Benchmarking a Sensor Billing Application using GridDB and MariaDB.

October 3, 2018

Revision 1.0

Table of Contents

Table of Contents 2

Executive Summary 3

Introduction 3

Environment 4

- Azure Configuration 4
- Software Version and Configuration 4

Test Methodology 5

- Test Design 5
  - GridDB Schema 5
  - SQL Schema 6
- Methodology 6

Benchmark Results 7

- Ingest 7
- Extract 8
- Resource Usage 8
  - Load Average 9
  - Memory Usage 10
- Tabular Results 11

Conclusion 12

Appendices 13

- SQL Schema 13
- GridDB Schema 14

Executive Summary

A common use case for GridDB is using sensor data to perform billing functions. This whitepaper builds a sample IoT billing application benchmark that can be used to compare the performance of GridDB against a relational database management system (RDBMS). MariaDB was chosen as the RDBMS due to its popularity and free and open source nature.

While relational databases have been the traditional choice for building such applications but as the results demonstrate, MariaDB's performance suffers as the number of records grow. One reason for this is that rather than storing all sensor read records in one table, GridDB utilizes containers. Each GridDB container only holds data for a particular device, which allows for improved and more consistent query times as the number of devices is increased.

Introduction

NoSQL databases were developed to overcome the performance and scalability issues that organizations faced as their datasets grew into the category of "BigData". It is exceedingly difficult for a typical relational database managed system (RDBMS) to scale out or to add additional computational nodes to increase performance. Instead, they require their administrators to scale up by adding more CPU cores and memory to their systems.

GridDB is developed by Toshiba Digital Solutions Corporation and can be used as an in-memory database, or as a hybrid composition. GridDB has many features, including a unique Key-Container model which may utilize any-key Collections or specialized TimeSeries containers.

MariaDB is a free and open source relational SQL database that is a derivative of MySQL. MySQL is the world's most popular[1] open source database and since it's introduction MariaDB has supplanted MySQL in many Linux distributions.

[1] https://db-engines.com/en/ranking

Environment

Azure Configuration

The benchmarks were performed using Microsoft Azure virtual machines in the WestUS region. There is one database server, a B2ms instance and a mix of three B2ms and two A2 client instances.

The B2ms instances have dual core Intel(R) Xeon(R) E5-2673 v3 running at 2.40GHz and 4GB of memory while A2 instances have a dual -core Intel(R) Xeon(R) E5-2660 CPU and 3.5GB of memory.

Software Version and Configuration

All instances are based on the RogueWave Software CentOS 7.5 image using Java 1.8.0 update 181.

GridDB 4.0.0 Community Edition was installed on both Azure servers using RPM packages downloaded from https://griddb.net. MariaDB version 5.5.56 was installed on the database server via CentOS's default YUM repositories. MySQL Connector 8.0.11 was used to create a JDBC connection to MariaDB.

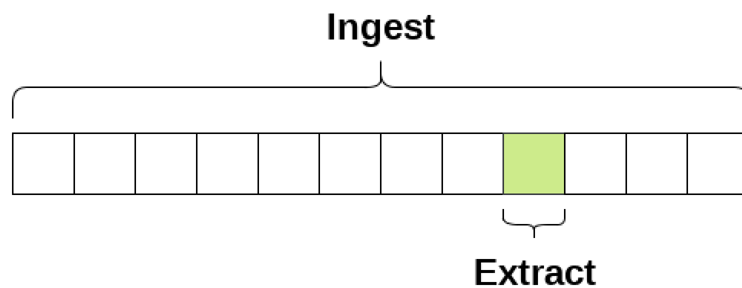
GridDB was configured to have a storeMemoryLimit of 1024MB (the amount of memory it will use for cached data) and a concurrency level of 4. MariaDB used the default configuration with the exception of increasing the max_connection_count to 320.

Test Methodology

Test Design

The goal of this testing is to fairly compare GridDB and a relational database in a "real world use case" where sensor data is collected from IoT devices or other sources and in the second, an aggregation function is used to provide a billing amount.

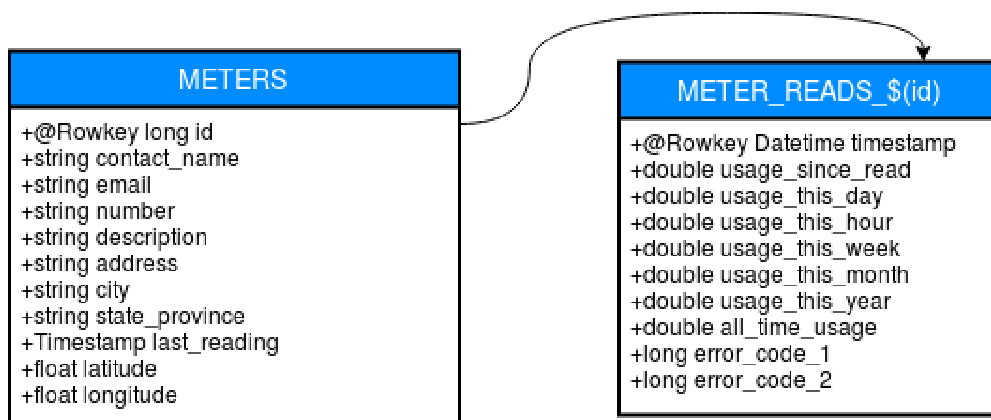
The application has been separated into two test cases: the first testcase is a load or ingestion test that shows the performance of updating and writing new records to the database; the second test is an extraction or aggregation test which performs an aggregation on a subset of the data generated in the load phase.



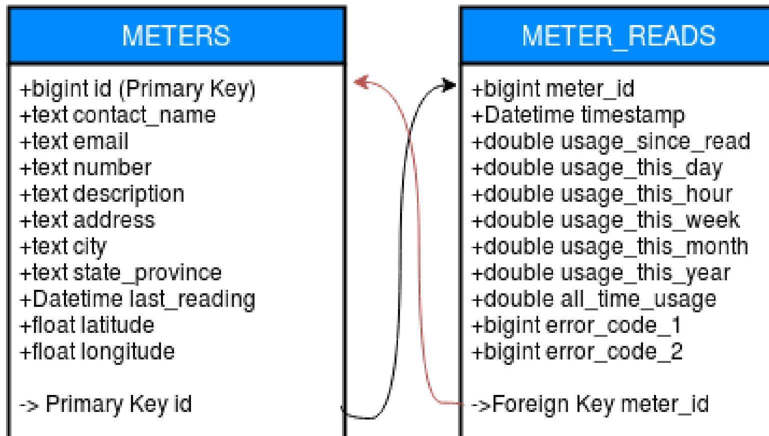
The Ingest test performs the ingest operations as quickly as possible where each thread is responsible for one device. Each insert requires one update to METERS and one write to METER_READS. The timestamp for each new record is incremented by one hour. One year of data (8760 records per device) is inserted in the ingestion phase and then one month (744 records) of data is aggregated for each device in the extraction test.

The benchmark application was implemented in Java and based on GridDB user feedback. We elected to ingest one year of data while performing monthly billing calculations.

GridDB Schema



SQL Schema



Methodology

Since performance often fluctuates when running applications on cloud services such as Azure, each test was repeated three times.

The first test uses five instances for generating database operations with thirty-two threads per instance. This test should generate a year worth of data. Each thread is designed to insert records into the database as fast as possible, with the aim of inspecting the overall efficiency and performance times of that database for writing and updating records. This test was run three times, with the median of those three taken as the value.

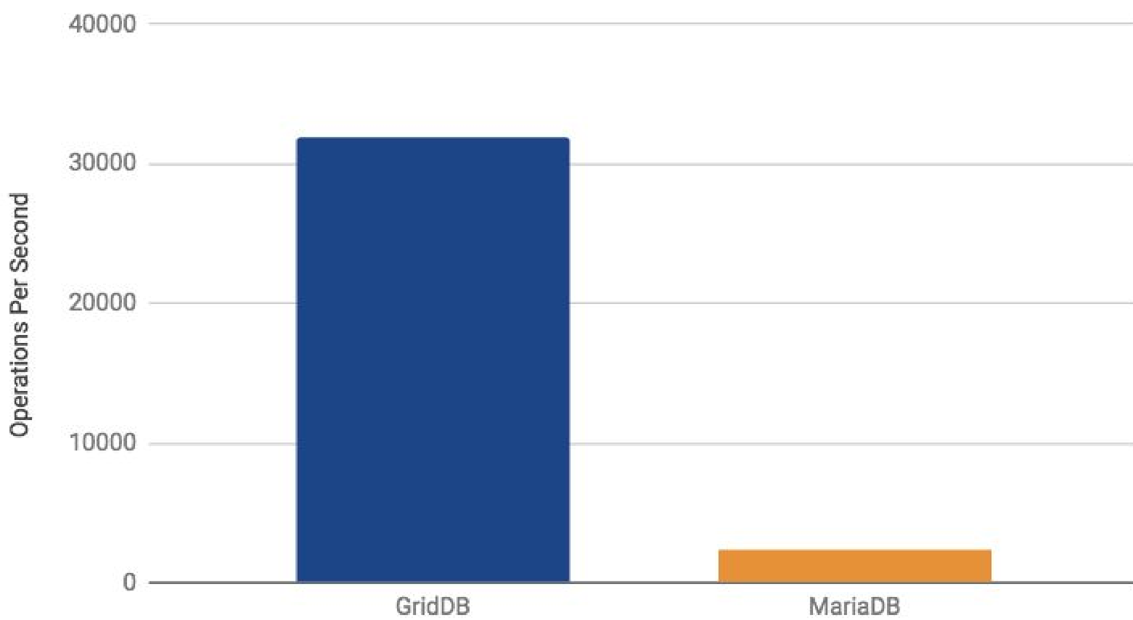
Fixstars attempted to find optimal conditions during initial testing. While GridDB was relatively insensitive to both the number of hosts and number of threads, MariaDB's performance was optimized with 5 client nodes and 32 threads per node. It was also found that individual connections for every thread was faster for both GridDB and MariaDB versus having a shared connection shared between all threads.

For each test, the database server would first have its data deleted and was then restarted; this was done before the workload application was started concurrently on all of the client hosts. For the extractaction test, devices were both loaded and queried in batches of 160 (5 nodes X 32 threads) at a time.

Benchmark Results

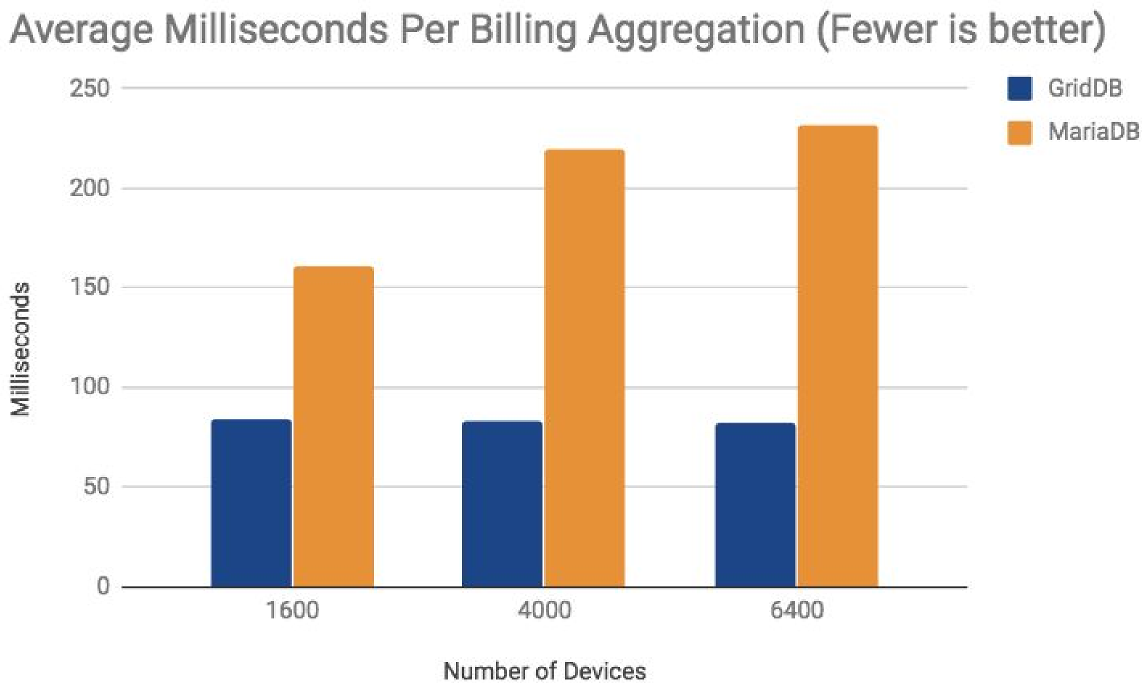
Ingest

The first test measured the number of updates per second that the database can write – where more is better. For each operation, the Meter record is updated (or added if it does not exist) and writes a new Meter_Read record to the database. GridDB was able to process 31,916 operations per second, nearly thirteen times more than MariaDB's 2,423 operations per second.



Extract

The extract test measured the SUM-aggregation of one month's data for each device in the database, with data from both MariaDB and GridDB comparing how long it takes to run the query against one device. With 1600 devices, MariaDB takes roughly twice the time as compared to GridDB to perform the aggregation operation. As the number of devices increases, MariaDB's performance worsens while GridDB's remains consistent.

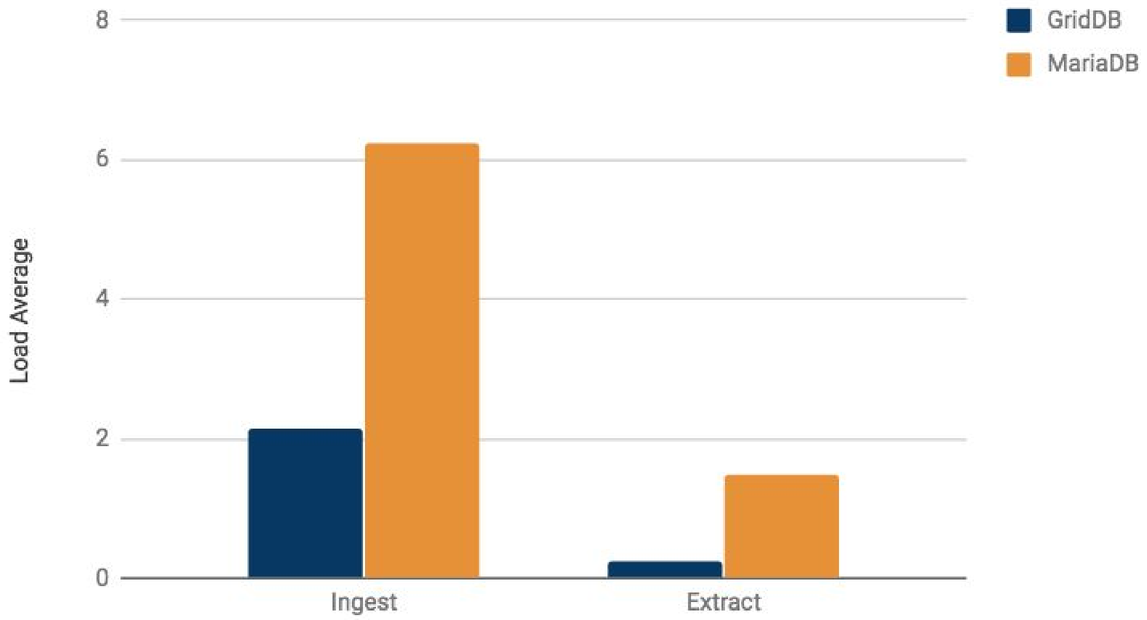


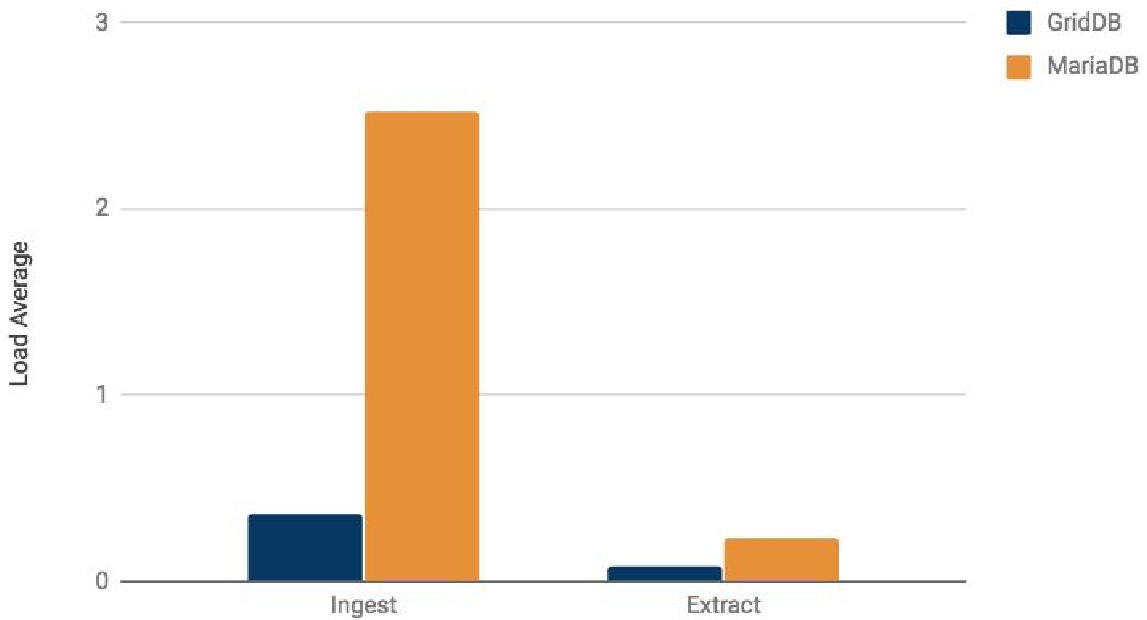
Resource Usage

MariaDB has a higher Load[2] than GridDB while all tests are running and both databases use all available memory on the server while the GridDB client uses significantly less memory.

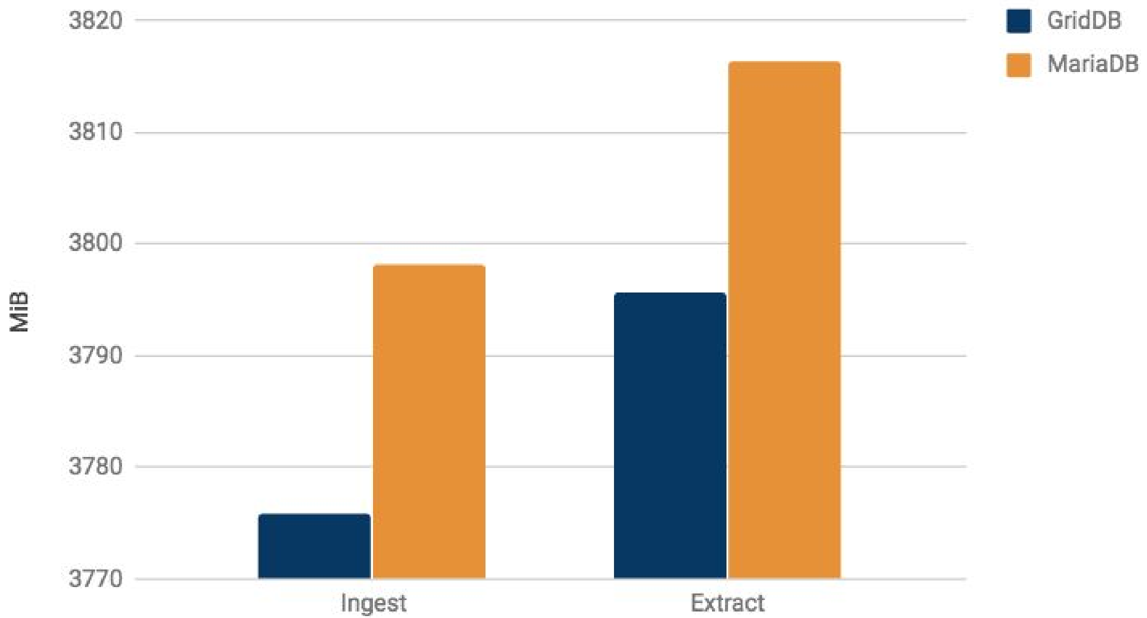
[2] Understand Linux Load Averages
https://www.tecmint.com/understand-linux-load-averages-and-monitor-performance/

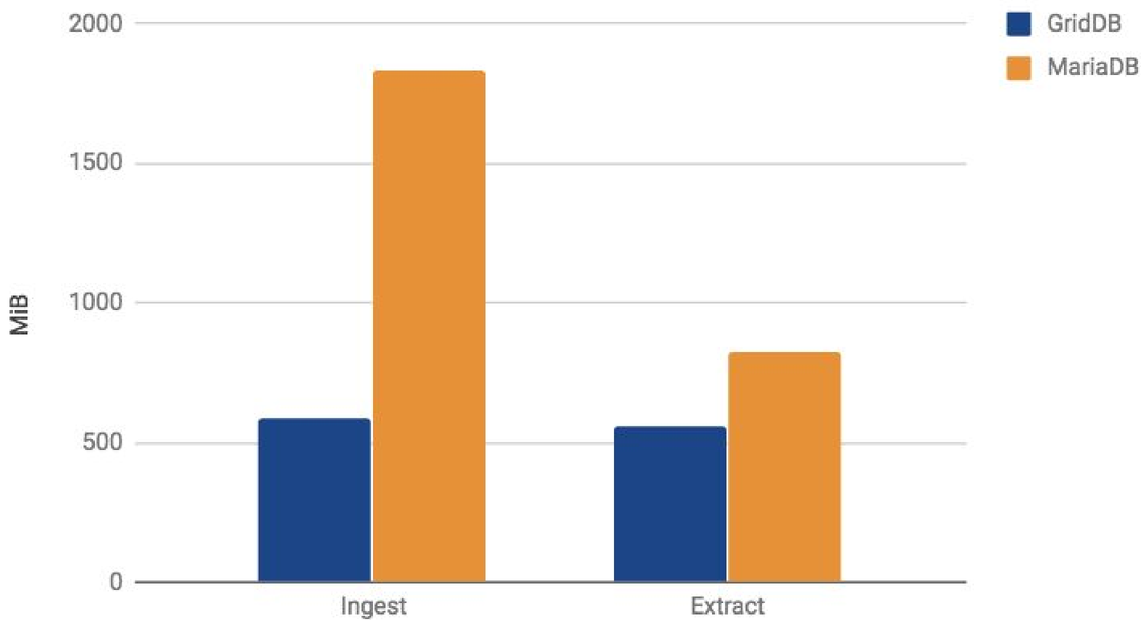
Load Average





Memory Usage





Tabular Results

All tabular results are listed in the order they are shown.

Throughput for Ingest Operations Per Second (More is better)

Database	Operations
MariaDB	2,423.3
GridDB	31,916.3

Milliseconds Per Billing Aggregation (Fewer is better)

Number of Devices	Database	Milliseconds
1600	GridDB	84.3
	MariaDB	160.6
4000	GridDB	82.9
	MariaDB	220.0
6400	GridDB	82.6
	MariaDB	231.4

Server Resource Usage

	Database	Load Average	CPU Usage	Memory Used
Ingest	GridDB	2.16	86.7%	3775 MB
	MariaDB	6.24	74.3%	3798 MB
Extract	GridDB	0.23	6.1%	3795 MB
	MariaDB	1.47	100%	3816 MB

Client Resource Usage

	Database	Load Average	CPU Usage	Memory Used
Ingest	GridDB	0.36	22.3%	584 MB
	MariaDB	2.52	75%	1832 MB
Extract	GridDB	0.08	72%	557 MB
	MariaDB	0.23	13%	825 MB

Conclusion

GridDB outperforms MariaDB for both the ingestion and extraction/aggregation workloads while maintaining a lower load average and memory usage. This means GridDB is well suited for billing applications that are typically based on data collected from a large number of sources.

GridDB's performance can be attributed to a number of factors: GridDB's memory-first architecture that enables it to better manage data that stays in memory reducing the amount of I/O requires while it's Key-Container architecture scales better than typical relational data models.

It should be reiterated that this configuration uses only one server instance, which is the typical configuration for MariaDB. It is, however, not an ideal configuration for GridDB, which would see even greater performance and reliability with multiple server nodes. GridDB's ability to scale-out increases reliability and allows GridDB to grow with your data while relational databases are limited to scaling up and the amount of cores, memory and disk that will fit in one physical system.

Appendices

SQL Schema

```
CREATE TABLE METER_READS (
   meter_id BIGINT NOT NULL,
   timestamp DATETIME,
   usage_since_read DOUBLE,
   usage_this_day DOUBLE,
   usage_this_hour DOUBLE,
   usage_this_week DOUBLE,
   usage_this_month DOUBLE,
   usage_this_year DOUBLE,
   all_time_usage DOUBLE,
   error_code_1 BIGINT,
   error_code_2 BIGINT,
   FOREIGN KEY (meter_id) REFERENCES METERS(id)
);

CREATE TABLE METERS (
   id BIGINT NOT NULL,
   contact_name TEXT,
   email TEXT,
   number TEXT,
   description TEXT,
   address TEXT,
   city TEXT,
   state_province TEXT,
   last_reading DATETIME,
   latitude FLOAT,
   longitude FLOAT,
   PRIMARY KEY (id)
);
```

GridDB Schema

```
// Container Key: METERS
class Meter {
	@RowKey
	public long id;

	public String contact_name;
	public String email;
	public String phone_number;
	public String description;
	public String address;
	public String city;
	public String state_province;
	public Date last_reading;
	public float latitude;
	public float longitude;
}
// Container Key: METER_READS_$id
class MeterRead {
	@RowKey
	public Date timestamp;

	public double usage_since_read;
	public double usage_this_hour;
	public double usage_this_day;
	public double usage_this_week;
	public double usage_this_month;
	public double usage_this_year;
	public double all_time_usage;
	public long error_code_1;
	public long error_code_2;
}
```