constants, hashing the input is more expensive than doing 2 or 3 direct equality comparisons. We tuned this optimization to only apply to constant lists which are large enough where it is appropriate.

Cache for Column-Based Cluster Metadata

Rockset increased the efficiency of the column-based cluster selection phase—the phase that determines what clusters need to be scanned in a query—by storing the metadata in memory.

In the first version of column-based clustering, we would iterate over the clusters in RocksDB, and skip any clusters where their key ranges did not intersect with the key ranges the query is looking for.

In the most recent version of column-based clustering, information about which clusters exist and what key ranges they cover is indexed and stored in memory. Since each cluster is fairly large (several thousand documents) and the metadata is only a few bytes per cluster, the memory overhead of this caching is only a fraction of a percent of the total data size. It makes the cluster selection phase measurably faster.

Conclusion

From the SSB results on Rockset, we observe that every query in the SSB suite executed in 88 milliseconds or less. The aggregate query runtime across all queries was 664 milliseconds.

The benchmark results demonstrate Rockset's ability to support complex analytical queries with millisecond-latency performance, a common requirement for applications like personalization, logistics tracking, fraud detection and operational analytics. This characteristic, along with the ability to handle high write rates, makes Rockset well suited for serving real-time analytics for a wide variety of use cases.

Appendix

Original SSB Queries

Q1.1

```
select sum(lineorder.lo_extendedprice * lineorder.lo_discount) as
revenue
from lineorder
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where dwdate.d_year = 1993
  and lineorder.lo_discount between 1 and 3
  and lineorder.lo_quantity < 25;
```

Q1.2

```
select sum(lineorder.lo_extendedprice * lineorder.lo_discount) as
revenue
from lineorder
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where dwdate.d_yearmonthnum = 199401
  and lineorder.lo_discount between 4 and 6
  and lineorder.lo_quantity between 26 and 35;
```

Q1.3

```
select sum(lineorder.lo_extendedprice * lineorder.lo_discount) as
revenue
from lineorder
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where dwdate.d_weeknuminyear = 6
  and dwdate.d_year = 1994
  and lineorder.lo_discount between 5 and 7
  and lineorder.lo_quantity between 26 and 35;
```

Q2.1

```
select sum(lineorder.lo_revenue), dwdate.d_year, part.p_brand1
from lineorder
  join part on lineorder.lo_partkey = part.p_partkey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where part.p_category = 'MFGR#12'
  and supplier.s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Q2.2

```
select sum(lineorder.lo_revenue), dwdate.d_year, part.p_brand1
from lineorder
  join part on lineorder.lo_partkey = part.p_partkey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where part.p_brand1 between 'MFGR#2221' and 'MFGR#2228'
  and supplier.s_region = 'ASIA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Q2.3

```
select sum(lineorder.lo_revenue), dwdate.d_year, part.p_brand1
from lineorder
  join part on lineorder.lo_partkey = part.p_partkey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where part.p_brand1 = 'MFGR#2221'
  and supplier.s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Q3.1

```
select customer.c_nation, supplier.s_nation, dwdate.d_year,
sum(lineorder.lo_revenue) as revenue
from lineorder
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
  join customer on lineorder.lo_custkey = customer.c_custkey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
where  customer.c_region = 'ASIA'
  and supplier.s_region = 'ASIA'
  and dwdate.d_year >= 1992
  and dwdate.d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

Q3.2

```
select customer.c_city, supplier.s_city, dwdate.d_year,
sum(lineorder.lo_revenue) as revenue
from lineorder
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
  join customer on lineorder.lo_custkey = customer.c_custkey
where customer.c_nation = 'UNITED STATES'
  and supplier.s_nation = 'UNITED STATES'
  and dwdate.d_year >= 1992
  and dwdate.d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Q3.3

```
select customer.c_city, supplier.s_city, dwdate.d_year,
sum(lineorder.lo_revenue) as revenue
from lineorder
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
  join customer on lineorder.lo_custkey = customer.c_custkey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
where (customer.c_city='UNITED KI1' or customer.c_city='UNITED
KI5')
  and (supplier.s_city='UNITED KI1' or supplier.s_city='UNITED
KI5')
  and dwdate.d_year >= 1992
  and dwdate.d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Q3.4

```
select customer.c_city, supplier.s_city, dwdate.d_year,
sum(lineorder.lo_revenue) as revenue
from lineorder
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
  join customer on lineorder.lo_custkey = customer.c_custkey
where (customer.c_city='UNITED KI1' or customer.c_city='UNITED
KI5')
  and (supplier.s_city='UNITED KI1' or supplier.s_city='UNITED
KI5')
  and dwdate.d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Q4.1

```
select dwdate.d_year, customer.c_nation, sum(lineorder.lo_revenue -
lineorder.lo_supplycost) as profit
from lineorder
  join customer on lineorder.lo_custkey = customer.c_custkey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
  join part on lineorder.lo_partkey = part.p_partkey
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where customer.c_region = 'AMERICA'
  and supplier.s_region = 'AMERICA'
  and (part.p_mfgr = 'MFGR#1' or part.p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation;
```

Q4.2

```
select dwdate.d_year, supplier.s_nation, part.p_category,
sum(lineorder.lo_revenue - lineorder.lo_supplycost) as profit
from lineorder
  join customer on lineorder.lo_custkey = customer.c_custkey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
  join part on lineorder.lo_partkey = part.p_partkey
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where customer.c_region = 'AMERICA'
  and supplier.s_region = 'AMERICA'
  and (dwdate.d_year = 1997 or dwdate.d_year = 1998)
  and (part.p_mfgr = 'MFGR#1' or part.p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category;
```

Q4.3

```
select dwdate.d_year, supplier.s_city, part.p_brand1,
sum(lineorder.lo_revenue - lineorder.lo_supplycost) as profit
from lineorder
  join customer on lineorder.lo_custkey = customer.c_custkey
  join supplier on lineorder.lo_suppkey = supplier.s_suppkey
  join part on lineorder.lo_partkey = part.p_partkey
  join dwdate on lineorder.lo_orderdate = dwdate.d_datekey
where customer.c_region = 'AMERICA'
  and supplier.s_nation = 'UNITED STATES'
  and (dwdate.d_year = 1997 or dwdate.d_year = 1998)
  and part.p_category = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city, p_brand1;
```

SSB Queries on Denormalized Data

Q1.1

```
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;
```

Q1.2

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

Q1.3

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

Q2.1

```
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;
```

Q2.2

```
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;
```

Q2.3

```
select sum(lo_revenue), d_year, p_brand1
from denormalized
where p_brand1 = 'MFGR#2221'
  and s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Q3.1

```
select c_nation, s_nation, d_year, sum(lo_revenue) as 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, revenue desc;
```

Q3.2

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

Q3.3

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

Q3.4

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

Q4.1

```
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;
```

Q4.2

```
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;
```

Q4.3

```
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 SSB Queries

Q1.1

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

            from denormalized HINT(access_path=column_scan)

            where d_year = 1993

              and lo_discount between 1 and 3

              and lo_quantity < 25;
```

Q1.2

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

            from denormalized HINT(access_path=column_scan)

            where d_year = 1994 and d_monthnuminyear = 1

              and lo_discount between 4 and 6

              and lo_quantity between 26 and 35;
```

Q1.3

```
select sum(lo_extendedprice * lo_discount) as revenue
            from denormalized HINT(access_path=column_scan)
            where d_weeknuminyear = 6
              and d_year = 1994
              and d_monthnuminyear = 2
              and lo_discount between 5 and 7
              and lo_quantity between 26 and 35;
```

Q2.1

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

Q2.2

```
         select sum(lo_revenue), d_year, p_brand1
            from denormalized HINT(access_path=column_scan)
            where p_brand1 between 'MFGR#2221' and 'MFGR#2228'
              and s_region = 'ASIA'
              and p_category = 'MFGR#22'
            group by d_year, p_brand1
            order by d_year, p_brand1;
```

Q2.3

```
select sum(lo_revenue), d_year, p_brand1
            from denormalized HINT(access_path=column_scan)
            where p_brand1 = 'MFGR#2221'
              and s_region = 'EUROPE'
              and p_category = 'MFGR#22'
            group by d_year, p_brand1
            order by d_year, p_brand1;
```

Q3.1

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
       from denormalized HINT(access_path=column_scan)
       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, revenue desc;
```

Q3.2

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
     from denormalized HINT(access_path=column_scan)
     where c_nation = 'UNITED STATES'
       and s_nation = 'UNITED STATES'
       and c_region = 'AMERICA'
       and s_region = 'AMERICA'
       and d_year >= 1992
       and d_year <= 1997
     group by c_city, s_city, d_year
     order by d_year asc, revenue desc;
```

Q3.3

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
      from denormalized HINT(access_path=column_scan)
      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
        and c_region = 'EUROPE'
        and s_region = 'EUROPE'
      group by c_city, s_city, d_year
      order by d_year asc, revenue desc;
```

Q3.4

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
    from denormalized HINT(access_path=column_scan)
    where (c_city='UNITED KI1' or c_city='UNITED KI5')
      and (s_city='UNITED KI1' or s_city='UNITED KI5')
      and c_region = 'EUROPE'
      and s_region = 'EUROPE'
      and d_year = 1997
      and d_monthnuminyear = 12
    group by c_city, s_city, d_year
    order by d_year asc, revenue desc;
```

Q4.1

```
  select d_year, c_nation, sum(lo_revenue - lo_supplycost) as
profit
            from denormalized HINT(access_path=column_scan)
            where c_region = 'AMERICA'
              and s_region = 'AMERICA'
              and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
              and p_category between 'MFGR#11' and 'MFGR#29'
            group by d_year, c_nation
            order by d_year, c_nation;
```

Q4.2

```
 select d_year, s_nation, p_category, sum(lo_revenue -
lo_supplycost) as profit
            from denormalized HINT(access_path=column_scan)
            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')
              and p_category between 'MFGR#11' and 'MFGR#29'
            group by d_year, s_nation, p_category
            order by d_year, s_nation, p_category;
```

Q4.3

```
select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as
profit
            from denormalized HINT(access_path=column_scan)
            where c_region = 'AMERICA'
              and s_nation = 'UNITED STATES'
              and s_region = 'AMERICA'
              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;
```

SSB Queries Explained

Flight 1

This query flight measures revenue increase by varying date predicates across its selectivity hierarchy `(d_year, d_yearmonthnum, d_year, d_weeknuminyear)`, and discount codes permitted `lo_discount` and product order quantities `lo_quantity`.

Q1.1

Measures revenue increase for year 1993 with restrictions for `lo_quantity` < 25 and `lo_discount` between 1 and 3

Q1.2

Measures revenue increase for the month of January of the year 1994 with restrictions on `lo_quantity` between 26 and 35 and `lo_discount` between 4 and 6

Q1.3

Measures revenue for 6th week of the year 1994 with restrictions on `lo_quantity` between 36 and 40 and `lo_discount` between 5 and 7.

Flight 2

The query flight compares revenues for certain products, suppliers in a certain region, aggregated over product classes and order years.

Q2.1
Calculates the revenue by restricting the `p_category` to `'MFGR#12'` and `s_region` to `'AMERICA'`

Q2.2
Calculates the revenue by restricting `p_brand1` between `'MFGR#2221'` and `'MFGR#2228'` and `s_region` to `'ASIA'`

Q2.3
Calculates the revenue by restricting `p_brand1` to `'MFGR#2339'` and `s_region='EUROPE'`

Flight 3

Calculates the revenue grouped by the customer and supplier nations and year, restricted by region and over some defined time period.

Q3.1
Calculates revenue with restrictions on `c_region` and `s_region` being `'ASIA'`, and `d_year` spanning 6-year period from 1992 to 1998, aggregated over `c_nation`, `s_nation` and `d_year`

Q3.2
Calculates revenue with restrictions on `c_region` and `s_region` being `‘UNITED STATES’`, and `d_year` spanning a 6-year period from 1992 to 1998, aggregated over `c_nation`, `s_nation` and `d_year`.

Q3.3
Calculates the revenue with restrictions on `c_city` and `s_city` being 2 cities from United Kingdom i.e. `'UNITED KI1'` or `'UNITED KI5'`, and `d_year` spanning a 6-year period from 1992 to 1998, aggregated over `c_nation`, `s_nation` and `d_year`.

Q3.4

Calculates the revenue with restrictions on `c_city` and `s_city` being 2 cities from United Kingdom i.e. `'UNITED KI1'` or `'UNITED KI5'`, and the date predicate spanning 1 month in Dec 1997, aggregated over `c_nation`, `s_nation` and `d_year`.

Flight 4

Calculates the total profit filtered by various different constraints based on the region and manufacturer

Q4.1

Calculates the total profit aggregated over year and customer region and restricting `c_region` and `s_region` to `'AMERICA'`, and restricting the manufacturer `p_mgfr` to `'MFGR#1'` `or` `'MFGR#2'`

Q4.2

Calculates the total profit aggregated over year, supplier nation, and product category restricting date to a span of 2 years 1997, 1998 and manufacturer `p_mgfr` to `'MFGR#1'` or `'MFGR#2'`

Q4.3

Calculates the total profit aggregated over year, supplier city, product brand restricting the date to a span of 2 years 1997, 1998 and category `p_category` to `'MFGR#14'`

List of Query Optimizations

The queries were rewritten in a way such that they could leverage clustering and execute more efficiently to yield better performance. Here is the clustering key for reference: `d_year`, `s_region`, `c_region`, `p_category`, `d_monthnuminyear`

Q1.1

- Enforce the access path to be column scan to take advantage of clustering on `d_year`.

Q1.2

- Enforce the access path to be column scan to take advantage of clustering on `d_year` and `d_monthnuminyear`.

- Predicate `d_yearmonthnum = 199401` is rewritten as `d_year = 1994 and d_monthnuminyear = 1` to take advantage of the clustering fields `d_year` and `d_monthnuminyear`

Q1.3

- Enforce the access path to be column scan to take advantage of clustering on `d_year` and `d_monthnuminyear`.
- Rewrite predicate `d_weeknuminyear = 6 and d_year = 1994` to `d_weeknuminyear = 6 and d_year = 1994 and and d_monthnuminyear = 2` to take advantage of clustering on fields `d_year` and `d_monthnuminyear`.

Q2.1

- Enforce the access path to be column scan to take advantage of clustering on fields `p_category` and `s_region`.

Q2.2

- Enforce the access path to be column scan to take advantage of clustering on fields `p_category` and `s_region`.
- Predicate `p_brand1 between 'MFGR#2221' and 'MFGR#2228' and s_region = 'ASIA'` is augmented with `p_category = 'MFGR#22'` to make use of the clustering on field `p_category`.

Q2.3

- Enforce the access path to be column scan to take advantage of clustering on `p_category` and `s_region`.
- Predicate `p_brand1 = 'MFGR#2221' and s_region = 'EUROPE'` is augmented with `p_category = 'MFGR#22'` to leverage clustering on the fields `p_category`.

Q3.1

- Enforce the access path to be column scan to take advantage of clustering on `c_region` and `s_region`.

Q3.2

- Enforce the access path to be column scan to take advantage of clustering on `c_region` and `s_region`.
- Region based predicates `c_nation = 'UNITED STATES' and s_nation = 'UNITED STATES'` are rewritten to `c_nation = 'UNITED STATES'`

  `and s_nation = 'UNITED STATES'`

  `and c_region = 'AMERICA'`

  `and s_region = 'AMERICA'` to leverage clustering on fields `s_region` and `c_rezgion`.

Q3.3

- Enforce the access path to use the search index since this query is highly selective overall, but does not offer much advantage in terms of pruning with the given clustering key.

Q3.4

- Enforce the access path to be column scan to take advantage of clustering on `c_region` and `s_region`.
- Predicate `(c_city='UNITED KI1' or c_city='UNITED KI5')`

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

  `and d_yearmonth = 'Dec1997' is rewritten to (c_city='UNITED KI1' or c_city='UNITED KI5')`

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

  `and c_region = 'EUROPE'`

  `and s_region = 'EUROPE'`

  `and d_year = 1997`

  `and d_monthnuminyear = 12`

  to leverage clustering on the fields `c_region`, `s_region`, `d_year`, `d_monthnuminyear`

Q4.1

- Enforce the access path to be column scan to take advantage of clustering on `c_region` and `s_region`.
- Rewrite constraint `(p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')` to `(p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')` and `p_category` between `'MFGR#11'` and `'MFGR#29'` to leverage clustering on field `p_category`

Q4.2

- Enforce the access path to be column scan to take advantage of clustering on `c_region` and `s_region`.
- Rewrite constraint `(p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')` to `(p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')` and `p_category` between `'MFGR#11'` and `'MFGR#29'` to take advantage of the clustering field `p_category`

Q4.3

- Enforce the access path to be column scan to take advantage of clustering on `c_region`, `s_region`, and `p_category`
- Rewrite constraint `s_nation = 'UNITED STATES'` to `s_nation = 'UNITED STATES'`
- `and s_region = 'AMERICA'`

  to take advantage of the clustering field `s_region`