```
 ______                __                     ____     ____
/\__  _\              /\ \                   /\  _`\  /\  _`\
\/_/\ \/     __      ___\ \ \___    __  __     ___   ___\ \ \/\ \ \ \L\ \
   \ \ \   /'__`\   /'___\ \  _ `\/\ \/\ \  / __`\ /' _ `\ \ \ \ \ \  _ <'
    \ \ \/\ \L\.\_/\ \__/\ \ \ \ \ \ \_\ \/\ \L\ \/\ \/\ \ \ \_\ \ \ \L\ \
     \ \_\ \__/.\_\ \____\\ \_\ \_\/`____ \ \____/\ \_\ \_\ \____/\ \____/
      \/_/\/__/\/_/\/____/ \/_/\/_/`/___/> \/___/  \/_/\/_/\/___/  \/___/
                                     /\___/
                                     \/__/
```

Final Report

Team 11

Matthew Dunn (m4dunn)

Erik Lungulescu (elungule)

Rajat Patwari (rpatwari)

Rick Liu (r344liu)

Srihari Vishnu (svishnu)

Table of Contents

Glossary

Term	Definition
.ty	The file extension used for TachyonDB data files.
C/C++	Two popular and related statically compiled languages. Supported by TachyonDB's client libraries.
Cache	An in memory data store that is used to speed up reads and writes for frequently accessed data.
CLI	Acronym for "Command Line Interface". This is a text based interface that can be executed from any shell.
Compression	The computer science concept of decreasing the size of data written to the file system.
CSV	Stands for comma separated values. Usually referring to a file with values that are separated by commas.
Data Corruption	Error in computer data that occurs during writing, reading, storage, transmission, or processing.
Database	A software system that allows for writing and querying data.
DB_DIR	The directory containing TachyonDB data.
Edge Device	A computing device that exists on the edge of a network. In general, these devices have limited memory and processing power.
Embedded Database	Database that is integrated into the user application.
Entry	A timestamp-value pair. The value is numeric.
Frame	A slot in the page cache
GUI	Acronym for "Graphical User Interface".
Interpolation	The process of inferring a missing value based on existing data.
IoT	Acronym for "Internet of Things".

Term	Definition
Label	An optional key-value string identifier associated with an entry. Each entry is associated with zero or more labels. E.g. in the stream `temperature{“location”=“America”}`, `“location”=“America”` is a label. Refer to Figure 6 for a detailed diagram.
Library	A collection of precompiled, reusable files, functions, scripts, routines, and other resources that can be referenced for software development.
Low Compute Environment	An environment where only minimal computing power / networks are available.
Metric	A mandatory string identifier associated with an entry. Each entry is associated with a metric. E.g. in the stream `temperature{“location”=“America”}`, `temperature` is the metric. Refer to Figure 6 for a detailed diagram.
Page	A fixed-size portion of a file.
Page Cache	A cache used to speed up data access to pages.
Process	An instance of a running program.
Prometheus	A very popular serverful time series database.
PromQL	A query language for time series databases. Used by Prometheus.
Python	A popular dynamically interpreted programming language. Supported by TachyonDB’s client libraries.
Query	A written command that is used to specify which data to retrieve from a database.

Term	Definition
REPL	REPL is an acronym for Read, Evaluate, Print, and Loop. It is a simple interactive computer program that takes user inputs, executes them, and returns the result to the user.
Rust	A statically compiled memory safe programming language. TachyonDB is written in Rust.
Scalar	A singular numeric value.
Serverless	Does not run as an independent process that must be communicated with over the network.
Shell	Computer program that exposes an operating system's services to a human user or other programs.
Stream	Time series data that belongs to a specific label and set of metrics. E.g. `temperature{“location”=“America”}` is a stream. Refer to Figure 6 for a detailed diagram.
Thread	Sequence of programmed instructions that can be managed independently. A single process may use multiple threads to run multiple sequences of instructions concurrently.
Time Series Data	Chronological data where each entry consists of a timestamp and value pair.
Time Series Database	Database specialized for time series data. Often optimizes for the insertion and querying of time series data at the cost of being more limited than general databases.
Timestamp	Sequence of characters identifying when an event occurred.
TQL	Acronym for “Tachyon Query Language”. The query language used to interact with TachyonDB.
Value	A numeric datum.

Notational Conventions

Calibri is used for normal text.
`Green Roboto Mono` is used for code snippets. `Consolas with syntax highlighting over a dark background` is used for longer code blocks.
`Blue Roboto Mono` is used for TQL queries.
`Red Roboto Mono` is used for commands to be executed by the user. Commands to be executed in a REPL are prefixed with `#`. Commands to be executed in a shell are prefixed with `$`.
`Bolded Consolas over a dark background` is used for program or CLI output.
Any text within <> is meant to be replaced by a specific value. The <> should not be included when the text is replaced.

Background

Time series data is everywhere in the modern world; from measuring electrical activity in the brain to monitoring stock market prices, any observable data over a period of time can be modeled as a “time series”. Typically, this data consists of pairs of (timestamp, number) entries; analysis on the data usually investigates possible trends and patterns of how the value changes over time. For example, two example time series that show temperature over time are shown below in **Figure 1**.

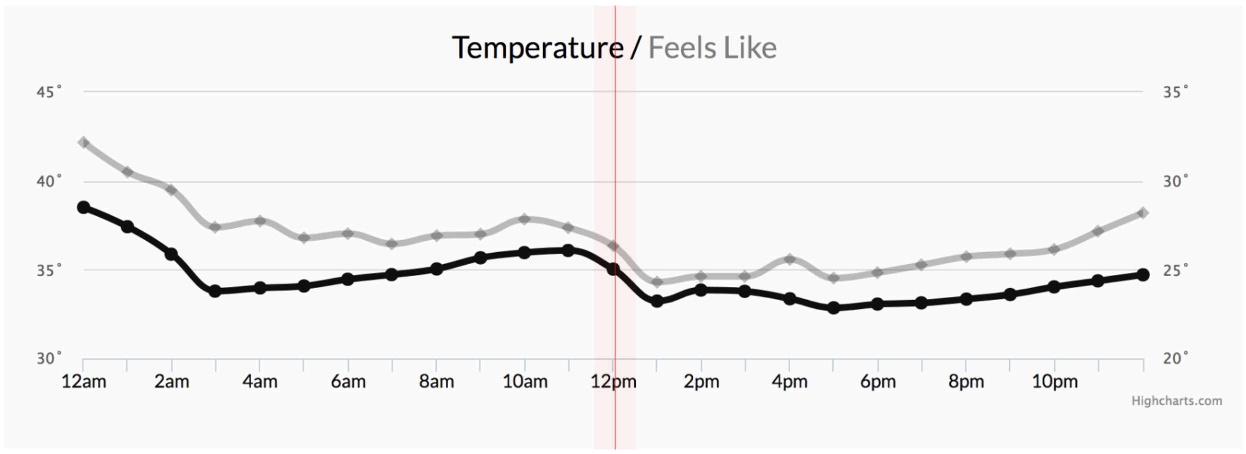


Figure 1: Temperature over Time Time Series [1]

As the world becomes increasingly instrumented, it is becoming more and more common to use IoT devices to collect large amounts of data. These lightweight, often smaller, devices collect information from their environment, and then send them to a system to be aggregated and analyzed. Since there can be many devices that are collecting or sending data, the aggregator system needs to be scalable and efficient to process large amounts of data. The system that is responsible and optimized for this use case is generally called a “time series database”.

Prometheus is one example of a time series database that is often used for monitoring the status of web services. It is a self-contained system that is meant to be run with its own independent process. However, there are many cases where edge devices may not have enough resources to run such a database locally. This is especially the case for devices in locations where network connectivity may be poor and cannot transmit the data immediately. This is where TachyonDB comes in.

Product Overview

TachyonDB is a lightweight, serverless, time series database designed to consume fewer CPU and memory resources than existing alternatives for time series data. This makes TachyonDB ideal for usage on embedded and edge systems with low compute power.

Users interact directly with the database through a CLI and GUI. User applications interact with the database through a client library, which can be dynamically linked or statically compiled. All database operations occur synchronously in the thread that they are executed from – no new processes or threads are created.

Time series data is composed of numeric entries over time, where each entry is a (timestamp, number) pair. The data is stored as basic files on the local file system and potentially in-memory as well for fast access. Users insert and query TachyonDB using the accompanying query language TQL, which is optimized for time series data queries. When data is inserted into TachyonDB, the user associates it with an identifier known as a metric as well as an optional list of key-value identifiers known as labels. The user queries the database by providing a time interval, a metric and a list of labels, and TachyonDB returns a stream composed of the data points that fall inside the time interval and match the metric and labels.

Product Requirements

TachyonDB requires:

- x64 or aarch64 CPU
- 256 MB of memory
- 128 MB of disk space to store the binary + additional space for data

Requirements Specification

Domain Model

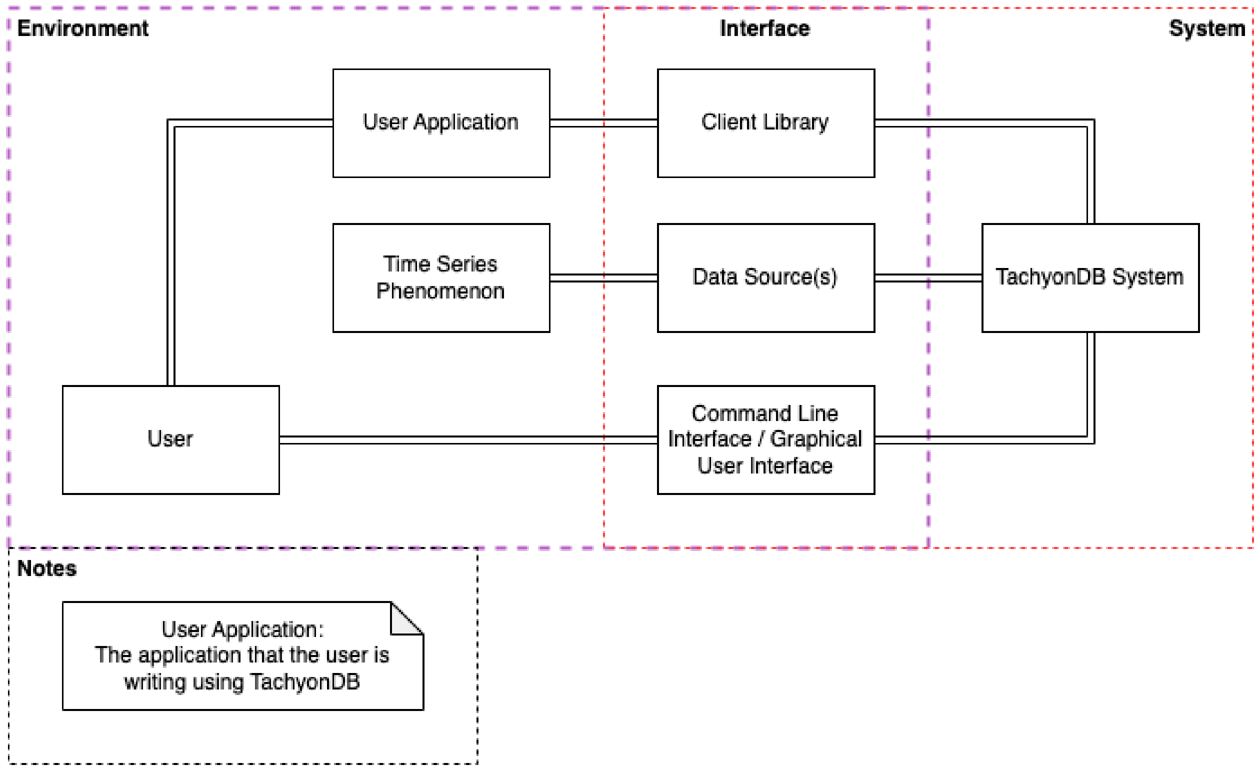

Environment
Interface
System
User Application
Client Library
Time Series Phenomenon
Data Source(s)
TachyonDB System
User
Command Line Interface / Graphical User Interface
Notes
User Application: The application that the user is writing using TachyonDB

Figure 2: Domain Model of TachyonDB

Use Case Model

U = User

1. U: Install the CLI.
2. U: Install the GUI.
3. U: Install the libraries.
4. U: Connect to a database.
5. U: Close connection to a database.
6. U: Create a database.
7. U: Delete a database.
8. U: Create a stream in the database.
9. U: Insert data into a stream.
10. U: Read data from a database.

Assumptions

A1: System is running on the target architecture.
A2: User application is allowed to make memory allocations.
A3: User application has read/write access.
A4: Database directory exists or can be created.
A5: The user does not manually edit database files.
A6: Data is sent in order of increasing timestamps.
A7: Each connection is single-threaded.
A8: At most one process is writing to the database.

Exceptions

E1: User does not connect to a database before writing queries/commands.
E2: User deletes a non-existent database.
E3: User reads from a non-existent stream.
E4: User inserts incorrectly-typed data.
E5: User inserts out-of-order data.
E6: User makes a syntax error in a query.
E7: User makes a syntax error in a CLI command.
E8: System is not running on the target architecture.
E9: User application does not have permission to make memory allocations.
E10: User application does not have read/write access.
E11: Database file is manually edited or deleted without using the interface.
E12: Database file gets corrupted.
E13: Computer crashes during execution.
E14: Computer runs out of memory.
E15: Computer runs out of storage.
E16: Multiple threads in a user application use the same database connection at the same time.
E17: Multiple threads or user applications insert data into a database at the same time.
E18: User runs GUI without pointing it to a valid backend instance.

Variations

V1: Database is used for non-time series data.
V2: Database is used as the primary database for a non-edge device.
V3: A value is used as a label — leading to a separate stream for each inserted value.
V4: Database is used as a key-value store.
V5: User creates a uniquely-named metric for each stream and never uses labels.

V6: User inserts all data into a single metric and differentiates streams purely using labels.
V7: User application uses the CLI instead of the client library.
V8: User queries data and manually performs aggregation operations on the queried data instead of directly using TQL's aggregate functions.
V9: User installs TachyonDB binaries manually.

Design Documentation

Architecture

Figure 3 shows the architecture of TachyonDB, how components interact with each other, and the flow of a typical database query operation. The core components of TachyonDB are highlighted in blue.

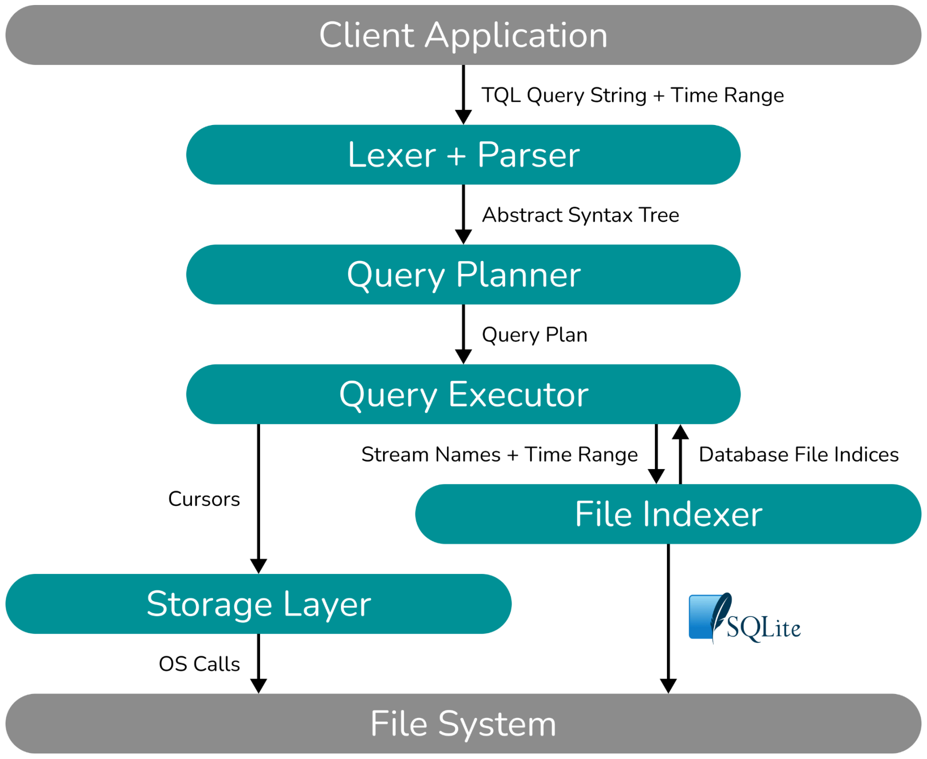


Figure 3: Architecture of TachyonDB and Flow of a Query Operation

Client-Facing Interface

TachyonDB provides users with a CLI, GUI, and library APIs in C/C++, Rust, and Python.

The CLI is intended for use by programmers who wish to manually run commands. Client applications instead integrate TachyonDB into their code by importing the library API and calling the provided API functions. The CLI and library APIs support all user operations (e.g. database and stream creation, data insertion, data querying).

The GUI only supports querying and visualizes query results in real time in a front end web page. It does so via a provided backend server that integrates with Grafana, a popular data visualization and observability platform. A Grafana plugin is provided that queries vector data from the backend server for any query that can be run. The data is then displayed in the Grafana platform and can be used to generate dashboards, alert mechanisms or any other helpful observability tools. The backend server can also be used to get data from the database via regular HTTP requests as well, if desired.

Query Lexer and Parser

Lexing is the process by which a query string is transformed into a list of tokens. Parsing is the process that transforms the tokens into an abstract syntax tree (AST) that encodes the query's structure and meaning.

Tachyon Query Language (TQL) is based on a fork of an open-source lexer and parser for PromQL, the query language used by the Prometheus database. TQL's grammar is thus very similar to PromQL but disallows some features while adding a few new ones that are not valid in Prometheus. Specifically, TQL allows square brackets after aggregation queries to specify an optional sub-period of time to aggregate over:

```
avg(temperature)[10m]
```

This syntax is not allowed in PromQL.

The reason why PromQL is not suitable for TachyonDB as-is is because TachyonDB's data model differs from Prometheus'. Data points in Prometheus are aligned to fixed timestamp intervals whereas TachyonDB data points can have any timestamp. Many query functions also behave differently. For example, aggregation queries in Prometheus aggregate vertically over different streams on the same timestamp, while aggregation queries in TachyonDB aggregate horizontally over time on the same stream. Despite these roadblocks, it was still faster to fork an existing query language as opposed to creating a new query language from scratch.

Query Planner

The Query Planner transforms the AST, which only encodes the structure of the query, into an intermediate tree structure where each node specifies what needs to be executed. This is passed to the query executor.

Query Executor

The Query Executor traverses the Query Planner's execution tree and executes each node. The tree contains what streams and time ranges need to be read; the Query Executor's job is to figure out how to read the actual data and process the various operations that should be applied to it. The Executor uses the File Indexer to find out which physical files need to be read. It then uses the Storage Layer to create cursors to the data for reading or writing.

File Indexer

The File Indexer is responsible for telling the Query Executor which .ty files contain the relevant entries for a given stream and time range. SQLite is used to store some relevant metadata in three separate tables. The first table maps metric and matcher strings to IDs, the second maps IDs to files, and the third maps IDs to a data type.

When a new stream is created, it is split into a metric and matcher strings. Each matcher has a name and a value, which are both strings. The metric is treated like a matcher with the special name "__name". A UUID is generated for the new stream and appended to a list of IDs for the metric and matchers individually in the stream to ID table. This ID is also added to the ID to filename table. When subsequent files are added for the same stream (as more data is written), a new row is inserted into the ID to filename table. Each row in this table keeps track of the start and end time of the file as well. The insertion process is visualized in **Figure 4**. Finally, a row is also added to the ID to data type table which sets the stream type to one of int (0), uint (1), or float (2).

requests{server="US", method="POST"} ⟶ id1
requests{server="EU", method="POST"} ⟶ id2

name	value	ids
"__name"	"requests"	id1, id2
"server"	"US"	id1
"server"	"EU"	id2
"method"	"POST"	id1, id2

id	filename	start	end
id1	"id1/file1.ty"	1	3
id1	"id1/file2.ty"	3	5
id2	"id2/file1.ty"	1	3

id	data_type
id1	2
id1	2
id2	2

Figure 4: Example of TachyonDB's Stream Representation in the File Indexer

To get the files required for a query, the File Indexer fetches the set of IDs for the metric and each matcher in the query, and then finds the intersection. This intersection is the set of stream IDs that are relevant to the query. A query is done on the ID to filename table for each ID in the intersection. Rows with a start and end time outside of the time range specified in the query are discarded. The remaining filenames are returned to be used by the Query Executor.

Writer

The writer is in charge of persisting data the user inserts onto disk. There are two versions of the writer, each optimized for different use cases:

1. **In-memory writer:** The first version buffers user data in memory until it reaches the target file size (62500 values). Once a full file is reached the data is compressed and then persisted to disk. This is a good option for users who want to prioritize speed and are willing to accept the risk of data loss in the event of a crash.
2. **Partially persistent writer:** This writer writes data to disk in chunks while buffering the latest chunk of data in memory. This allows users to read data from a partial file while it is being written. Once a full file size is reached, the file is marked as committed and cannot be updated. This writer strikes a balance between performance and minimizing data loss in the event of a crash. To facilitate this, a file can be in one of two states:
    a. Committed: no changes can be made to the file.
    b. Uncommitted: data can still be appended to it by the writer.

Both writers allow the user to flush data onto disk. This is the only way to guarantee that data is persisted on disk. Both writers have a risk of losing buffered data in the event of a crash. Once a flush occurs the file created by the flush cannot be modified by future inserts. For the partially persistent writer, a flush marks all files it affects as committed. Both writers are in charge of updating the File Indexer's list of stream-to-file name mappings. Lastly, concurrent writes are not supported. Only one process can be writing to a given stream at a time.

Compression Details

A core feature of TachyonDB is a timestamp and value compression algorithm implemented within the codebase. This compression method is uniquely optimized for noisy, sequential data, which is found in traditional time series datasets. Broadly, there are 4 steps in the integer compression algorithm:

1. **Delta** – the difference of consecutive values in the integer stream is taken. In time series data, since values are sequential (increasing/decreasing/roughly constant), by taking

deltas, the magnitude of the integers is significantly reduced. For example, if the values of the stream are 1744813335, 1744813337, 1744813340, the deltas are 2,3. The initial value is the only full value that needs to be stored.

2. **Double-Delta** – we can take this one step further and compute the “delta of the deltas”. This operation will reduce the magnitude of the deltas, and can be efficiently stored.

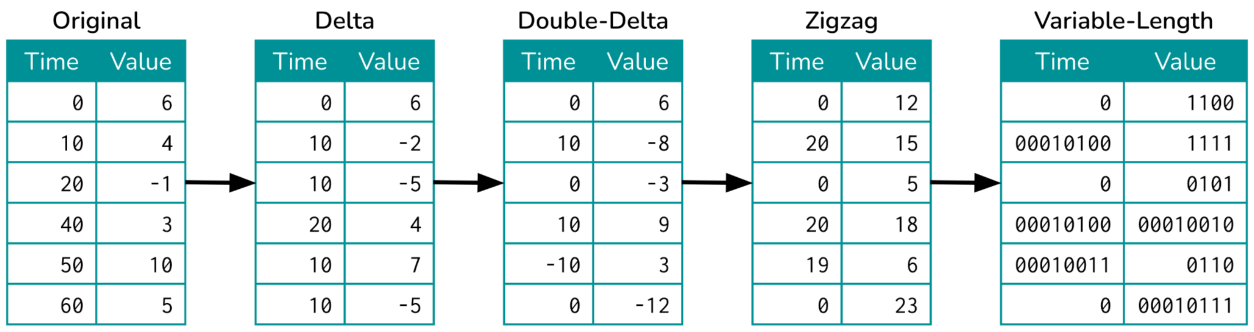
3. **Zig-Zag** – double deltas will produce negative integers which are usually represented in two's-complement form (the most significant bit is set). Since variable-length encoding requires storing some suffix of the bits of each integer, if we stored negative integers as-is, it would cause each value to require 4 bytes. Instead, we map the signed integers into the unsigned integer space, such that the resulting stream has no negative values, and the encoded value is minimized.

   This is done using the following formula (assume we are encoding n):
   - If n is non-negative, the encoded result is $2 \times n$
   - If n is negative, the encoded result is $2 \times |n| - 1$

   For example, 0 encodes to 0, -1 encodes to 1, 1 encodes to 2 etc. in a zig-zag-like fashion.

4. **Variable-Length Encoding** – all bits from the leftmost set bit to the least significant bit are required to represent the value. These bits are then prefixed with zeros to pad it up to the nearest byte – this enables fast decompression since values are byte aligned. Also, instead of storing a length for each value, a single length header is used for 16 consecutive values since generally double deltas do not vary much in a small interval. This value provides a tradeoff between decompression performance and compression size.

A visual representation of the compression pipeline can be found in **Figure 5**. Benchmark results for compressing and decompressing data can be found in the Benchmarks section.



Original

Time	Value
0	6
10	4
20	-1
40	3
50	10
60	5

Delta

Time	Value
0	6
10	-2
10	-5
20	4
10	7
10	-5

Double-Delta

Time	Value
0	6
10	-8
0	-3
10	9
-10	3
0	-12

Zigzag

Time	Value
0	12
20	15
0	5
20	18
19	6
0	23

Variable-Length

Time	Value
0	1100
00010100	1111
0	0101
00010100	00010010
00010011	0110
0	00010111

Figure 5: Example of TachyonDB's Multi-Step Compression Scheme

User Manual: Tachyon Query Language (TQL)

TQL is the official query language of TachyonDB and is a variation of the highly popular query language PromQL. It's a functional query language that lets users select and aggregate time series data. These queries can be run via the TachyonDB CLI, the TachyonDB GUI, or the TachyonDB library API.

Data Types

Range Vector

A list of lists of entries containing data over a range of time for a singular stream. Each entry consists of a timestamp value pair.

E.g. CPU Temperature:

```
[(5, 48), (9, 51), (10, 50), (18, 46)]
```

Scalar

A numerical value.

E.g. Average CPU Temperature:

```
48.75
```

Time Durations

Time durations in queries are specified as a number followed by any of the following allowed units:

- `ms`: milliseconds
- `s`: seconds
- `m`: minutes
- `h`: hours
- `d`: days
- `y`: years

Time Series Selectors

Range Vector Selectors

Range vector selectors allow the selection of a set of time series where each time series is a list of entries. Simply put, range vector selectors return a variable number of range vectors for every single stream that the query matches. In the simplest query, a metric name needs to be provided.

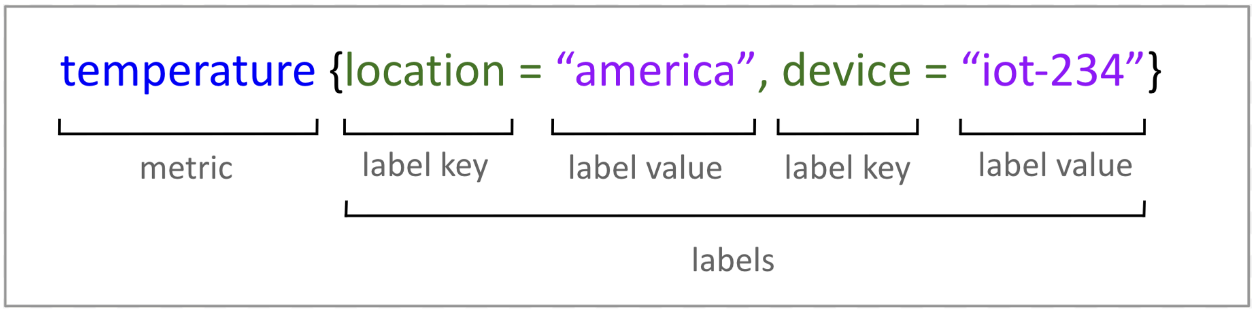


Figure 6: Components of a Stream: Distinction Between Metric and Labels

Example Range Vector Queries

The following examples explore querying CPU temperature data for two different IoT devices named “`iot-a-234`” and “`iot-b-567`”.

Example Query 1

```
temperature
```

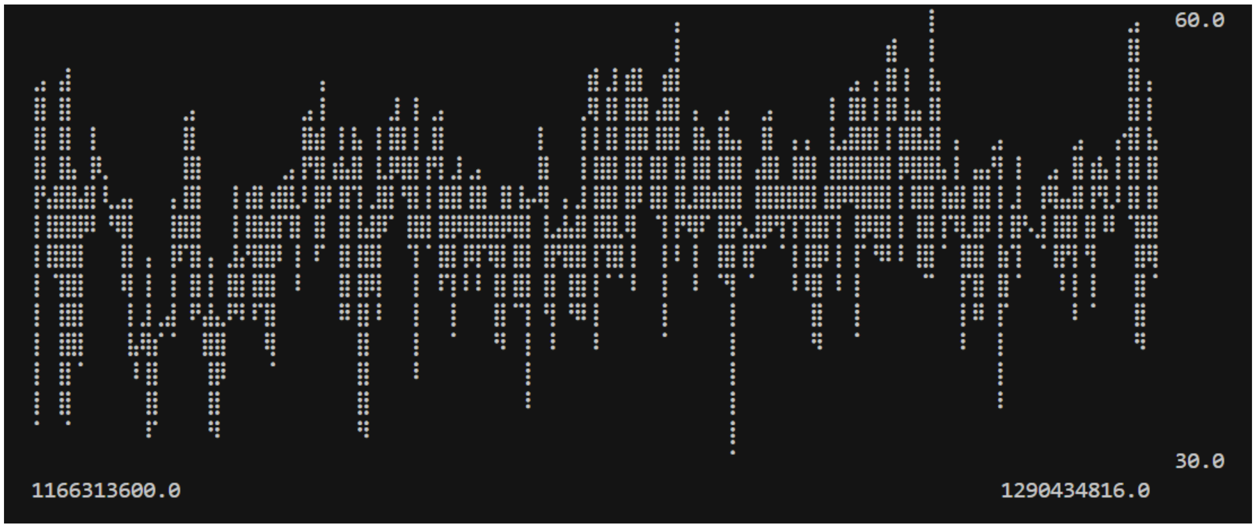
Query Description

The first example queries only the metric name. This returns back the associated time series data for each stream that matches the metric.

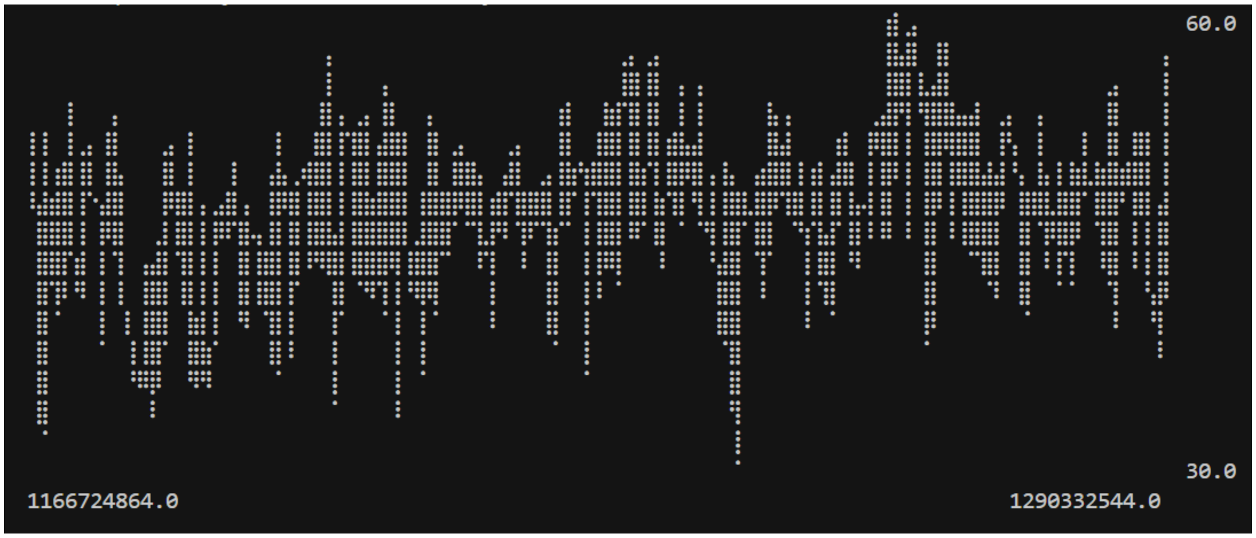
CLI Output

The output consists of all matched streams, which in this example corresponds to both of the IoT devices. The CLI output displays the resulting data in a graphical form for all returned streams.

Stream: `temperature{device="iot-a-234"}`



Stream: `temperature{device="iot-b-567"}`



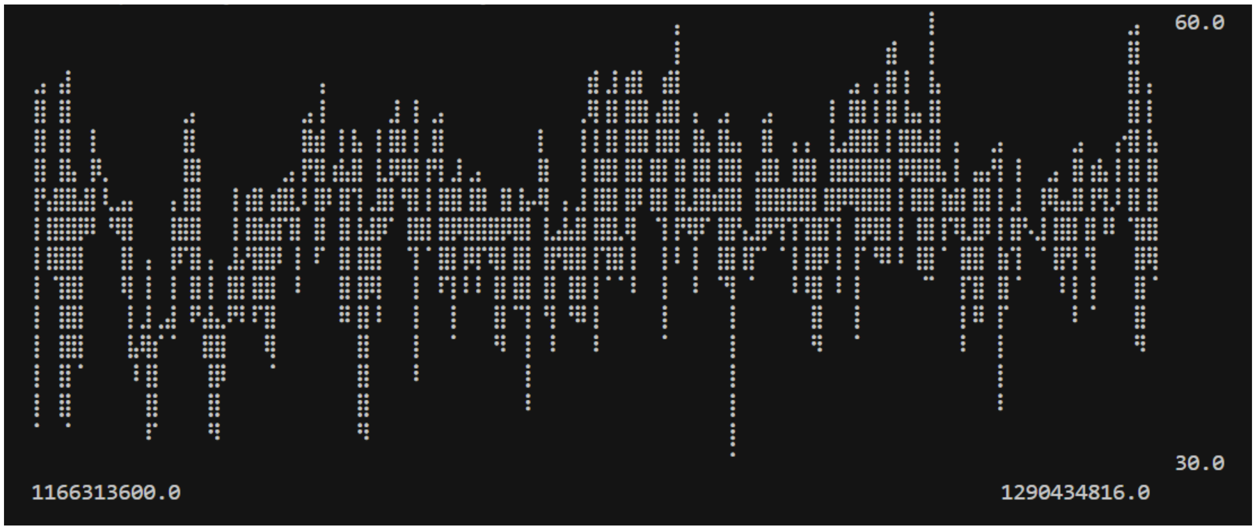
Example Query 2

`temperature{device=”iot-a-234”}`

Query Description

To narrow down the scope of the query to a subset of time series, labels can be provided. This example specifies the device name via the label `”device”`. This results in a single range vector being returned for a specific IoT device.

CLI Output



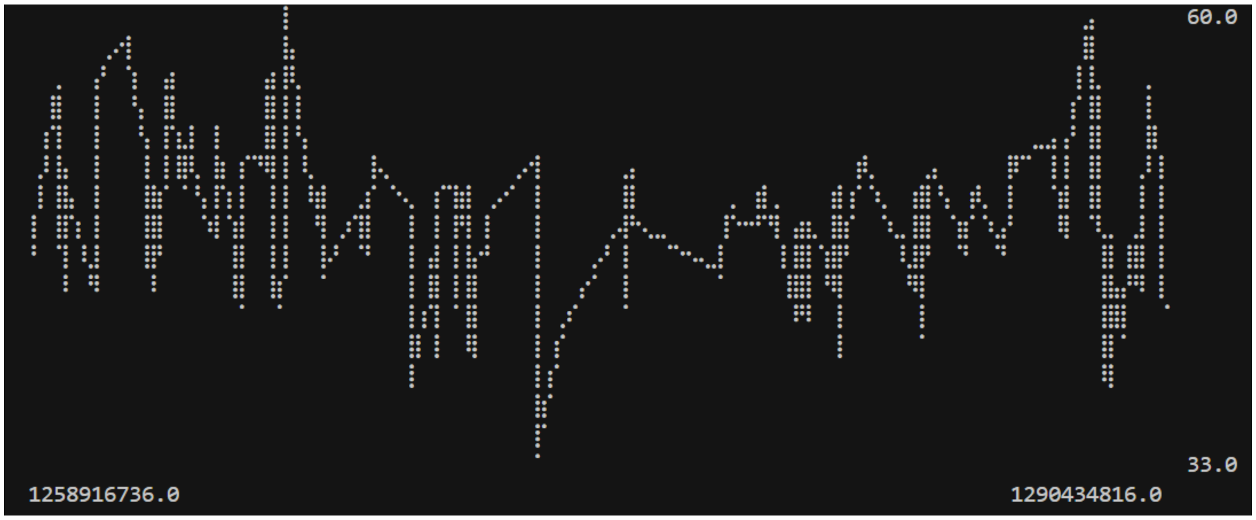
Example Query 3

`temperature{device=”iot-a-234”}[1y]`

Query Description

To modify the time range at which data is returned, different time ranges can be appended to the query. The example query returns back the CPU temperature data belonging to IoT device `“iot-a-234”` for the last year.

CLI Output



Aggregation Queries

Aggregation queries take in a single range vector and return back a scalar. They take the following form:

```
<aggregation-function>(range vector)
```

- `avg`: Returns the average value in the range vector.
- `topk`: Returns the largest k entries in the range vector.
- `bottomk`: Returns the smallest k entries in the range vector.
- `count`: Returns the number of entries in the range vector.
- `max`: Returns the maximum value in the range vector.
- `min`: Returns the minimum value in the range vector.
- `sum`: Returns the sum of the values in the range vector.

Example Aggregation Query

```
avg(temperature{device=”iot-a-234”})
```

Query Description

To calculate the average temperature of the IoT device `“iot-a-234”` for all time, use the `avg` aggregation query.

Aggregation by Subperiod

Some aggregation queries optionally take a time duration, in which case the aggregation function is applied over time periods of the specified length. A list of vectors is returned, where the timestamp is the end timestamp of the period and the value is the aggregated value over the period.

```
<aggregation-function>(range vector)[time duration]
```

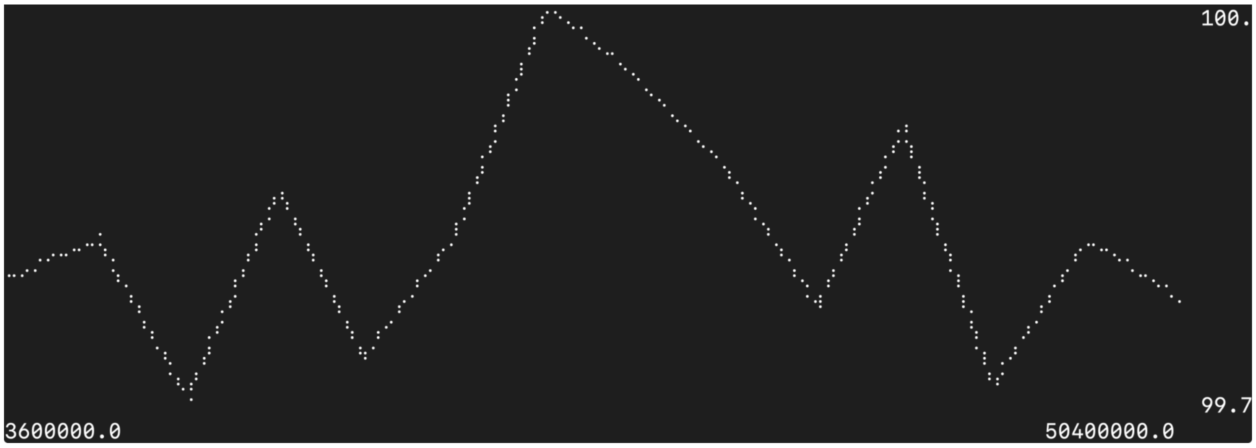
Example Aggregation Query

```
avg(temperature{device=”iot-a-234”})[10m]
```

Query Description

Calculates the average temperature of the IoT device `“iot-a-234”` over every 10 minute interval.

CLI Output



Binary Operations

Binary operations are split into three groups of operators based on the data types of the two operands.

Scalar-to-Scalar Operators

Scalar-to-scalar operators take a scalar and another scalar and return a scalar. They take the following form:

```
scalar1 <op> scalar2
```

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `%`: Modulo
- `^`: Exponentiation
- `==`: Equals
- `!=`: Not equals
- `>`: Greater than
- `<`: Less than
- `>=`: Greater than or equal
- `<=`: Less than or equal

Example Scalar-to-Scalar Query
```
(avg(temperature{device=”iot-a-234”})+avg(temperature{device=”iot-b-56
7”})) / 2
```

Query Description
This example query finds the average temperature between both IoT devices. It does so by summing the averages of each device individually and then dividing the total sum by two.

CLI Output
```
46
```

Vector-to-Scalar Operators

Vector-to-scalar operators take a vector and a scalar and return a vector. They take the following form:

```
vector <op> scalar OR scalar <op> vector
```

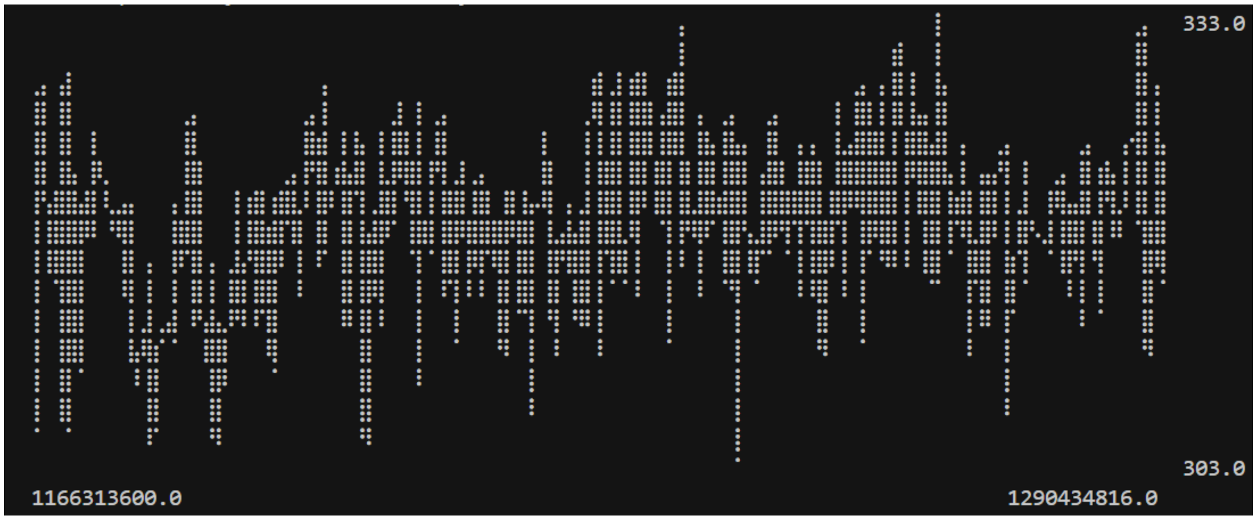
- `+`: Each value in the `vector` adds the `scalar`.
- `-`: Same as above except values are subtracted by the `scalar`.
- `*`: Same as above except values are multiplied by the `scalar`.
- `/`: Same as above except values are divided by the `scalar`.
- `%`: Same as above except values are mod `scalar`.
- `^`: Same as above except values are raised to the power of the `scalar`.

Scalar-to-scalar operators are **commutative**, so the order of the `vector` and the `scalar` does not matter.

Example Vector-to-Scalar Query
```
temperature{device=”iot-a-234”} + 273
```

Query Description
This example query converts the temperature of the IoT device `“iot-a-234”` from celsius to kelvin.

CLI Output



Vector-to-Vector Operators

Vector-to-vector operators take two vectors and return a vector. They take the following form:

```
vector1 <op> vector2
```

- `+`: Each value in `vector1` and `vector2` is added.
- `-`: Same as above except values are subtracted.
- `*`: Same as above except values are multiplied.
- `/`: Same as above except values are divided.
- `%`: Same as above except each value is the remainder after division.
- `==`: Equals
- `!=`: Not equals
- `>`: Greater than
- `<`: Less than
- `>=`: Greater than or equal
- `<=`: Less than or equal

If both vectors have a value at the same timestamp, the operator is applied to the two values. However, if only one of the streams has a value at a timestamp, the other stream **interpolates** a value at that timestamp using values from nearby timestamps. This is shown in **Figure 7**.

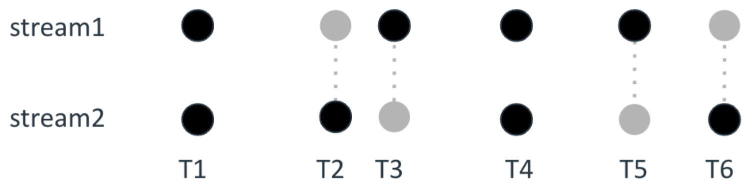


Figure 7: Interpolation Between Streams with Misaligned Timestamps

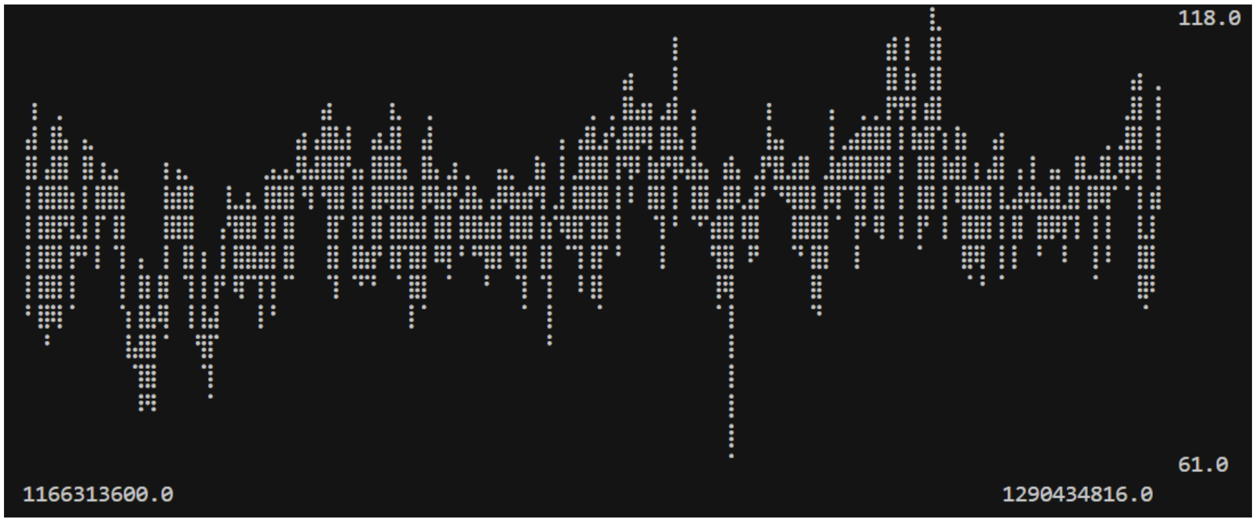
Example Vector-to-Vector Query

`temperature{device="iot-a-234"} + temperature{device="iot-b-567"}`

Query Description

This example query sums the temperature of the IoT devices `"iot-a-234"` and `"iot-b-567"` at every timestamp in the two streams.

CLI Output



User Manual: Installation

The TachyonDB project provides prebuilt binaries for most common Unix-based operating systems. TachyonDB does not currently support the Windows operating system.

Installing the TachyonDB CLI

1. Ubuntu (x64, aarch64)

- To install the TachyonDB CLI, first update the package list:

```
$ sudo apt update
```

- Now install the CLI:

```
$ sudo apt install tachyondb-cli
```

2. MacOS (x64, aarch64)

- To install the TachyonDB CLI on MacOS it is assumed Homebrew is already set up on the user's machine. Then install the CLI with Homebrew:

```
$ brew install tachyondb-cli
```

Installing the TachyonDB GUI

1. Ubuntu (x64, aarch64)

- To install the TachyonDB GUI, first run a Grafana instance (with Docker for example):

```
$ docker run grafana
```

- Now install the TachyonDB plugin from the Grafana interface

Installing the TachyonDB Libraries

1. Ubuntu (x64, aarch64)

- To install the TachyonDB libraries to integrate with user applications, first update the package list:

```
$ sudo apt update
```

- Now install the libraries:

```
$ sudo apt install tachyondb-lib
```

2. MacOS (x64, aarch64)

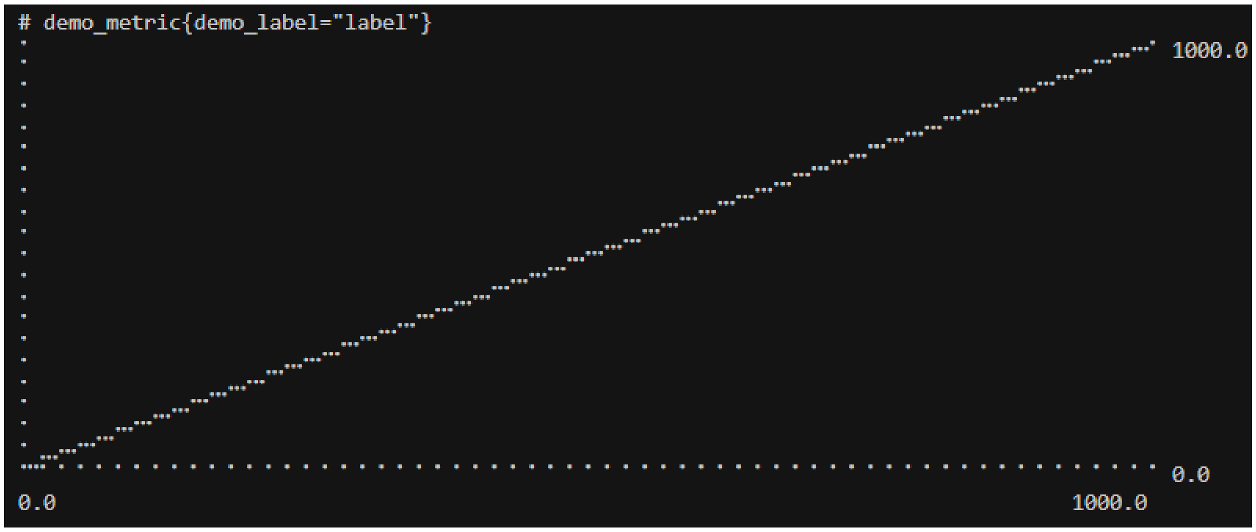
- To install the TachyonDB libraries on MacOS it is assumed Homebrew is already set up on the user's machine. Then install the libraries with Homebrew:

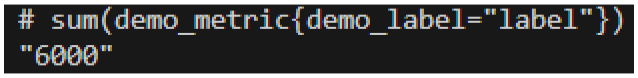
  ```
  $ brew install tachyondb-lib
  ```

User Manual: CLI

Quickstart

The user should follow these steps to get started with TachyonDB through the CLI:

1. Create a database and start the interactive shell. The database that will be created at `./demo_db`:
   ```
   $ tachyon demo_db
   ```
2. Create a stream (or use `# .write -c` in step 3 below)
   ```
   # .create demo_metric{demo_label=”label”}
   ```
3. Insert some data into a stream with metric `demo_metric` and a label `demo_label=”label”`:
   ```
   # .write dataset.csv demo_metric{demo_label=”label”}
   ```
   The data from the csv file will be parsed into an array of timestamp value pairs. This data will be inserted into the database.
4. Execute a query to retrieve all of the data:
   ```
   # demo_metric{demo_label=”label”}
   ```
   
   ```
   # demo_metric{demo_label="label"}
   ```

5. Execute a query to get the sum of the data:
   ```
   # sum(demo_metric{demo_label=”label”})
   ```
   
   ```
   # sum(demo_metric{demo_label="label"})
   "6000"
   ```
6. Exit the REPL:
   ```
   # .exit
   ```

Typical Scenarios

Creating and Connecting to a Database

When the user creates a REPL, a connection to the database is opened. Further operation on the database can be done from within the REPL. Create a REPL as follows:

```
$ tachyon <db_dir>
```

If no database exists at `<db_dir>`, an empty database will automatically be created at `<db_dir>`.

Closing Connection to a Database

From within the REPL,

```
# .exit
```

Deleting a Database

To delete a database, manually delete the DB_DIR directory and all of its contents.

Creating a Stream and Inserting Data

From inside the REPL, insert data manually into the database with a given stream and data type:

```
# .write [options] <path> <stream>
```

In both cases, the stream `<stream>` needs to exist. To automatically create it add the optional flag: `-c, −create` if it doesn't already exist.

Reading Data from a Database

From inside the REPL, execute a query (see the TQL section for more details). The result of a range vector selector query is returned in a graphical format in the REPL. The result of an aggregation query is a single scalar printed in the REPL.

For example, write a vector selector query like:

```
# my_metric{my_label=”dummy”}
```

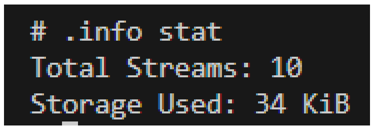
or an aggregate query like:

```
# sum(my_metric{my_label=”dummy”})
```

Listing Stats

From inside the REPL, to list out information about the database use:

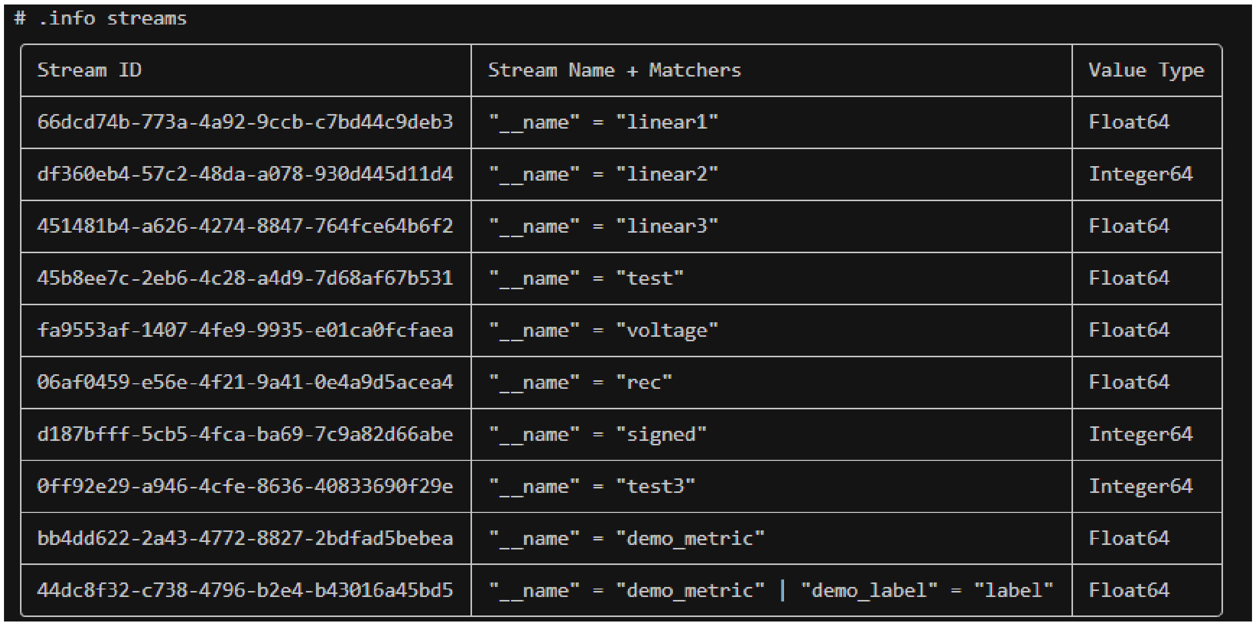
```
# .info stat
```



```
# .info stat
Total Streams: 10
Storage Used: 34 KiB
```

To list out all of the streams:

```
# .info streams
```



```
# .info streams
| Stream ID                            | Stream Name + Matchers                            | Value Type |
| 66dcd74b-773a-4a92-9ccb-c7bd44c9deb3 | "__name" = "linear1"                              | Float64    |
| df360eb4-57c2-48da-a078-930d445d11d4 | "__name" = "linear2"                              | Integer64  |
| 451481b4-a626-4274-8847-764fce64b6f2 | "__name" = "linear3"                              | Float64    |
| 45b8ee7c-2eb6-4c28-a4d9-7d68af67b531 | "__name" = "test"                                 | Float64    |
| fa9553af-1407-4fe9-9935-e01ca0fcfaea | "__name" = "voltage"                              | Float64    |
| 06af0459-e56e-4f21-9a41-0e4a9d5acea4 | "__name" = "rec"                                  | Float64    |
| d187bfff-5cb5-4fca-ba69-7c9a82d66abe | "__name" = "signed"                               | Integer64  |
| 0ff92e29-a946-4cfe-8636-40833690f29e | "__name" = "test3"                                | Integer64  |
| bb4dd622-2a43-4772-8827-2bdfad5bebea | "__name" = "demo_metric"                          | Float64    |
| 44dc8f32-c738-4796-b2e4-b43016a45bd5 | "__name" = "demo_metric" | "demo_label" = "label" | Float64    |
```

Changing Configuration Options

From inside the REPL, to change a CLI configuration option:

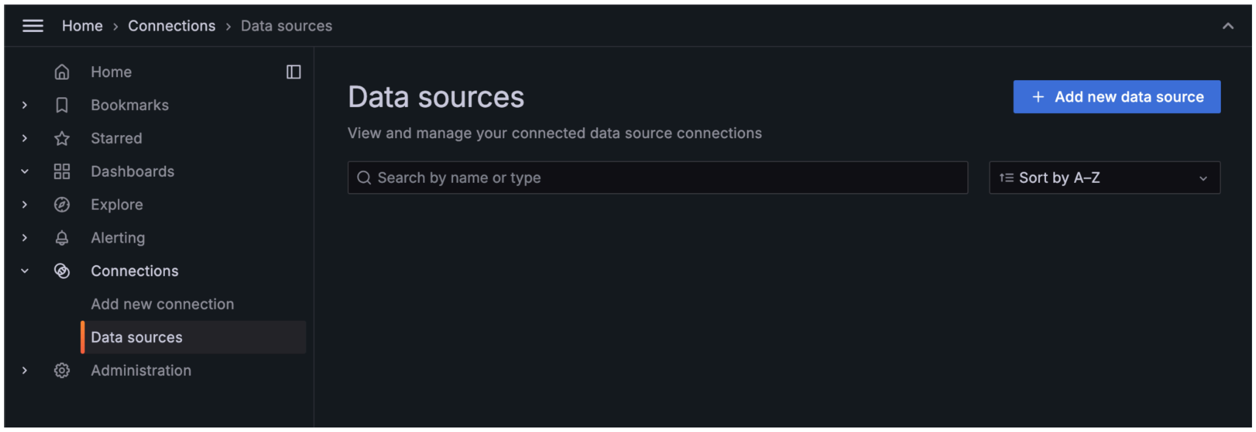
```
# .mode
Options:
  -o, --output-mode <OUTPUT_MODE>  [possible values: graphical, tabular, file]
  -p, --path <PATH>
  -v, --value-type <VALUE_TYPE>    [possible values: i64, u64, f64]
  -h, --help
```

User Manual: GUI

Quickstart

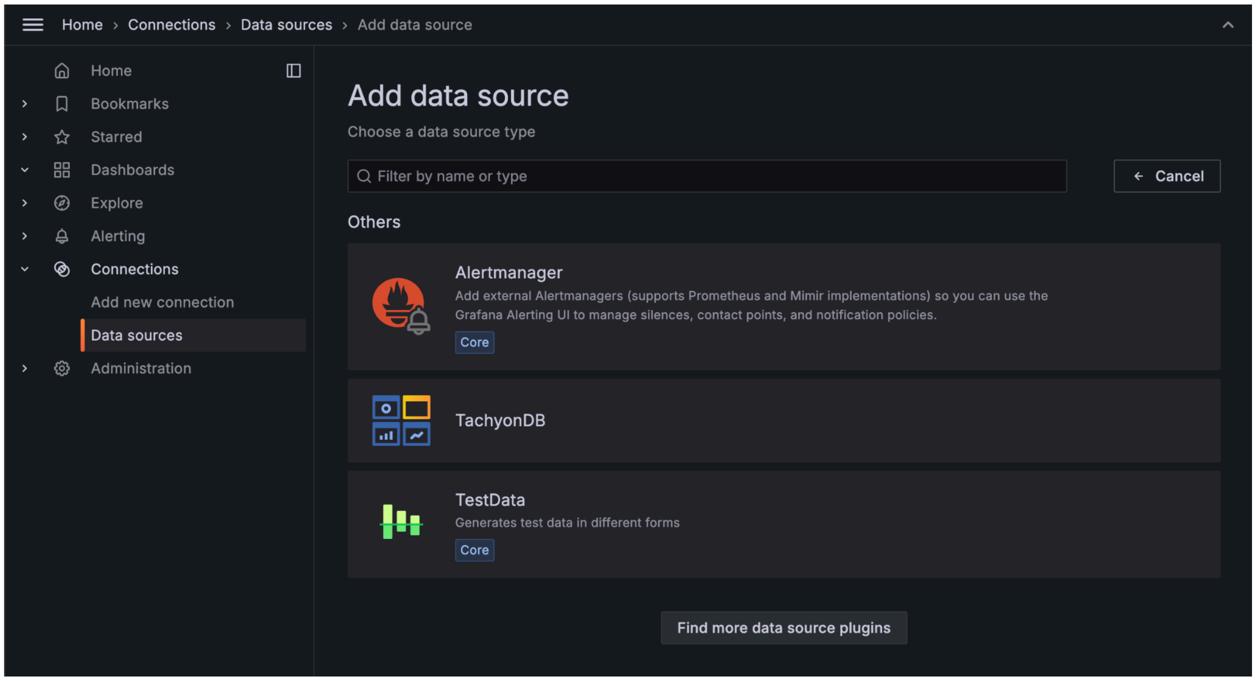
The user should follow these steps to get started with connecting a TachyonDB database to the GUI using a Grafana plugin. This guide assumes the user has a basic knowledge of Grafana and has a Grafana server set up:

1. Start the TachyonDB web backend server and make note of the URL the backend is running on:
   ```
   $ tachyon_web_backend
   ```
2. Suppose a Grafana server is set up and running. Go to the dashboard and install the TachyonDB plugin.
3. On the left panel, go to `Connections > Data sources`. Press `Add new data source`, then look for the TachyonDB data source type:



Data sources

View and manage your connected data source connections


Home › Connections › Data sources › Add data source

Add data source

Choose a data source type

Filter by name or type

Others

Alertmanager

Add external Alertmanagers (supports Prometheus and Mimir implementations) so you can use the Grafana Alerting UI to manage silences, contact points, and notification policies.

Core

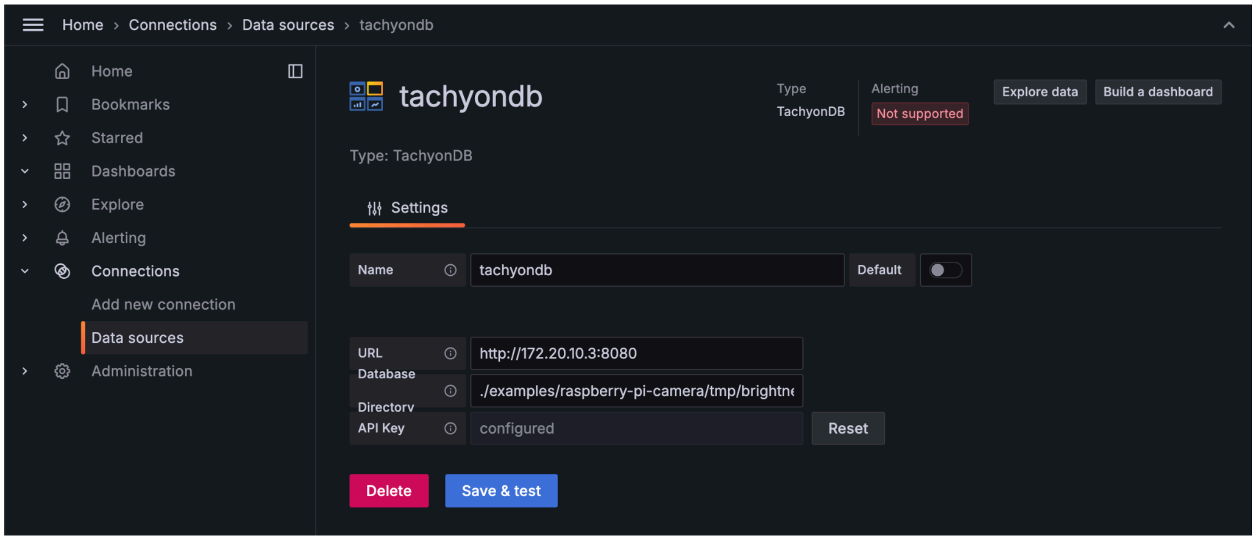
TachyonDB

TestData

Generates test data in different forms

Core

4. Set `URL Database` to the URL of the TachyonDB backend server from Step 1. Set `Database Directory` to the director of the database on the database device:


Home › Connections › Data sources › tachyondb

tachyondb

Type: TachyonDB

Alerting: Not supported

Type: TachyonDB

Settings

Name: tachyondb

Default

URL Database: http://172.20.10.3:8080

Directory: ./examples/raspberry-pi-camera/tmp/brightne

API Key: configured

5. Press `Save & test` to check the connection. Grafana dashboards can now be created to run queries.

User Manual: Rust

Quickstart

To install the TachyonDB library crate run:

```
$ cargo add tachyondb-lib
$ cargo init
```

The following code inserts the integers from 0-99 into a stream called latency{service= “web”} and queries the sum of the inserted data.

```rust
use tachyon_core::*;

let root_dir = "./tmp/db";
// Creates a new connection to a database at "./tmp/db"
let mut connection = Connection::new(root_dir).unwrap();
connection.create_stream(r#"latency{service = "web"}"#).unwrap();

let mut inserter = connection.prepare_insert(r#"latency{service = 
"web"}"#).unwrap();

for i in 0..100 {
    // Inserts data
    inserter.insert_uinteger64(i, i.into()).unwrap();
}
// the stream latency{service = "web"} now contains
// timestamps 0 to 99 with their corresponding values

// Flushes the writer and persists data to disk
inserter.flush().unwrap();
drop(inserter);

// Prepares a new vector query
let mut vector_statement = connection.prepare_query(
    r#"latency{service = "web"}"#,
    Some(0), // start timestamp
    None // end timestamp (None means query until the last timestamp)
).unwrap();
```

```
while let Some(vector) = vector_statement.next_vector() {
    println!("Vector: {}", vector);
}

drop(vector_statement);

// Prepares a new aggregate query
let mut sum_statement = connection.prepare_query(
    r#"sum(latency{service = "web"})"#,
    Some(0), // start timestamp
    None // end timestamp (None means query until the last timestamp)
).unwrap();
// Gets the result as a scalar and outputs it
let sum_result = sum_statement.next_scalar().unwrap();
// Outputs "Result: Value { uinteger64: 4950 }"
println!("Result: {}", sum_result);

drop(sum_result);
```

Typical Scenarios

To install the TachyonDB library crate run:

```
$ cargo add tachyondb-lib
```

Make sure to import the TachyonDB API:

```
use tachyon_core::*;
```

Creating and Connecting to a Database

```
fn Connection::new(db_dir: impl AsRef<str>) -> Result<Connection,
TachyonErr>;
```

If no database exists at `db_dir`, an empty database will automatically be created at `db_dir`.

Closing Connection to a Database

The connection closes automatically when the `Connection` object goes out of scope. To immediately close a connection, use Rust's builtin `drop()` function.

Deleting a Database

To delete a database, manually delete the DB_DIR.

Creating a Stream

```
fn Connection::create_stream(
    &mut self,
    stream: impl AsRef<str>,
    value_type: ValueType
) -> Result<Inserter, TachyonErr>;
```

Inserting Data

1. Prepare an inserter to insert data into an open connection with:

```
fn Connection::prepare_insert(
    &mut self,
    stream: impl AsRef<str>,
) -> Result<Inserter, TachyonErr>;

fn Inserter::insert(timestamp: Timestamp, value: Value) -> Result<(),
TachyonErr>;
```

2. The `Value` type is defined as:

```
#[derive(Clone, Copy)]
pub union Value {
    integer64: u64,
    uinteger64: i64,
    float64: f64,
}
```

3. When done inserting, make sure to call:

```
fn Inserter::flush() -> Result<(), TachyonErr>;
```

Reading Data from a Database

1. Prepare a statement using:

```
fn Connection::prepare_query(
    &mut self,
    query: impl AsRef<str>,
    start: Option<Timestamp>,
    end: Option<Timestamp>
) -> Result<Query, TachyonErr>;
```

   Note that `start` and `end` can be `None`. This can be thought of as setting `start` to zero and `end` to infinity. If both are `None`, this means all entries are included in the query.

2. If the result of the query is a scalar, use:

```
fn Query::next_scalar(&mut self) -> Option<Value>;
```

3. If the result of the query is a vector, use:

```
fn Query::next_vector(&mut self) -> Option<Vector>;
```

4. To get all results from a query, continue using these methods until the optional return is `None`.

User Manual: C/C++

Quickstart

The following code inserts the integers from 0-99 into a stream called latency{service= “web”} and queries the sum of the inserted data. This code shows how to deal with error handling functionality for the first call into the library then assumes that all subsequent calls succeed without errors.

```c
#include <TachyonDB.h>

// Creates a new connection to a database at "./tmp/db"
void *connection_out = NULL;
uint8_t result = tachyon_new("./tmp/db", &connection_out);
if (result != 0) {
    // Error handling when result is not a success
    tachyon_error_print(result, connection_out);
    tachyon_error_free(result, connection_out);
    return 1;
}
struct TachyonConnection *connection = (struct TachyonConnection *)
connection_out;

void *stream_create_out = NULL;
tachyon_stream_create(connection, "latency{service = \"web\"}",
TachyonValueType_UInteger64, &stream_create_out);

struct TachyonInserter *inserter_out = NULL;
tachyon_insert_prepare(connection, "latency{service = \"web\"}",
&inserter_out);
struct TachyonInserter *inserter = (struct TachyonInserter *)
inserter_out;

for (int i = 0; i < 100; ++i) {
    void *insert_out = NULL;
    // Inserts data
    tachyon_insert_uinteger64(inserter, i, value, &insert_out);
}
```

```
// Flushes the writer and persists data to disk
void *flush_out = NULL;
tachyon_insert_flush(inserter, &flush_out);
tachyon_insert_free(inserter);

const TachyonTimestamp start = 0;

// Prepares a new vector query
void *vector_statement_out = NULL;
tachyon_query_prepare(
    connection,
    "latency{service = \"web\"}",
    &start,
    NULL, // end timestamp NULL means query until the last timestamp
    &vector_statement_out
);
struct TachyonQuery *vector_statement = (struct TachyonQuery *)
vector_statement_out;

struct TachyonVector vector;
while (tachyon_query_next_vector(vector_statement, &vector)) {
    printf(
        "Result: (%lu, %lu)\n",
        vector.timestamp,
        vector.value.uinteger64
    );
}

// Prepares a new aggregate query
struct TachyonQuery *sum_statement_out = NULL;
tachyon_query_prepare(
    connection,
    "sum(latency{service = \"web\"})",
    &start,
    NULL, // end timestamp NULL means query until the last timestamp
    &sum_statement_out
);
struct TachyonQuery *sum_statement = (struct TachyonQuery *)
```

```
sum_statement_out;

// Gets the result as a scalar and outputs it
union TachyonValue value;
tachyon_query_next_scalar(sum_statement, &value);

// Outputs "Result: 4950"
printf("Result: %lu\n", value.uinteger64);

// Close statements and connection
tachyon_query_free(vector_statement);
tachyon_query_free(sum_statement);
tachyon_free(connection);
```

Typical Scenarios

Remember to install the library and include the TachyonDB header:

```
#include <TachyonDB.h>
```

Creating and Connecting to a Database

```
uint8_t tachyon_new(const char *db_dir, void **out);
```

If no database exists at `db_dir`, an empty database will automatically be created at `db_dir`. The return value indicates a success or failure that will be returned in the `out` parameter. The values will follow the function documentation.

Closing Connection to a Database

When finished, make sure to free the connection's memory by calling:

```
void tachyon_free(struct TachyonConnection *connection);
```

Deleting a Database

To delete a database, manually delete the DB_DIR directory and all of its contents.

Creating a Stream

```
uint8_t tachyon_stream_create(
```

```c
    struct TachyonConnection *connection,
    const char *stream,
    TachyonValueType value_type,
    void **out
);
```

The `value_type` can be assigned from:

```c
enum TachyonValueType {
  TachyonValueType_Integer64,
  TachyonValueType_UInteger64,
  TachyonValueType_Float64,
};

typedef uint8_t TachyonValueType;
```

Inserting Data

1. Insert data using:

```c
void tachyon_insert_prepare(
    struct TachyonConnection *connection,
    const char *stream,
    void **out
);
```

2. The `TachyonValue` type is defined as:

```c
union TachyonValue {
    int64_t integer64;
    uint64_t uinteger64;
    double float64;
};
```

3. Once data insertion is finished, make sure to call:

```c
void tachyon_insert_flush(
    struct TachyonConnection *connection
);
```

Reading Data from a Database

1. Create a query using:

```c
struct TachyonQuery *tachyon_query_prepare(
    struct TachyonConnection *connection,
    const char *query,
    const TachyonTimestamp *start,
    const TachyonTimestamp *end,
    void **out
);
```

Note that `start` and `end` can be `null`. This can be thought of as setting `start` to zero and `end` to infinity. If both are `null`, this means all entries are included in the query.

2. Get the next entry using either of:

```c
bool tachyon_query_next_scalar(
    struct TachyonQuery *query,
    union TachyonValue *scalar
);

bool tachyon_query_next_vector(
    struct TachyonQuery *query,
    struct TachyonVector *vector
);
```

The output is written into the `scalar` or `vector` parameter.

The `bool` return value is `true` if there are more entries, otherwise `false`.

3. The `TachyonVector` type is defined as:

```c
struct TachyonVector {
    TachyonTimestamp timestamp;
    union TachyonValue value;
};
```

4. When finished, make sure to free the query's memory by calling:

```c
void tachyon_query_free(struct TachyonQuery *query);
```

User Manual: Python

Quickstart

To install the TachyonDB library, run:

```
$ pip install tachyondb-lib
```

The following code inserts the integers from 0-99 into a stream called latency{service= “web”} and queries the sum of the inserted data. This code shows how to deal with error handling functionality for the first call into the library then assumes that all subsequent calls succeed without errors.

```python
from tachyondb_lib import *

# Creates a new connection to a database at "./tmp/db"
try:
    connection = Connection("./tmp/db")
except e:
    print(e)

# Creates stream if it doesn’t exist
if not connection.check_stream_exists("""latency{service =
"web"}"""):
    connection.create_stream("""latency{service = "web"}""",
ValueType.UInteger64)

inserter = connection.prepare_insert("""latency{service = "web"}""")

for i in range(100):
    # Inserts data
    inserter.insert_uinteger64(
        timestamp = i,
        value = i
    )
# the stream latency{service = "web"} now contains
# timestamps 0 to 99 with their corresponding values

# Flushes the writer and persists data to disk
```

```python
inserter.flush()

# Prepares a new vector query
(timestamps, values) = connection.query(
    """latency{service = "web"}""",
    start = 0,
    end = None, # None means query until the last timestamp
).get()

# Outputs the timestamp and value of all inserted entries
for timestamp, value in zip(timestamps, values._0):
    print(timestamp, value)

# Prepares a new aggregate query
(_, values) = connection.query(
    """sum(latency{service = "web"})""",
    start = 0,
    end = None, # None means query until the last timestamp
).get()

# Outputs "Result: 4950"
print("Result: ", values._0[0])
```

Typical Scenarios

To install the TachyonDB library, run:

```
$ pip install tachyondb-lib
```

Make sure to import TachyonDB:

```python
from tachyondb_lib import *
```

Creating and Connecting to a Database

To connect to a database, use:

```python
connection = Connection(db_dir: str)
```

If no database exists at `db_dir`, an empty database will automatically be created at `db_dir`.

Deleting a Database

To delete a database, manually delete the DB_DIR directory and all of its contents.

Creating a Stream

```
Connection.create_stream(
    self,
    name: str,
    value: ValueType
) -> None
```

Inserting Data

1. To insert data into a stream, first create an Inserter:

```
Connection.prepare_insert(
    self,
    stream: str,
) -> None
```

2. Then use the following Inserter function to insert data:

```
Inserter.insert(
    self,
    timestamp: int,
    value: ValueType
) -> None
```

3. The type `ValueType` is defined as:

```
class ValueType(Enum):
     SignedInteger = 1
     UnsignedInteger = 2
     Float = 3
```

4. After data insertion, call flush:

```
Inserter.flush(self) -> None
```

Reading Data from a Database

1. Make a query using:

```
Connection.query(
     self,
    query: str,
    start: Optional[int],
    end: Optional[int],
) -> QueryResult
```

Note that `start` and `end` can be `None`. This can be thought of as setting `start` to zero and `end` to infinity. If both are `None`, this means all entries are included in the query.

2. Get the results using:

```
QueryResult.get(
    self
) -> ([Timestamp], [ValueType])
```

The timestamp and value results can be zipped together and then iterated through.

Exceptions and Alternative Scenarios

Exceptions

Syntax Errors

TachyonDB expects syntax to be well-formed. If the user provides a query string or CLI command that is invalid, the user receives a syntax error and the action is not completed. Possible exceptions include:

- **E6: User makes a syntax error in a query.**
- **E7: User makes a syntax error in a CLI command.**

Invalid Data

TachyonDB has certain assumptions about the data being inserted into the system. When these assumptions are violated in an insert operation, the entries are not inserted and the user receives an error. Possible exceptions include:

- **E4: User inserts incorrectly-typed data.**
    - TachyonDB supports only entries with numerical values. This can include unsigned, signed, and floating point numbers. Additionally, all values in a TachyonDB stream must be of the same type. It is possible for different streams with the same metric and/or labels to have different data types.
- **E5: User inserts out-of-order data.**
    - TachyonDB expects points to be written in chronological order.

Invalid Arguments

TachyonDB assumes that the arguments provided to a command or function are properly formed and valid. When this assumption is violated, TachyonDB gives the user an error. Possible exceptions include:

- **E1: User does not connect to a database before writing queries/commands.**
- **E2: User deletes a non-existent database.**
- **E3: User reads from a non-existent stream.**

System/Architecture

TachyonDB assumes it is running on sufficient hardware, has the necessary permissions, and is not tampered with on the file system. When this assumption is violated, the system's behavior is undefined unless otherwise noted. Possible exceptions include:

- **E8: System is not running on the target architecture.**

- **E9: User application does not have permission to make memory allocations.**
- **E10: User application does not have read/write access.**
- **E11: Database file is manually edited or deleted without using the interface.**
    - TachyonDB expects that files or directories in DB_DIR are not modified. If a file or directory is changed, the user might receive invalid data or the read operation may fail. Thus, the user is strongly warned not to manually modify any database files.
- **E12: A database file is corrupted.**
- **E13: Computer crashes during execution.**
    - TachyonDB expects that the computer does not crash during execution. If this does happen, most data is preserved but some may be lost.
- **E14: Computer runs out of memory.**
- **E15: Computer runs out of storage.**
    - TachyonDB expects when writing time series data for there to be sufficient disk space to store the data. If this is not the case, an out of storage error is thrown and data might not be persisted.

Concurrency

TachyonDB assumes there is at most one process writing data and at most one connection per thread. When this assumption is violated, the system's behavior is undefined. Possible exceptions include:

- **E16: Multiple threads in a user application use the same database connection at the same time.**
    - Each thread should create its own connection.
- **E17: Multiple threads or user applications insert data into the same stream in the same database at the same time.**

Setup and Configuration

TachyonDB requires some configuration to be done by users, specifically for the GUI. If these configurations aren't done correctly, the user will be presented with an error. Possible exceptions include:

- **E18: User runs GUI without pointing it to a valid backend instance.**
    - An error is displayed on the web Grafana frontend saying that it can't connect to the specified backend

Alternative Scenarios

V1: Database is used for non-time series data.
TachyonDB requires entries to consist of a timestamp and a numerical value. The user could technically log any form of numerical data by setting the timestamp value to the current time and ignoring it in the future. Since TachyonDB is optimized for time series data, it may perform worse in this scenario.

V2: Database is used as the primary database for a non-edge device.
While TachyonDB is optimized to run in low compute environments, such as edge devices, it can be used for any application that records time series data. However, this is not the primary purpose of TachyonDB, so its performance as a primary database isn't guaranteed.

V3: A value is used as a label — leading to a separate stream for each inserted value.
It is not recommended to use values as labels (or any label value where the set of possible values is not known) due to the number of streams that may be created. Streams have non-negligible overhead, and for each unique set of metric name + labels, a new stream is created. This would result in slower and more complex queries to access the data.

V4: Database is used as a key-value store.
While TachyonDB is specialized for storing multiple entries of data over time, it can be used to store a single entry per stream, with key as the stream and the value as the entry.

V5: User creates a uniquely-named metric for each stream and never uses labels.
The user can create a separate metric for every stream and not use labels if the user desires. This is not recommended though, since using labels with metrics helps the user more easily categorize streams.

V6: User inserts all data into a single metric and differentiates streams purely using labels.
The metric name is syntactic sugar. Although there are technically no downsides to avoiding the use of labels, having logical groupings of streams helps to write maintainable and understandable queries.

V7: User application uses the CLI instead of the client library.
Although it is possible to execute queries by executing the Tachyon CLI from a user application, it is not recommended since it is noticeably slower than using the library.

V8: User queries data and manually performs aggregation operations on the queried data instead of directly using TQL's aggregate functions.

The user is free to query data using TQL and operate on the results afterwards inside the user application. However, this may be slower than using TQL's built-in aggregation and binary operations.

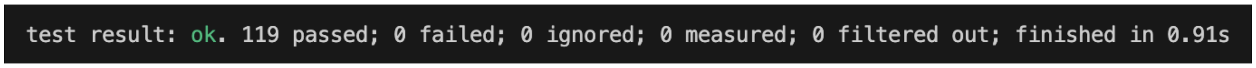
V9: User installs TachyonDB binaries manually.

The user is free to clone the TachyonDB repository and compile the CLI and client library binaries locally. However, it is easier to install them through the methods mentioned previously.

Verification and Validation

Testing

TachyonDB's correctness was verified with a comprehensive set of unit tests and end-to-end tests. Unit tests covered critical paths and edge cases for key components including but not limited to query parsing and execution, file indexing, reading and writing, interpolation, and compression. End-to-end tests were implemented to validate user flows and integration. Test coverage was required for all code changes, and all tests were regression tested as part of TachyonDB's CI/CD pipeline.



```
test result: ok. 119 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.91s
```

Benchmarks

To evaluate the performance of Tachyon, a set of benchmarks was conducted comparing it to several widely used embedded and time-series databases: SQLite, TimescaleDB, DuckDB, and QuestDB. The focus of the evaluation was on read latency, insert performance, and storage efficiency.

Benchmark Setup

All benchmarks were implemented using Criterion.rs, a benchmarking framework for Rust that provides statistically robust and reproducible measurements. Each test was repeated multiple times to minimize variance and ensure accuracy.

Benchmarks were executed on two different devices:

- Raspberry Pi 4B (64-bit) with 2GB of RAM and a 1.4GHz quad-core processor
- M1 Max Macbook Pro 2021. 32GB of RAM

This hardware was selected to reflect performance in resource-constrained environments, such as edge or embedded systems, as well as usage on a higher-powered machine.

Datasets

Two distinct time-series datasets were used:

1. *Memory Dataset*

This dataset was generated from memory usage statistics collected on a personal laptop (M1 Max Macbook Pro 2021). It represents a high-frequency telemetry stream and was used to evaluate short-range queries and rapid insert operations.

2. *Voltage Dataset*
   This dataset consists of voltage readings taken over a period of five years from a residential house in France. Due to its size and temporal range, it was used to test large-scale insert throughput and performance on long-range queries.

Systems Evaluated

The following systems were included in the benchmark:

- **Tachyon** – evaluated using the version corresponding to PR #86.
- **SQLite** – used with default configuration.
- **TimescaleDB** – a PostgreSQL extension optimized for time-series data.
- **DuckDB** – a lightweight, in-process analytical database.
- **QuestDB** – a specialized time-series database supporting SQL and high-ingest throughput.

Metrics Collected

- **Read Latency** – measured as the time required to execute a range query over the dataset.
- **Insert Latency** – measured as the time to fully ingest each dataset.
- **Size on Disk** – measured as the total disk usage after all data had been inserted.

Results

The following benchmarks are the ones that were conducted on a Raspberry Pi 4B. This models a traditional use case of using an IOT device in a low-compute environment.

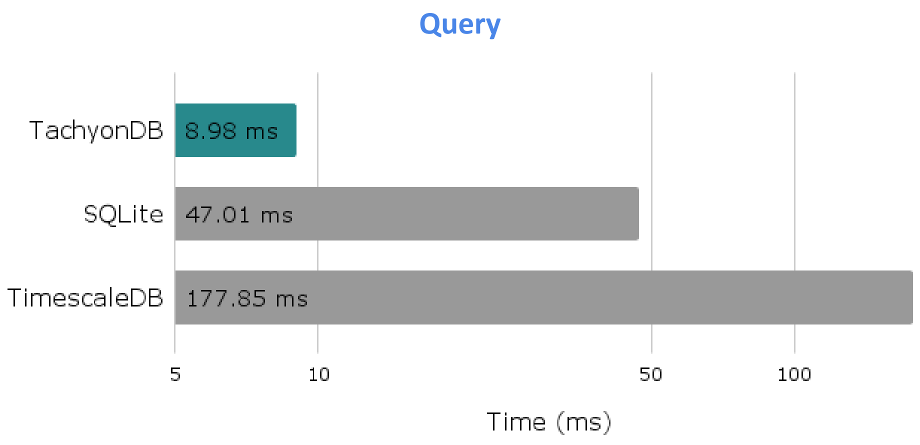


Figure 8: Time to Read 120k Entries on Raspberry Pi 4B

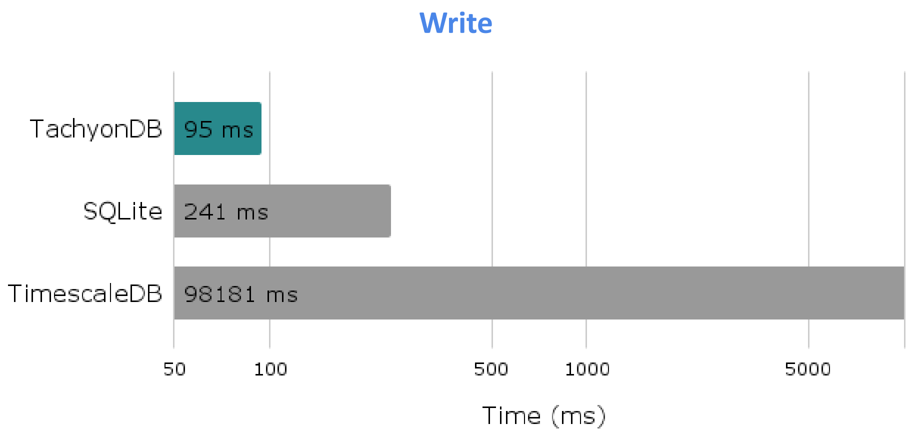


Figure 9: Time to Write 120k Entries on Raspberry Pi 4B

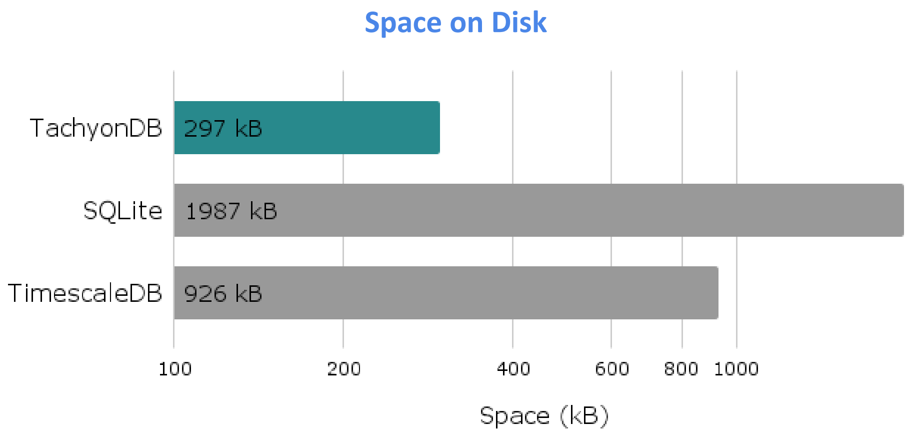
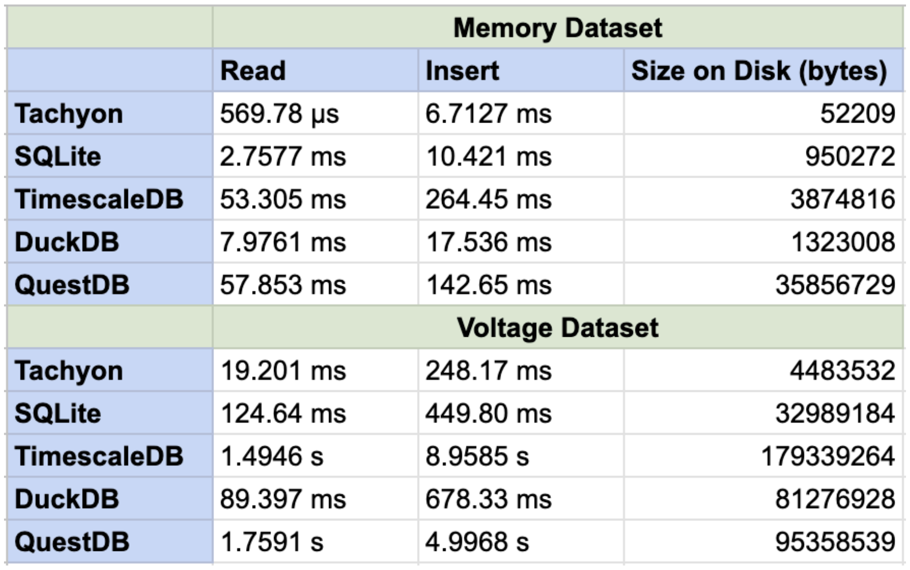


Figure 10: Space to Store 120k Entries on Raspberry Pi 4B

Due to compatibility constraints, resource limitations, and system requirements of the Raspberry Pi, additional benchmarks were conducted on a MacBook across several other databases. The results for these database instances are presented in **Figure 11**.



	Memory Dataset		
	Read	**Insert**	**Size on Disk (bytes)**
Tachyon	569.78 µs	6.7127 ms	52209
SQLite	2.7577 ms	10.421 ms	950272
TimescaleDB	53.305 ms	264.45 ms	3874816
DuckDB	7.9761 ms	17.536 ms	1323008
QuestDB	57.853 ms	142.65 ms	35856729
	Voltage Dataset		
Tachyon	19.201 ms	248.17 ms	4483532
SQLite	124.64 ms	449.80 ms	32989184
TimescaleDB	1.4946 s	8.9585 s	179339264
DuckDB	89.397 ms	678.33 ms	81276928
QuestDB	1.7591 s	4.9968 s	95358539

Figure 11: Reading, Insertion and Size Benchmarks for TachyonDB and Comparable Databases on the Macbook M1 Max

Citations

[1] “What is time series data?: Definition, examples, types & uses,” InfluxData, https://www.influxdata.com/what-is-time-series-data/ (accessed Jul. 11, 2024).