# Benchmarking SnappyData with TPC-H

A SnappyData Performance Report

## Abstract

2

## Workload and Metrics

3

## Load Generation

5

- Steps to run the benchmark
- Data Generation
- Data Loading
- Query Execution
- Test Configuration

5

## TPC-H Results for SnappyData 1.0.2

9

- Aggregate Statistics/ Highlights
- Execution Time of TPC-H queries: Spark versus SnappyData

9

## Optimizations

13

- How SnappyData Improves on Spark Performance
- Impact of SnappyData optimizations on TPC-H Performance

13

## Test Environment

15

- SUT (System Under Test) Configuration and Tuning

16

## Summary

17

- References

17
Abstract

This report describes the findings of the performance benchmarking exercise for SnappyData 1.0.2.1 with TPC-H benchmark, along with the experiment process and details. SnappyData 1.0.2.1 was released in November 2018. Performance of this release was evaluated using the TPC-H benchmark.

There is a lack of standardization in the big data world for benchmarking of streaming/big-data systems and in-memory cloud native databases. TPC-H is an almost 20 years benchmark that defies its age and refuses to be superseded by the more comprehensive and younger TPC-DS benchmark for decision support systems; because it is well-designed, widely used and representative though relatively simple. As described in the paper 'TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark'[ref], It consists of an analytical workload with 22 ad-hoc SQL queries that present several technical challenges for database vendors to support with high performance that can be categorised into 6 choking points, viz: Aggregation Performance, Join Performance, Data Access Locality, Expression Calculation, Correlated Subqueries and Parallel Execution.

We haven't formally audited the results with the TPC council since the process is tedious and time consuming. However, SnappyData is open source. Our benchmark scripts, data and all the environment details etc. are available too, to make the benchmark reproducible and trustworthy. We encourage the readers to reproduce the benchmark. The objective of this document is to act as a reference guide and help reproduce the benchmark, by describing the experiment setup, benchmark configuration and benchmarking process.

This document focuses on the more important ‘timing test’ in TPC specification where queries are run sequentially and query power is evaluated. The ‘throughput test’ measure performance with concurrent queries. We have measured SnappyData performance in this area too and found it to be vastly superior. However, it will be addressed in a separate blog post and accompanying document with workloads consisting of concurrent analytical queries, or a mix of analytical and point-lookup queries.

The results show that not only has SnappyData performance further improved from release 1.0, but SnappyData continues to outperform Apache Spark by orders of magnitude.
Workload and Metrics

The TPC Benchmark™H (TPC-H) is a decision support benchmark. It models and represents complex, high data volume, decision support environments.

The queries and the data populating the database have been chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implementation. This benchmark illustrates decision support systems that

- Examine large volumes of data;
- Execute queries with a high degree of complexity;
- Give answers to critical business questions.

TPC-H Schema ER Diagram
A couple of tables (region and nation) are very small with less than 100 rows. Other tables are large, with the count of rows running into millions - depending on the scale factor chosen.

<table>
<thead>
<tr>
<th>Table</th>
<th>SF=1</th>
<th>SF=10</th>
<th>SF=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER</td>
<td>150,000</td>
<td>1,500,000</td>
<td>15,000,000</td>
</tr>
<tr>
<td>LINEITEM</td>
<td>6,000,000</td>
<td>59,986,052</td>
<td>600,037,902</td>
</tr>
<tr>
<td>ORDERS</td>
<td>1,500,000</td>
<td>15,000,000</td>
<td>150,000,000</td>
</tr>
<tr>
<td>PART</td>
<td>200,000</td>
<td>2,000,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>PARTSUPP</td>
<td>800,000</td>
<td>8,000,000</td>
<td>80,000,000</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>10,000</td>
<td>100,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>NATION</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>REGION</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

We run the queries sequentially and measure the geometric mean, arithmetic mean, median time (50th %ile), total time (sum) to run all the queries. We report geomean and total time in the final results as representative metrics.
Load Generation

This section describes where to find the benchmark code, its structure and steps to reproduce the benchmark.

The load generator code is available on GitHub in SnappyData repository. The test harness scripts are in the tests/benchmarks folder. The benchmark test suite(s) are part of the cluster project in the location cluster/src/test/scala/io/snappydata/benchmark.

In order to run the benchmark, build the SnappyData project(s) by following the build instructions in SnappyData documentation.
To summarise, run `./gradlew buildAll` to build all projects including test suites.

Steps to run the benchmark

The load generator harness is a simple set of shell scripts invoked sequentially. Some of these shell scripts call Spark/ Snappy jobs to perform tasks like stopping/ starting the cluster, cleaning up residual logs or temporary metadata, loading reference data in Spark cache/ Snappy tables, performing query on these tables/ cache etc.
Data Generation

The test data is generated using the dbgen tool provided by TPC. To generate TPC-H data, the dbgen program can be found in the TPC-H toolkit from TPC website.

The toolkit doesn't contain the binary for dbgen. The program can be compiled by generating a makefile (cp makefile.suite makefile) and editing the following properties:

```plaintext
CC = gcc
# Current values for DATABASE are: INFORMIX, DB2, TDAT (Teradata)
#   SQLSERVER, SYBASE, ORACLE
# Current values for MACHINE are: ATT, DOS, HP, IBM, ICL, MVS,
#   SGI, SUN, U2200, VMS, LINUX, WIN32
# Current values for WORKLOAD are: TPCH
DATABASE= ORACLE
MACHINE = LINUX
WORKLOAD = TPCH
```

The output of dbgen is CSV formatted files (with .tbl extension). These files are optionally converted to Parquet files as this format is supported by Spark for loading data at high performance.
Data Loading

To load data into Spark cluster, a Spark job reads these Parquet/CSV files and loads them in memory (Java heap and off-heap storage) as Spark dataframes. The load generator performs Spark SQL queries over these dataframes.

To load data into SnappyData cluster, a Snappy job (similar to a Spark job) reads the Parquet files and creates SnappyData tables. These tables support Persistence but reside in memory for our tests for SnappyData and any other products being compared (using Java heap and off-heap storage, utilizing the large amount of memory in the test environment). Note that SnappyData supports durability and data can spill over to disk if required.

Query Execution

In case of Apache Spark, the same job that loads data into Spark cache from Parquet/csv files also runs queries, since Spark cache is ephemeral and the data loaded in Spark cache is available only until the job is terminated.

In case of SnappyData, after the job responsible for data loading finishes, another job is run to perform queries on these tables. The tables are persistent (persistence is on by default) and the long running cluster holds the data in memory for the lifetime of the cluster.

Test Configuration

A configuration file PerfRun.conf contains the metadata about cluster like IP addresses of the machines involved so that the cluster can be created/destroyed.

It also contains several other options to control the size of the cluster, tune memory parameters etc. and other test parameters.

All experiment parameters for SnappyData are controlled through this file. By modifying this file and running the shell scripts provided, the configuration files for SnappyData are controlled. E.g. the cluster size and scale, various tuning parameters are varied.

In case of Apache Spark, the two configuration files $sparkHome/conf/spark-env.sh and $sparkHome/conf/spark-defaults.conf contain additional configuration parameters.

When setting up a Spark cluster in standalone mode, the following properties need to be set when multiple (say: 2) executors/worker instances are run per worker node:

```
SPARK_WORKER_INSTANCES=2
SPARK_WORKER_CORES=<n: number of physical cores by default>
```

The configuration file PerfRun.conf contains a number of parameters that are described here:

- Queries: list of queries to run - one can run all 22 queries or a subset
- sparkProperties: various spark parameters (refer an example file)
- sparkHome: home of spark cluster executables
- Master: Qualified name of the master host machine
- Slaves: list of slave machines - their IP addresses or names
- Client: client machine where the job is run
- NumberOfLoadStages=<int> The number of stages to load large data into system
- Parquet: (true|false) whether to use parquet files (or csv) to load data from
- IsDynamic: (true|false) whether to use dynamic query parameters controlled by a random seed
- ResultCollection: (true|false) whether results are to be collected for verification
- WarmupRuns: <int> Number of samples where the timing is not reported
- AverageRuns: <int> These values are reported - excludes WarmupRuns
- sparkSqlProperties: properties to be set in query session
- rePartition: (true|false) whether the data is to be repartitioned after loading
- buckets_Supplier, buckets_Supplier, buckets_Order_Lineitem, buckets_Cust_Part_PartSupp= <int> number of buckets to be used in SnappyData tables/ Parquet files generated
- TPCHJar: Path of the jar that contains the benchmark test jars: ../../cluster-tests.jar
- dataSize: (=1 GB| 10 GB| 100GB) the size of the data, depending on the scale factor
- dataDir: directory containing input data files
- outputLocation: directory where the results will be stored
- queryPlan: (true|false) whether to save query plans generated
- traceevents: (true|false) whether to record some low level trace events
- cacheTables: (true|false) whether tables are to be loaded into the Spark cache
- randomSeed: <int> the seed used for random number to generate dynamic query parameters
TPC-H Results for SnappyData 1.0.2

SnappyData supports all 22 queries in the TPC-H specification like Apache Spark SQL. The queries are run sequentially over multiple iterations and response time statistics like averages are computed. We measured performance for scale factor 100 (data size 100 GB).

Aggregate Statistics/ Highlights

The Geometric mean (a measure of centrality similar to median) of response time is 8x lower for SnappyData than Spark. Roughly speaking, SnappyData queries are 8x faster on average.

**GeoMean Query Response Time (SF100)**

(seconds - lower is better)

<table>
<thead>
<tr>
<th></th>
<th>Spark 2.1</th>
<th>Spark 2.3</th>
<th>SnappyData (1.0.2.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric mean</td>
<td>21.34</td>
<td>19.28</td>
<td>2.48</td>
</tr>
</tbody>
</table>

The total time to run the all 22 queries (sum of averages) is nearly 6x lower for SnappyData than Spark. i.e. SnappyData runs all 22 queries in sixth of the time Spark takes.

**Total Time to run all queries (SF100)**

(seconds - lower is better)

<table>
<thead>
<tr>
<th></th>
<th>Spark 2.1</th>
<th>Spark 2.3</th>
<th>SnappyData (1.0.2.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time to run all the queries (seconds)</td>
<td>821.43</td>
<td>755.20</td>
<td>113.73</td>
</tr>
<tr>
<td>Count of distinct queries run</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
</tbody>
</table>
Execution Time of TPC-H queries: Spark versus SnappyData

Let us drill down into the execution times of individual queries. As shown in the following graphs, SnappyData easily outperforms Apache Spark for all the queries by orders of magnitude. Many queries run 20x faster. Q6 runs 150x faster.

<table>
<thead>
<tr>
<th>Average (Mean) response time (seconds)</th>
<th>37.34</th>
<th>34.33</th>
<th>5.17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median response time (seconds)</td>
<td>34.12</td>
<td>28.83</td>
<td>2.71</td>
</tr>
<tr>
<td>GeoMean response time (seconds)</td>
<td>28.99</td>
<td>26.56</td>
<td>2.76</td>
</tr>
</tbody>
</table>

TPC-H Benchmark: Spark vs SnappyData 1.0.2.1
100 GB data (scale factor 100)
TPC-H Benchmark: Spark vs SnappyData 1.0.2

100 GB data (scale factor 100) cluster in Azure

Mean Response Time (seconds)
## Query Mean Response Time: SnappyData 1.0.2 vs Apache Spark

The following table lists the average (mean) response time for each of the 22 queries in the benchmark for all the systems tested.

<table>
<thead>
<tr>
<th>Query</th>
<th>Spark 2.1.1</th>
<th>Spark 2.3.1</th>
<th>SnappyData (1.0.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>6,198</td>
<td>4,938</td>
<td>1,932</td>
</tr>
<tr>
<td>Q2</td>
<td>6,886</td>
<td>8,299</td>
<td>1,262</td>
</tr>
<tr>
<td>Q3</td>
<td>37,932</td>
<td>46,738</td>
<td>4,659</td>
</tr>
<tr>
<td>Q4</td>
<td>26,747</td>
<td>33,649</td>
<td>4,660</td>
</tr>
<tr>
<td>Q5</td>
<td>59,729</td>
<td>34,255</td>
<td>3,127</td>
</tr>
<tr>
<td>Q6</td>
<td>10,913</td>
<td>18,922</td>
<td>200</td>
</tr>
<tr>
<td>Q7</td>
<td>43,829</td>
<td>48,665</td>
<td>6,637</td>
</tr>
<tr>
<td>Q8</td>
<td>73,480</td>
<td>36,383</td>
<td>1,703</td>
</tr>
<tr>
<td>Q9</td>
<td>87,067</td>
<td>60,784</td>
<td>5,051</td>
</tr>
<tr>
<td>Q10</td>
<td>29,253</td>
<td>34,753</td>
<td>1,155</td>
</tr>
<tr>
<td>Q11</td>
<td>7,542</td>
<td>7,136</td>
<td>1,956</td>
</tr>
<tr>
<td>Q12</td>
<td>23,012</td>
<td>30,008</td>
<td>431</td>
</tr>
<tr>
<td>Q13</td>
<td>11,247</td>
<td>10,622</td>
<td>8,655</td>
</tr>
<tr>
<td>Q14</td>
<td>11,013</td>
<td>13,138</td>
<td>543</td>
</tr>
<tr>
<td>Q15</td>
<td>22785</td>
<td>26683</td>
<td>1,307</td>
</tr>
<tr>
<td>Q16</td>
<td>30799</td>
<td>12855</td>
<td>1,581</td>
</tr>
<tr>
<td>Q17</td>
<td>69816</td>
<td>31917</td>
<td>12,330</td>
</tr>
<tr>
<td>Q18</td>
<td>39437</td>
<td>48883</td>
<td>11,730</td>
</tr>
<tr>
<td>Q19</td>
<td>4157</td>
<td>3957</td>
<td>929</td>
</tr>
<tr>
<td>Q20</td>
<td>18387</td>
<td>18972</td>
<td>4,454</td>
</tr>
<tr>
<td>Q21</td>
<td>60143</td>
<td>24909</td>
<td>23,576</td>
</tr>
<tr>
<td>Q22</td>
<td>3957</td>
<td>4028</td>
<td>2,451</td>
</tr>
</tbody>
</table>
Optimizations

How SnappyData Improves on Spark Performance

Apache Spark’s optimizations are designed for disparate data sources which tend to be mostly external, such as HDFS or Alluxio. For better response times to queries on a non-changing data sets, Spark recommends caching data from external data sources as cached tables in Spark. Then, they recommend running the queries on these cached data structures where the data is stored in optimized column formats. While this dramatically improves performance, we found a number of areas for further improvements. For instance, a scan of columns managed as byte arrays is copied into an "UnsafeRow" object for each row, and then the column values are read from this row breaking vectorization and introducing lots of expensive copying.

Addressing these inefficiencies, however, is not that easy as the data in the column store may have been compressed using a number of different algorithms like dictionary encoding, run length encoding etc. SnappyData has implemented alternate decoders for these, so it can get the full benefit of code generation and vector processing.

Significant optimizations in SnappyData include the following:

- SnappyData storage layer allows for collocation of partitioned tables. This information has been used to eliminate the most expensive portions of many joins (like shuffle) and, turned them into collocated one-to-one joins. This release significantly enhances the physical plan strategy phase to aggressively eliminate data shuffle and movement for many more cases.
- Support for plan caching to avoid query parsing, analysis, optimization, strategy and preparation phases.
- Spark has been changed to broadcast data with tasks themselves to significantly reduce task latencies. A set of tasks scheduled from the driver to an executor are grouped as a single TaskSet message with common task data sent only once instead of separate messages for each task. Task data is also compressed to further reduce network usage.
- An efficient pooled version of Kryo serializer has been added that is now used by default for data, closures and lower level netty messaging. This, together with improvements mentioned in the previous point, significantly reduce overall latency of short tasks (from close to 100ms down to a few ms). These also reduce the CPU consumed by the driver enhancing the concurrency of tasks, especially shorter ones.
- Enhancements in column level statistics allow for skipping column batches based on query predicates if possible. For example time range based queries will now be able to scan only the batches that fall in the said range and skip others, providing a huge boost to such queries on time-series data.
Alternate hash aggregation and hash join operators have been added that have been finely optimized for SnappyData storage to make full use of storage layout with vectorization, dictionary encoding to provide an order of magnitude performance advantage over Spark's default implementations.

Impact of SnappyData optimizations on TPC-H Performance

We cite examples of how the optimizations in SnappyData result in performance improvement in TPC-H.

The paper ‘TPC-H Analyzed…’ identifies 28 different such choke points, grouped into six categories: Aggregation Performance, Join Performance, Data Access Locality, Expression Calculation, Correlated Subqueries and Parallel Execution.

SnappyData optimisations responsible for speedup in queries are enumerated here:

- O1: Much more efficient column table scans compared to Spark caching (10-20X in single thread queries).
- O2: About 4-10X faster hash ‘group by’ aggregations. Likewise, faster hash joins with reference tables compared to hash broadcast join of Spark. SnappyData also add combining partial+final into a single hash aggregation when grouping keys are same/superset of table partitioning keys.
- O3: Collocation of tables to avoid shuffle (Exchange operator in SQL plans) in a large number of cases.
- O4: Use of hash joins in combination with colocated tables where possible which is much faster than sort merge join in Spark.
- O5: Better determination of estimated sizes to use broadcast more extensively than Spark.
- O6: Efficient handling of DATE string comparisons.

The following table conveys which optimisations help the queries.

Legend:

<table>
<thead>
<tr>
<th>Major benefit</th>
<th>Minor benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O4</th>
<th>O5</th>
<th>O6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Test Environment

All the tests were performed in Azure public cloud environment. A cluster of machines was set up in the same virtual network. These are memory optimised machines of instance type Standard E32s v3 (32 vcpus, 256 GB memory). Data is loaded into memory from Azure ‘Premium Storage’ SSDs attached to the virtual machines.
Each machine has 256 GB RAM and the products have been configured such that the entire TPC-H schema data resides in-memory for all the products.

Configuration : 4 Azure VMs of instance type/ size Standard E32s v3 (32 vcpus, 256 GB memory)
OS : RHEL 7
SnappyData cluster configuration: 1 lead, 1 locator on 1 azure instance, 3 or 6 server processes on 3 separate azure instances.  
Apache Spark: 1 Master node on 1 azure instance and and 3 or 6 executors on 3 separate azure instances as workers.

SUT (System Under Test) Configuration and Tuning

Product configuration/ tuning options for SnappyData/ Spark are as specified in the test harness scripts.

There is ample memory available in commodity machines in the cloud nowadays.

To utilize the 256 GB RAM on the machines in our cluster, it is possible to run a single worker/ server on each node with high memory (scale up), or scale out to multiple workers with relatively smaller memory.

The following configurations were used for Apache Spark:

1. One executor (worker instances) on each worker node: The executor JVM was provided with 40 GB heap and 100 GB off-heap memory.
2. Two executors (worker instances) on each worker node: (we saw the best results with this configuration): Each executor was provided with 40 GB heap and 32 GB off-heap memory.

We obtained the best results for Apache Spark with 40 GB heap, though some reports suggest using a smaller heap. E.g. Q3 runs about 10 times slower with a smaller heap like 16 GB or 24 GB.

For SnappyData, the following memory configurations were used for the server processes:

1. One server (executor) on each worker node: 16-to-40 GB heap and 100 GB off-heap memory
2. Two servers (executors) on each worker node: Each having 16-32 GB heap and 32-80 GB off-heap memory

The second configuration (two servers per node) is more performant in our test environment. (The optimal value of worker instances per node for a given environment may vary depending on the number of CPU cores/ processors/ NUMA nodes available etc.) Scaling out to more nodes in a cluster also reduces the cost of rework if any tasks in the Spark jobs fail and get re-submitted. Also, running with a heap of 16 GB results in less GC than a very large heap running into 100s of GB. Instead, off-heap storage provided by SnappyData enterprise is highly performant.

In either case, the lead component runs fine with 16 GB heap (no off-heap memory is required/ useful on the lead). Similarly, a heap of 12 GB was sufficient for the Spark driver.
In addition, we tuned the parallelism settings for Spark for the long-running TPC-H queries. We increased the spark.executors.cores value to 48 from 32; and (increased) the locality.wait and locality.wait.process values to 30 seconds (from the default value of 3 seconds). Without these settings, Spark performed far worse with large outliers (also resulting in very high variance) since straggler tasks got reassigned to other nodes on the cluster, resulting in data shuffling/exchanges and further slowdown due to underutilization of CPU. SnappyData too can benefit from these settings. However, it performs well with default settings in this area. By default, SnappyData employes spark.executor.cores = 2 * physical CPU cores. It is a reasonable default and works well for the TPC-H workload. (SnappyData also has a separate low latency pool for point lookup queries so that analytical queries won’t consume all the CPU power).

Overall, we tried to optimize/tune both SnappyData and Apache Spark to the best of our knowledge. We spent a significant chunk of this effort in tuning Apache Spark.
Summary

SnappyData continues to make positive strides as a high performance product, offering a low-latency, high-throughput data platform. SnappyData 1.0.2 release includes several bug fixes and features besides improved performance.

Stay tuned for our results with a concurrent workload consisting of TPC-H analytical queries and optionally additional point-lookup queries.

References

1. The TPC-H benchmark specification can be downloaded from this section on the TPC (Transaction Processing Council) website.
2. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark: [paper, slides]
3. [SnappyData: A unified cluster for streaming, transactions, and interactive analytics](CIDR 2017)
4. SnappyData blog
   a. [JOINING A BILLION ROWS 20X FASTER THAN APACHE SPARK]
   b. [SNAPPYDATA, MEMSQL-SPARK & CASSANDRA-SPARK: A PERFORMANCE BENCHMARK]
   c. [RUNNING SPARK SQL CERN QUERIES 5X FASTER ON SNAPPYDATA]
   d. [HOW MUTABLE DATAFRAMES IMPROVE JOIN PERFORMANCE IN SPARK SQL]