GridDB

Data Model Comparison

Version 1.1

February 5, 2024

Fixstars Solutions, Inc.

Table of Contents

Executive Summary

In this comparison paper, four different data models were examined with GridDB to determine their relative weaknesses and strengths in ingest and query performance, storage requirements, along with other factors such as ease of development and flexibility.

Overview

To compare the four different data models, TSBS's (https://github.com/timescale/tsbs) IoT data set was used which mimics time series data generated by a fleet of trucks. The data includes not only plain sensor data such as location and velocity but also diagnostic data such as fuel level status and further meta data such as driver name, fleet name, and truck type.

Data Models

The four data models selected for comparison were:

MultiWide
- One table per truck where the data values were stored in individual columns.

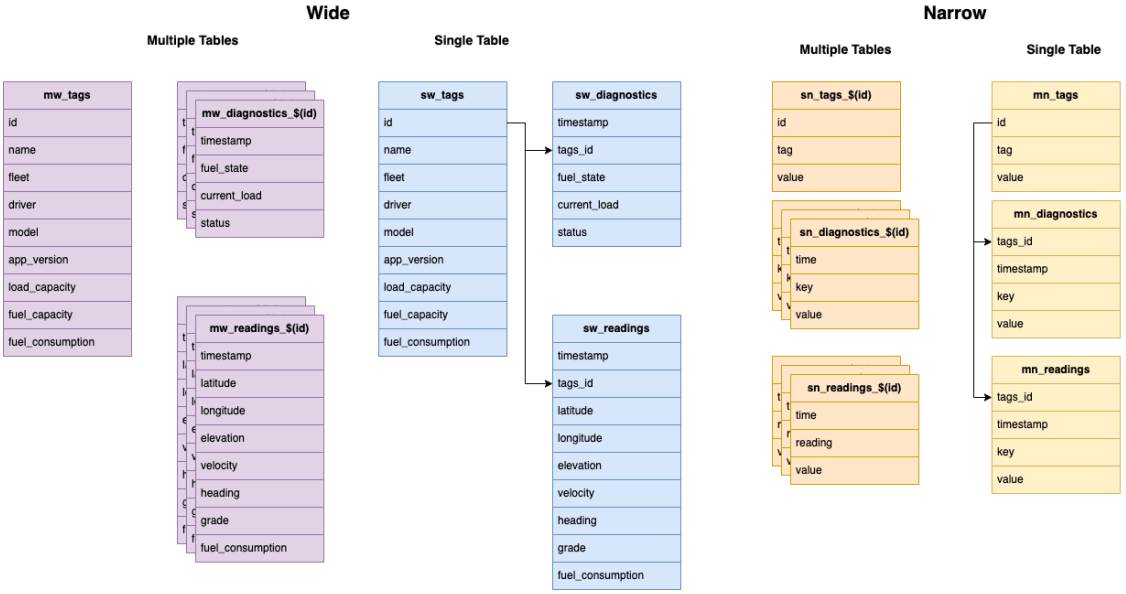
SingleWide
- One table for all trucks where the data values were stored in individual columns.

MultiNarrow
- One table per truck where the data values were stored as key-value pairs.

SingleNarrow
- One table for all trucks where the data values were stored as key-value pairs.

The primary advantage of the Narrow data models is that any key value pair can be stored while Wide data models are limited to the fixed column names. It was assumed that the fixed Wide data models would have better performance and we looked to see how much the Narrow data model's flexibility gave up performance wise.

While GridDB has two different container types, time series for temporal indexed data and collections for everything else, only MultiWide uses time series containers for the reading and diagnostic tables due unique key constraints on the timestamp column. Multiple trucks can use the same timestamp in SingleWide multiple tags use the same timestamp in MultiNarrow while both conditions are true in SingleNarrow.



Data Sample

The following tables show a sampling of the data stored in each table.

MultiWide

mw_tags

	id	name	fleet	driver	model	app_version	load_capacity	fuel_capacity	fuel_consumption
0	6	truck_6	South	Derek	F-150	v2.0	2000.0	200.0	15.0
1	7	truck_7	East	Rodney	G-2000	v2.3	5000.0	300.0	19.0
2	0	truck_0	North	Derek	F-150	v1.0	2000.0	200.0	15.0

mw_readings_2

	timestamp	latitude	longitude	elevation	velocity	heading	grade	fuel_consumption
0	2022-12-31 16:00:00	-52.933569	112.685841	79.461181	70.014057	33.416775	3.870407	4.935967
1	2022-12-31 16:00:10	-52.932112	112.687435	13.127632	19.916423	154.641063	-19.000868	0.581635
2	2022-12-31 16:00:20	-52.932537	112.687769	18.012328	0.121823	316.538792	-3.405608	4.258235

mw_diagnostics_7

	timestamp	fuel_state	current_load	status
0	2022-12-31 16:00:00	39.463768	0.155640	0.0
1	2022-12-31 16:00:10	215.797361	0.909274	0.0
2	2022-12-31 16:00:20	127.681254	0.549360	0.0

SingleWide

sw_tags

	id	name	fleet	driver	model	app_version	load_capacity	fuel_capacity	fuel_consumption
0	2	truck_2	South	Albert	H-2	v2.0	1500.0	150.0	12.0
1	0	truck_0	North	Derek	F-150	v1.0	2000.0	200.0	15.0
2	3	truck_3	East	Andy	F-150	v2.3	2000.0	200.0	15.0

sw_readings

	id	timestamp	latitude	longitude	elevation	velocity	heading	grade	fuel_consumption
0	3	2022-12-31 16:00:00	-39.061459	119.000323	17.258688	95.669745	295.455239	-13.457177	0.034694
1	3	2022-12-31 16:00:10	-39.060460	118.997621	61.844788	2.127305	324.154888	1.710904	6.685572
2	3	2022-12-31 16:00:20	-39.060417	118.997581	0.176427	31.941456	197.098320	-0.908386	13.018863

sw_diagnostics

	id	timestamp	fuel_state	current_load	status
0	5	2022-12-31 16:00:00	89.339224	0.513903	0.0
1	5	2022-12-31 16:00:10	36.956331	0.909743	0.0
2	5	2022-12-31 16:00:20	134.814962	0.548199	1.0

MultiNarrow

mn_tags

	id	tag	strVal	dblVal
0	2	name	truck_2	0.0
1	2	fleet	South	0.0
2	2	driver	Albert	0.0

mn_readings_6

	timestamp	key	value
0	2022-12-31 16:00:00	latitude	-83.629383
1	2022-12-31 16:00:00	longitude	-105.748127
2	2022-12-31 16:00:00	elevation	53.728022

mn_diagnostics_3

	timestamp	key	value
0	2022-12-31 16:00:00	fuel_state	190.417241
1	2022-12-31 16:00:00	current_load	0.231459
2	2022-12-31 16:00:00	status	0.000000

SingleNarrow

sn_tags

	id	tag	strVal	dblVal
0	4	name	truck_4	0.0
1	4	fleet	North	0.0
2	4	driver	Seth	0.0

sn_readings

	timestamp	tags_id	key	value
0	2022-12-31 16:00:00	2	latitude	-5.810646
1	2022-12-31 16:00:00	2	longitude	-102.811993
2	2022-12-31 16:00:00	2	elevation	21.859351

sn_diagnostics

	timestamp	tags_id	key	value
0	2022-12-31 16:00:00	6	fuel_state	106.228604
1	2022-12-31 16:00:00	6	current_load	0.906214
2	2022-12-31 16:00:00	6	status	1.000000

GridDB

GridDB is an open source time series database optimized for IoT and Big Data with NoSQL and SQL interfaces. GridDB's hybrid composition of in memory and Disk storage architecture is designed for maximum performance while its unique Key-Container data model represents data in the form of collections that are referenced by keys making it ideal for applications with large amounts of heterogeneous data.

Execution Environment

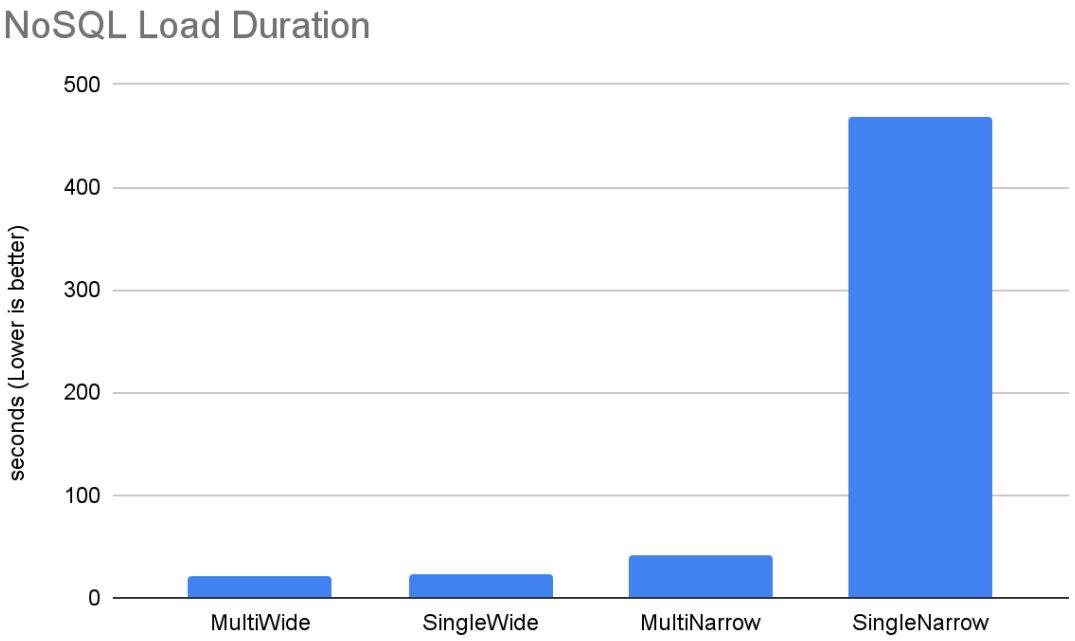
The benchmarks were implemented using Java using the GridDB NoSQL interface for data ingestion and both the GridDB NoSQL (TQL) and JDBC interface (SQL) for querying. They were run on an Azure Standard D4s v3 Virtual Machine using Ubuntu 20.04 with GridDB version 5.3.

Benchmark Configuration

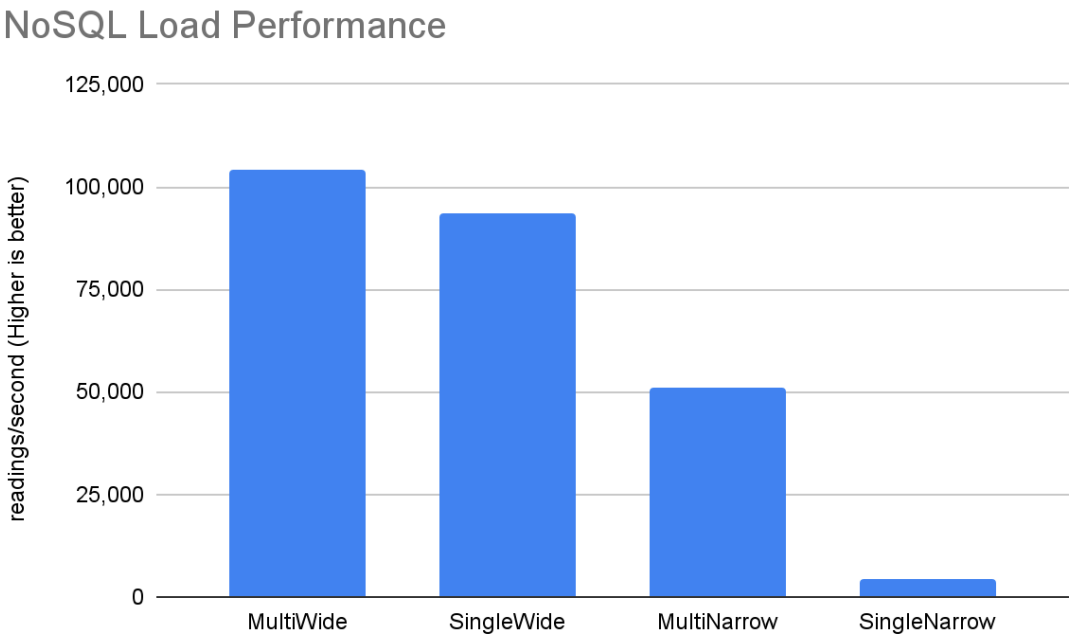
The performance comparison generated data for eight trucks over a three month period in ten second increments. Each truck would have a total of 267,840 readings for a total of 2,142,720 readings.

Ingest/Load Performance

To load data into GridDB, the application used 8 threads (1 per truck) to insert batches of 1000 readings using GridDB's NoSQL interface which is significantly faster for ingestion than the SQL interface as the SQL interface does not support batch insert in version 5.3.



MultiWide took the least amount of time to ingest all 267,840 readings with small increases for SingleWide and MultiNarrow data models while the SingleNarrow data model took significantly longer. Calculating ingestion rates shows the differences further.



The MultiWide schema inserted 104,208 readings per second while the SingleWide schema inserted 93,511 readings per second. The performance dropped considerably for the Narrow schemas due to them inserting 6x the number of rows.

```
sw_tags
  id     name  fleet driver  model app_version  load_capacity  fuel_capacity  fuel_consumption
0   0  truck_0  North  Derek  F-150        v1.0         2000.0      200.0        15.0


sw_readings
  id    timestamp   latitude  longitude  elevation velocity  heading  grade  fuel_consumption
0   0 2022-12-31 16:00:00  45.998  94.853  97.0833  86.503475  205.98 -3.76     9.0222


sw_diagnostics
  id           timestamp  fuel_state  current_load  status
0   0 2022-12-31 16:00:00 173.847796      0.340449     1.0
```

If we compare the writes made for one reading with SingleWide and Single Narrow we see that SingleWide writes 3 rows while SingleNarrow writes 18 rows.

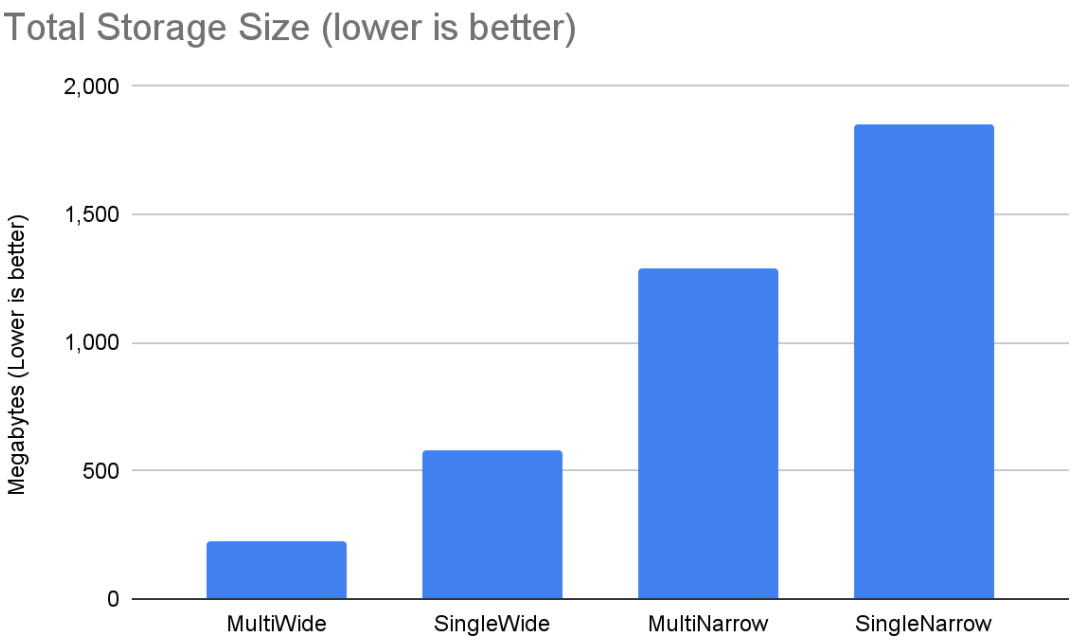
```
sn_tags
  id               tag   strVal  dblVal
0   0              name  truck_0     0.0
1   0             fleet    North     0.0
2   0            driver    Derek     0.0
3   0             model    F-150     0.0
4   0    device_version     v1.0     0.0
5   0     load_capacity     none  2000.0
6   0     fuel_capacity     none   200.0
7   0  fuel_consumption     none    15.0


sn_readings
           timestamp  tags_id              key       value
0 2022-12-31 16:00:00        0         latitude   62.242578
1 2022-12-31 16:00:00        0        longitude  -75.184678
2 2022-12-31 16:00:00        0        elevation   44.387040
3 2022-12-31 16:00:00        0         velocity   66.024441
4 2022-12-31 16:00:00        0          heading  321.052663
5 2022-12-31 16:00:00        0            grade  -10.488837
6 2022-12-31 16:00:00        0  fuel_consumption    2.861151


sn_diagnostics
           timestamp  tags_id           key       value
0 2022-12-31 16:00:00        0    fuel_state  190.683612
1 2022-12-31 16:00:00        0  current_load    0.004060
2 2022-12-31 16:00:00        0        status    1.000000
```

MultiNarrow inserted 51,209 readings per second and SingleNarrow inserted just 4578 readings per second. The large drop in performance with SingleNarrow is likely due to the increase of overhead of not only inserting more rows but also having to build multiple indexes for each row.

Data Storage Size

After the data was loaded, the total storage size was examined with the gs_stat tool. storeTotalMem increased dramatically as the number of indexes and number of rows inserted

increased with MultiWide requiring the least amount of storage and SingleNarrow requiring the most.

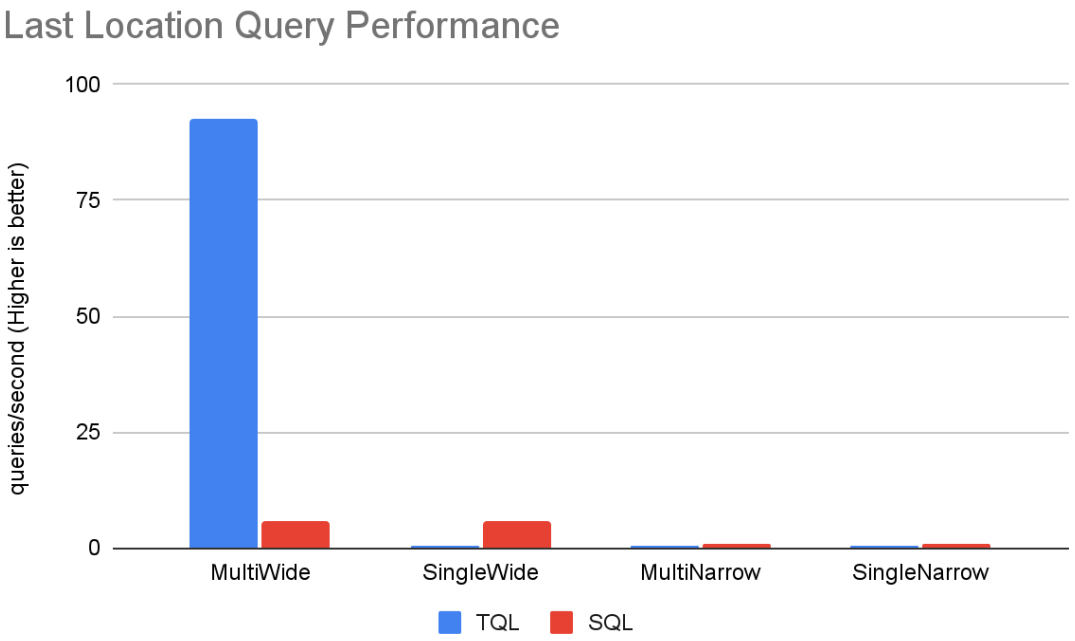


Query Performance

For each query type, queries were built using GridDB's NoSQL interface (TQL) and GridDB's JDBC interface (SQL) to see how not only each data model performed but the differences in performance

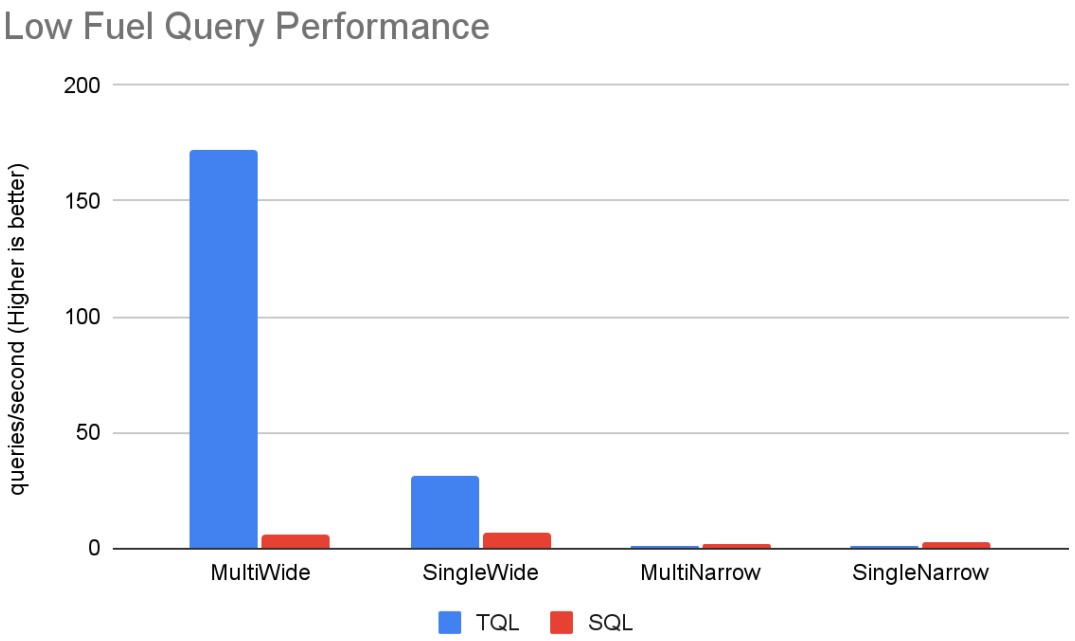
Last Location

The Last Location query fetched the most recently recorded location of each truck. Using the `time_prev(*, now())` TQL time series function that was only suitable for the MultiWide query as it is the only data model using time series containers. Using time_prev showed remarkable performance while all data model schemas required the use of some variation of `order by timestamp desc limit 1` query which does not perform as well.



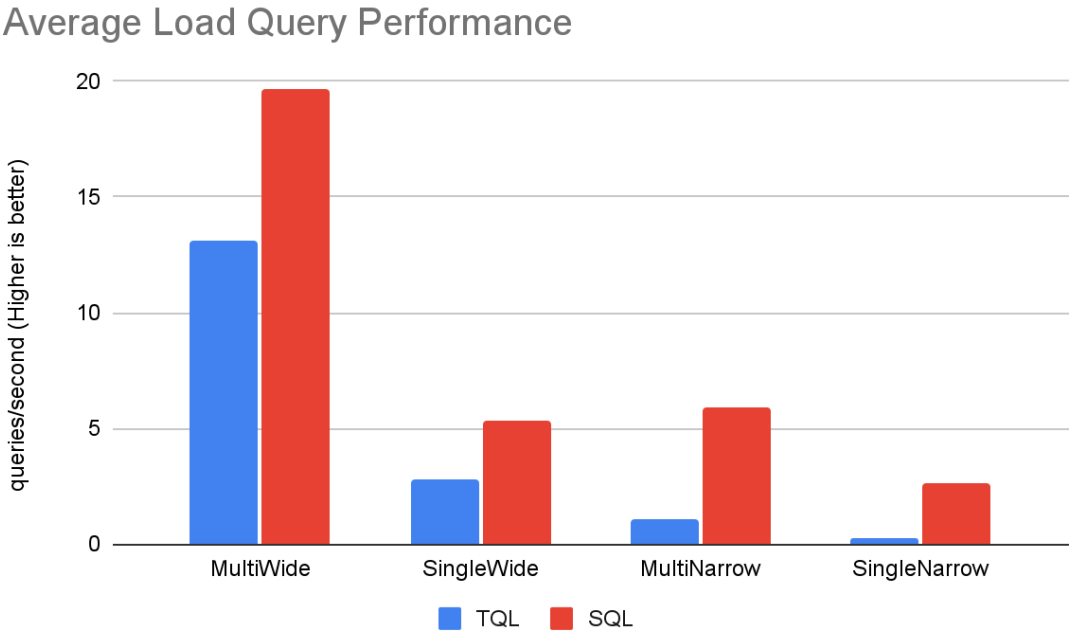
Low Fuel

The Low Fuel query fetched all trucks that currently had less than 10% of their total fuel capacity. Like Last Location, the low fuel query used time_prev(*, now()) for MultiWide while all other queries once again used `order by timestamp desc limit 1.`



Average Load

Average Load calculates the average load per fleet where there are four fleets. In this query, Single table data models have relatively better performance compared to queries whose result only pertain to a single truck as only one table would need to be queried for each fleet rather than querying individual trucks and computing the average.

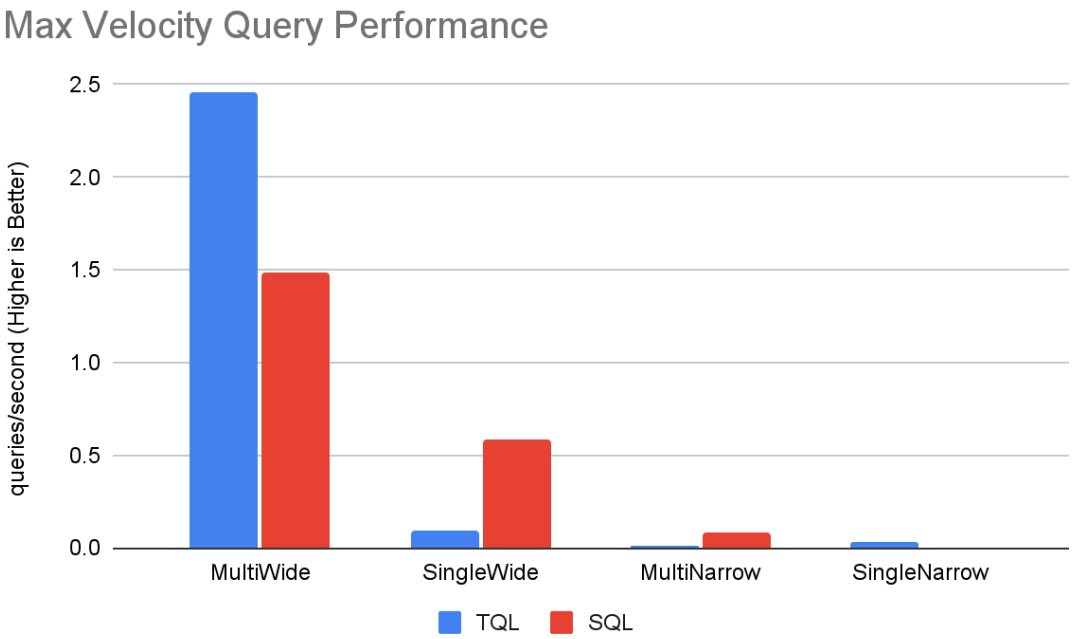


Max Daily Velocity

Max Daily Velocity finds the maximum daily velocity for every day over three months for every truck. In this query, the SQL group by range functionality is used while TQL needs to perform multiple queries for every individual day. The MultiWide data model still performs the best, but the SQL query for SingleWide also performs relatively better than its TQL counterpart while the Narrow data models still have poor performance.

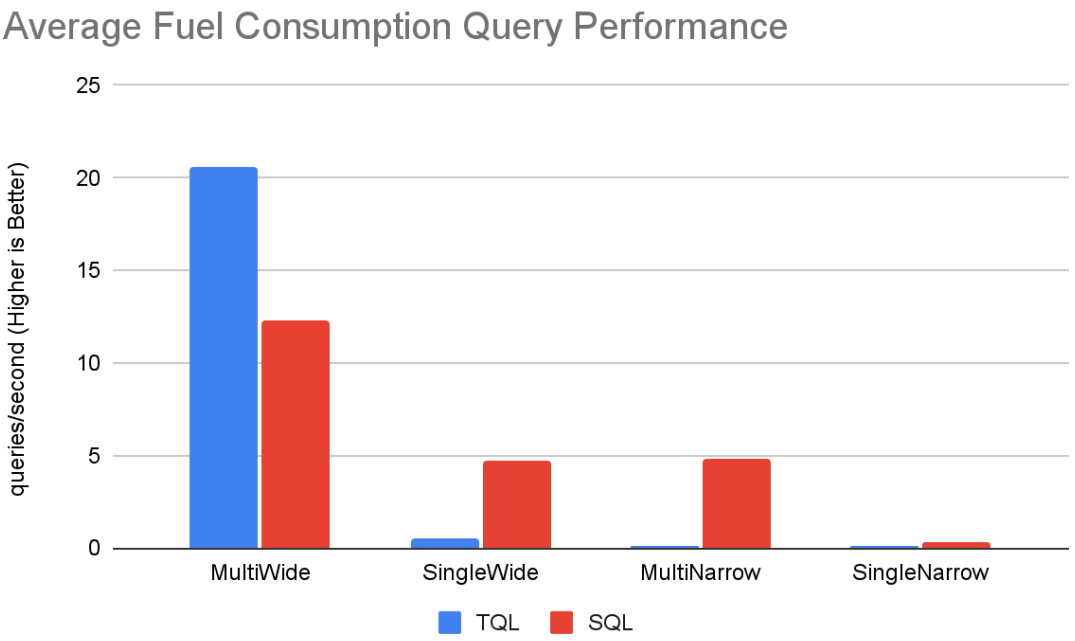
To explain why TQL and SQL relative performance can differ when switching data models we need to consider the time required for sort processing. The difference was noticeable due to the SQL queries being used. As far as data reading processing is concerned, TQL and SQL are roughly the same but as there is no need to specify acquisition conditions, it can be done efficiently using TQL. Moreover, if the query is of a scale that can be processed in 10ms to 100ms it is likely more suitable to use TQL.

Since the query is executed by specifying additional conditions that are not available in MultiWide, data read time is important. The difference between collections and time series container types will also affect performance greatly depending on whether or not an index is used and which index is used as the choice depends on how the search conditions are written in the query. The difference may become smaller if you combine SQL, TQL, index usage, etc. However, SQL is relatively better suited for processing complex conditions and large amounts of data in a single statement such as using join statements.



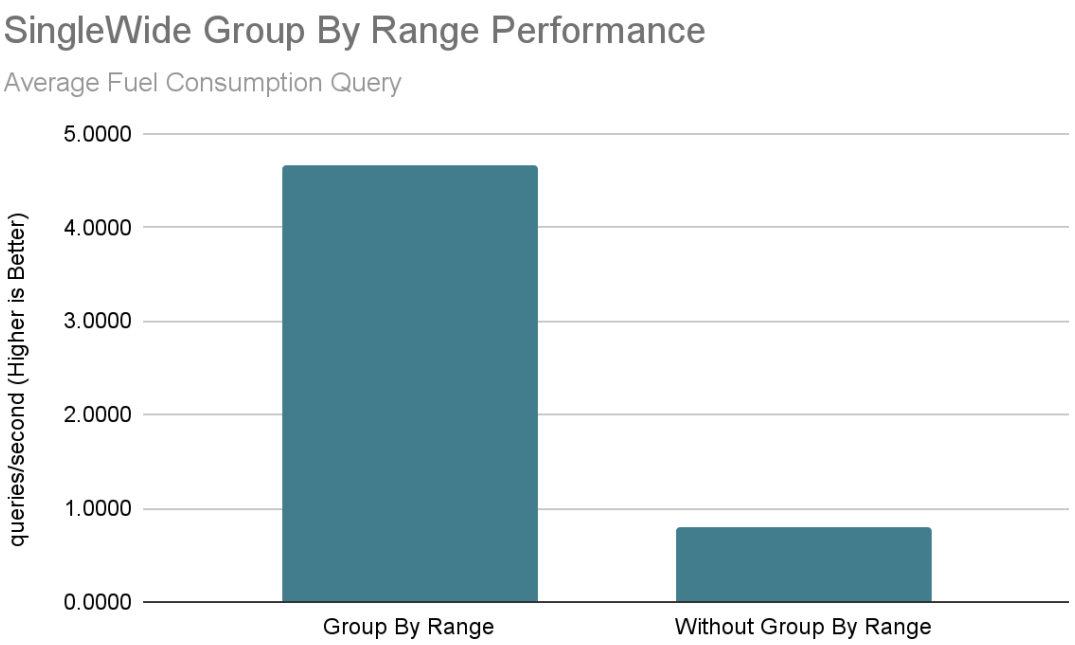
Average Daily Fuel Consumption

Like Max Daily Velocity, Average Daily Fuel Consumption calculates an aggregate for every day in the three month time range but instead of performing the calculation for every truck, it performs it for a single truck. While MultiWide is still the fastest, MultiNarrow's performance improves to be greater than that of SingleWide.



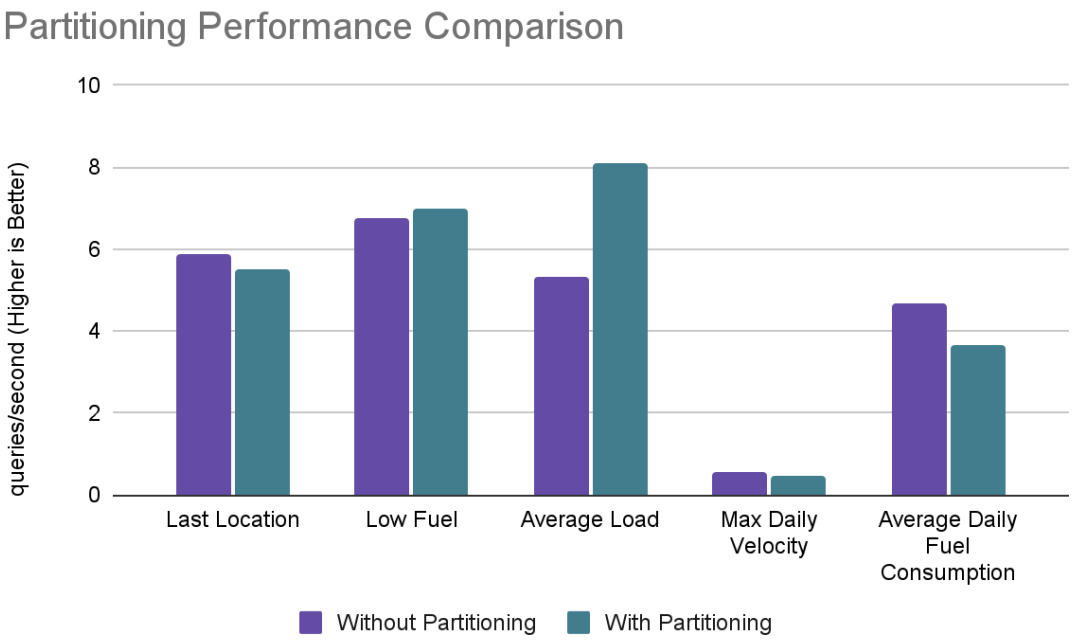
Time Bucketing vs. Group By Range

In this comparison, we compared using the SQL group by range feature versus executing multiple SQL queries for each day in the time range using the SingleWide data model. The Average Fuel Consumption query was used as it meant only query would need to be executed versus a query per truck with the MultiWide data model. SQL was used for both queries to remove the differences between SQL and TQL execution. Group By Range performs considerably better as expected except for the MultiWide data model. This is likely because with SQL, the query is executed by specifying additional conditions that are not available in MultiWide, data read time is important. As well as the difference between collections and time series, performance also changes greatly depending on whether or not an index is used and which index is used (the choice depends on how the search conditions are written).



Partitioning

In this comparison, the effect of GridDB's single table partitioning function (https://www.toshiba-sol.co.jp/en/pro/griddb/docs-en/v4_0_3/GridDB_TechnicalReference.html#sec-4.3.8) was examined using the SingleWide data model. Partitioning is only significantly better for Average Load which is likely caused by the entire data set being aggregated versus only fetching the last points in the Last Location and Low Fuel queries. Max Daily Velocity and Average Daily Fuel Consumption use group by range which appears to have a negative impact on performance for unknown reasons.



Other Considerations

The primary consideration not related to performance is the difficulty in writing individual queries.

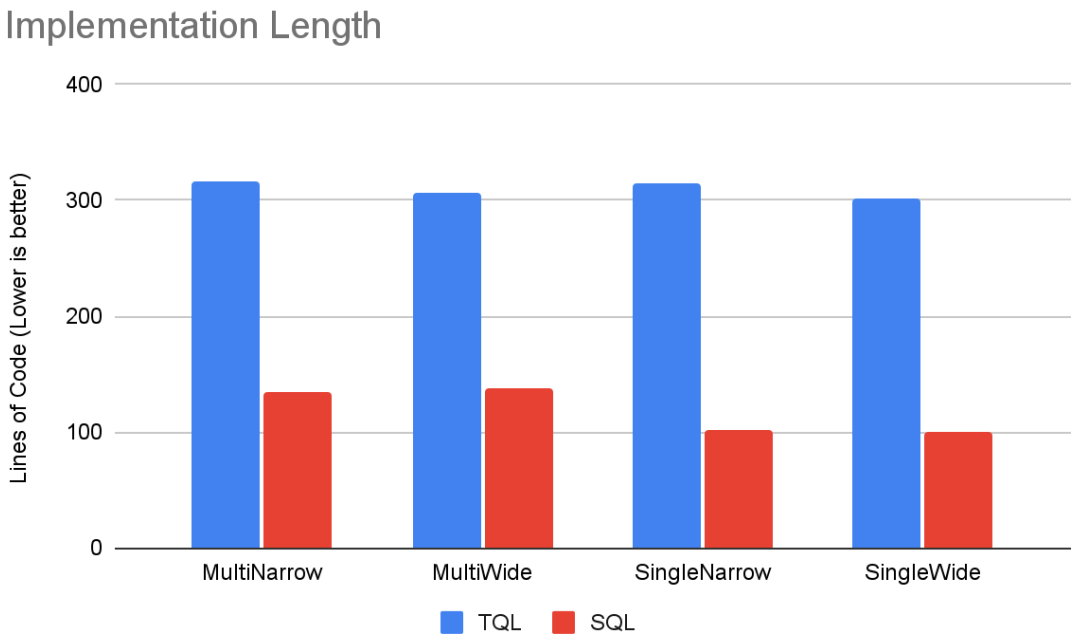
TQL moves most of the logic to the application programming language while the SQL queries themselves can be slightly more complex. Thus for TQL, the Implementation (queries plus supporting Java code) itself is longer but the queries themselves are shorter. To show the difference, here is the code for the SingleWide model's TQL and SQL functions.

TQL Average Load Query:

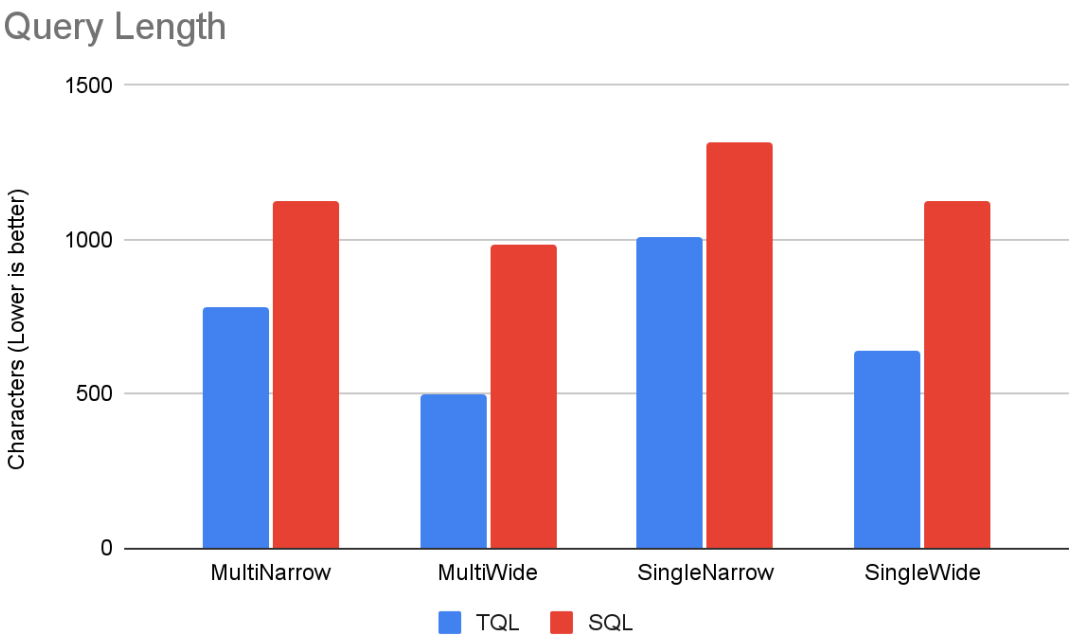
```
public void avgLoad()  throws GSException {
    ArrayList<SWTags> trucks = getTrucks("");
    HashMap<String, Double> sums = new HashMap();
    HashMap<String, Integer> counts = new HashMap();
    List<com.toshiba.mwcloud.gs.Query<AggregationResult>> queryList = new ArrayList();
    for(SWTags truck : trucks) {
            Container<Integer, Row> container = store.getContainer("sw_diagnostics");
            if ( container == null ){
                System.err.println("Container sw_diagnostics not found.");
            }
            queryList.add(container.query("select avg(current_load) where id="+truck.id,
                AggregationResult.class));
    }
    store.fetchAll(queryList);
    for (int i = 0; i < queryList.size(); i++) {
            SWTags truck = trucks.get(i);
            String fleet = truck.fleet;
            com.toshiba.mwcloud.gs.Query<AggregationResult> query = queryList.get(i);
            RowSet<AggregationResult> rs = query.getRowSet();
            if (rs.hasNext()) {
                AggregationResult row = rs.next();
                double avg = row.getDouble();
                if (sums.get(fleet) == null)
                    sums.put(fleet, avg);
                else
                    sums.put(fleet, avg+sums.get(fleet));
                if (counts.get(fleet) == null)
                    counts.put(fleet, 1);
                else
                    counts.put(fleet, 1+counts.get(fleet));
            }
        }
    for (String fleet : sums.keySet()) {
            Double avg = sums.get(fleet)  / counts.get(fleet);
            if(first)
                System.out.println(fleet+": "+avg);
    }
}
```

SQL Average Load Query:

```
    public void avgLoad()  throws SQLException {
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery("select * from sw_tags t inner join ( select id,
avg(current_load) from  sw_diagnostics group by id ) d where d.id = t.id group by
fleet");
        ResultSetMetaData md = rs.getMetaData();
        while (rs.next()) {
            if (PRINT_DATA) {
                for (int i = 0; i < md.getColumnCount(); i++) {
                    System.out.print(rs.getString(i + 1) + "|");
                }
                System.out.println("");
            }
        }
        rs.close();
        st.close();
    }
```



TQL requires three times the amount of Java code to implement the five queries (above) while the queries themselves are somewhat longer with SQL.



The difficulty in creating either set of queries is subjective and depends on the developers skills and preference.

Conclusion

Each of the data models have their own set of pros and cons, but the MultiWide data schema performed the best in all performance comparisons, as expected. The TQL queries, on the other hand, usually perform better than SQL, but using SQL, and especially the group by range functionality, can have query performance benefits while the lower weight TQL interface is always better for writing. Of course, an application can utilize multiple data models so each component uses the data model with the ideal trade offs.

If migrating from a conventional SQL database with single tables, SingleWide still performs adequately for applications that cannot be implemented otherwise. The performance advantage of MultiWide, which achieves performance through parallel processing on the applications side, may not provide enough benefit to re-implement applications using TQL. Single tables versus multiple tables have the primary benefit of being able to use SQL JOINs.

The flexibility a Narrow schema provides is rarely worth the performance loss and increased storage as compared to a Wide schema, especially considering columns can easily be added, removed or modified. Narrow should be considered if the data collected as a disjoint set of value keys or the keys may change often such as when building a software monitoring system.

Finally, using the NoSQL interface does move some computational load to the application instead of the database, which may be beneficial when scaling as it can be easier to scale stateless application servers rather than stateful database servers. However, SQL is relatively better suited for processing complex conditions and large amounts of data in a single statement such as joins or subqueries while TQL would require multiple queries and in-application data processing to do the same. Thus, SQL is recommended unless extremely low latency is required.

While we were trying to mimic particular use cases, this may deviate depending on the query frequency and patterns of the entire system. The source code for the performance comparisons is available on GridDB.net's GitHub here: https://github.com/griddbnet/data-model-comparison

Appendixes

Load/Query Performance Raw Data

	MW TQL	MW SQL	SW TQL	SW SQL
	Multiple Queries using TS Container	Multiple Queries using TS Container	Multiple Queries using Collection	Joins/Sub Queries using Collection
NoSQL Load (records/sec, higher is better)	104207.76		93511.39	
NoSQL Load (seconds, lower is better)	20.56		22.91	
storeTotalUsed (MB, lower is better)	226.19		576.88	
last-loc (queries/sec, higher is better)	92.59	5.94	0.05	5.89
low-fuel (queries/sec, higher is better)	172.41	6.21	31.55	6.78
avg-load (queries/sec, higher is better)	13.12	19.69	2.75	5.31
n-max-daily-velocity (queries/sec, higher is better)	2.46	1.48	0.09	0.58
1-avg-daily-fuel-consumption (queries/sec, higher is better)	20.62	12.25	0.51	4.67
	MN TQL	MN SQL	SN TQL	SN SQL
	Multiple Queries using TS Container	Multiple Queries using TS Container	Multiple Queries using Collection	Single query using Collection
NoSQL Load (records/sec, higher is better)	51208.57		4577.64	
NoSQL Load (seconds, lower is better)	41.84		468.08	
storeTotalUsed (MB, lower is better)	1290.06		1851.44	
last-loc (queries/sec, higher is better)	0.03	0.86	0.02	0.74
low-fuel (queries/sec, higher is better)	0.06	1.97	0.05	2.60
avg-load (queries/sec, higher is better)	1.06	5.91	0.29	2.63
n-max-daily-velocity (queries/sec, higher is better)	0.02	0.08	0.03	0.00
1-avg-daily-fuel-consumption (queries/sec, higher is better)	0.14	4.77	0.08	0.31

Group By Range Comparison Raw Data

Group By Range	4.6729
Without Group By Range	0.7976

Partitioning Comparison Raw Data

	With Partitioning	Without Partitioning
Last Location	5.530973451	5.889
Low Fuel	7.007708479	6.780
Average Load	8.110300081	5.311
Max Daily Velocity	0.457101065	0.580
Average Daily Fuel Consumption	3.661662395	4.673

Implementation Length Raw Data

The number of lines of code needed to implement each data model's query:

	Lines of Code	
Query	**SQL**	**TQL**
MultiNarrow	135	317
MultiWide	137	306
SingleNarrow	102	315
SingleWide	100	301

The number of characters used in the executed queries used to implement the compared query for each data model:

	Characters	
	SQL	**TQL**
MultiNarrow	1124	781
MultiWide	985	500
SingleNarrow	1314	1011
SingleWide	1125	637