GridDB Fundamentals

How to Build IoT Applications using GridDB

Table of Contents

Chapter 1. An Introduction to IoT and Time-Series Data

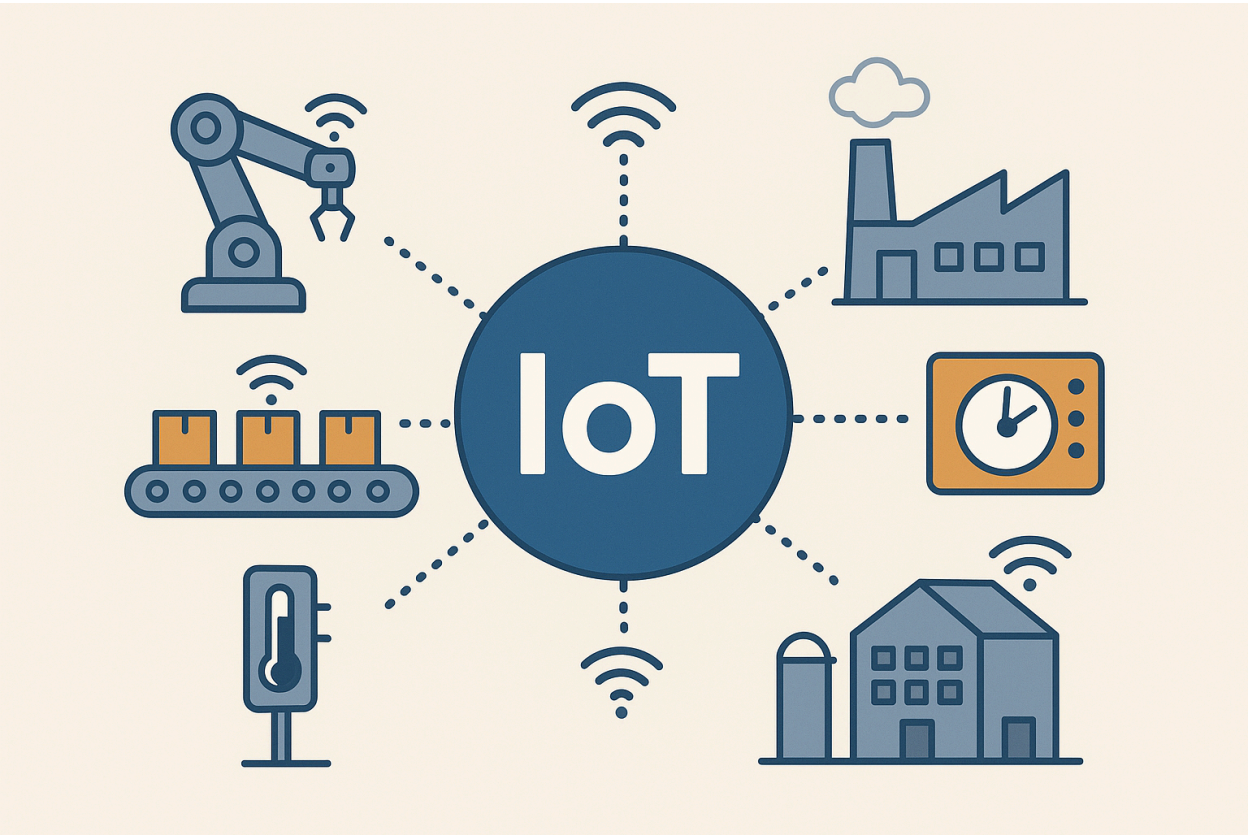
The world is becoming a network of sensors. From the watch on your wrist tracking your heart rate, to the thermostat on your wall adjusting to the weather, and the industrial machinery on a factory floor predicting its own maintenance needs, we are surrounded by the **Internet of Things (IoT)**. At its core, IoT is a vast, interconnected system of physical devices that collect and exchange data about the world around them.

This explosion of connected devices is not just a novelty; it's a revolution. It enables real-time insights, intelligent automation, and efficiencies that were previously unimaginable. But this revolution is built on a tidal wave of data—a constant, high-velocity stream of information that presents a unique and formidable engineering challenge. This chapter introduces the nature of that challenge and why a specialized database like GridDB is essential for building robust, scalable IoT applications.

1.1 The IoT Data Deluge

An IoT device is fundamentally a data producer. A single sensor in a smart factory might report temperature, vibration, and pressure every 100 milliseconds. A connected vehicle might stream its location, speed, and fuel status every second. Now, multiply that by thousands or even millions of devices. The result is a data management problem characterized by three key attributes:

- **Volume:** The sheer amount of data is immense. A modest deployment of 1,000 sensors reporting every second generates over 86 million data points per day. Traditional storage systems can quickly become overwhelmed by the cost and complexity of managing this scale.
- **Velocity:** Data arrives at an extremely high speed. The database must be capable of ingesting millions of writes per second without falling behind. This is often called the *ingestion rate*, and for many IoT systems, it is the most critical performance metric.
- **Variety:** While much of the data is uniform *time-series* data (a value and a timestamp), the devices themselves have different attributes and metadata. A temperature sensor has a location and a calibration date, while a smart meter has a customer account number and a service tier. The system must manage both the fast-changing sensor data and the slow-changing descriptive data efficiently.



1.2 Why Traditional Databases Struggle

When faced with this data deluge, an engineer's first instinct might be to reach for a familiar tool, like a relational database (e.g., PostgreSQL, MySQL) or a general-purpose NoSQL database. However, these systems were designed for different problems and often falter under the unique pressures of IoT workloads.
Relational databases, built on a foundation of structured tables and complex JOIN operations, are poorly suited for the task. Ingesting billions of simple, timestamped rows can be inefficient, and querying data across a time range—a fundamental IoT operation—can be surprisingly slow. Furthermore, their rigid schemas make it cumbersome to adapt as new device types with different data structures are added to the network.

While some NoSQL databases offer better scalability, they often lack the specific features needed for time-series analysis. A general-purpose document store or key-value store doesn't inherently understand time, forcing developers to build complex, application-level logic for common tasks like data aggregation, downsampling, or setting automatic data expiry policies. This leads to brittle, unperformant systems.

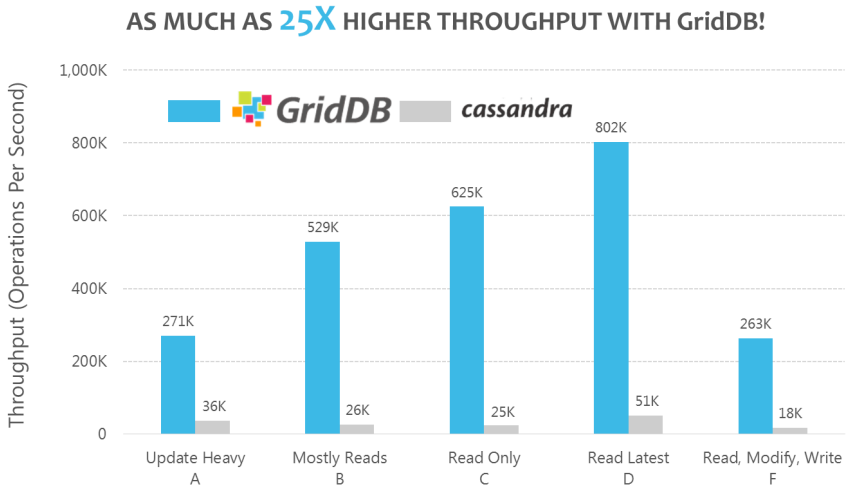
The bottom line is that IoT isn't just another "big data" problem. It's a problem that demands a purpose-built solution.

1.3 Introducing GridDB: A Database Built for IoT

GridDB is an open-source, highly scalable time-series database engineered specifically to solve the challenges of IoT. It was designed from the ground up with the understanding that in IoT, **time is the primary axis**. This principle influences every aspect of its architecture.
GridDB combines the speed of in-memory processing with the durability of disk storage, all within a distributed architecture that can scale horizontally by simply adding more nodes. Let's look at its core architectural tenets:

- **Key-Container Data Model:** Instead of tables or documents, GridDB uses a **Container** model. A container holds a collection of rows with a defined schema, similar to a table. You access containers with a key (its name). This model is simple, powerful, and maps naturally to IoT device data.
- **Time-Series First:** GridDB has a special TIME_SERIES container type where the timestamp is the primary key. This container is highly optimized for complex time-based queries, such as fetching data from the last hour, calculating moving averages, or interpolating missing values.
- **In-Memory Architecture:** GridDB prioritizes performance by leveraging a hybrid in-memory and disk-based system. Hot data (the most recent, frequently accessed information) is kept in memory for lightning-fast reads and writes, while older, colder data is flushed to disk for cost-effective long-term storage.
- **Elastic Scalability:** GridDB is a distributed database. As your data volume and ingestion rate grow, you can scale the system horizontally by adding more commodity servers to the cluster. This peer-to-peer architecture ensures there is no single point of failure.

This specialized design yields significant performance benefits. In the industry-standard Yahoo! Cloud Serving Benchmark (YCSB), GridDB has been shown to outperform other popular NoSQL databases like Cassandra by up to 25x on read-heavy workloads and 7x on write-heavy workloads, which are typical patterns in IoT applications. Read more about that in the Cassandra White Paper: https://griddb.net/en/docs/Fixstars_NoSQL_Benchmarks.pdf



1.4 GridDB in Action: A Smart Factory Use Case

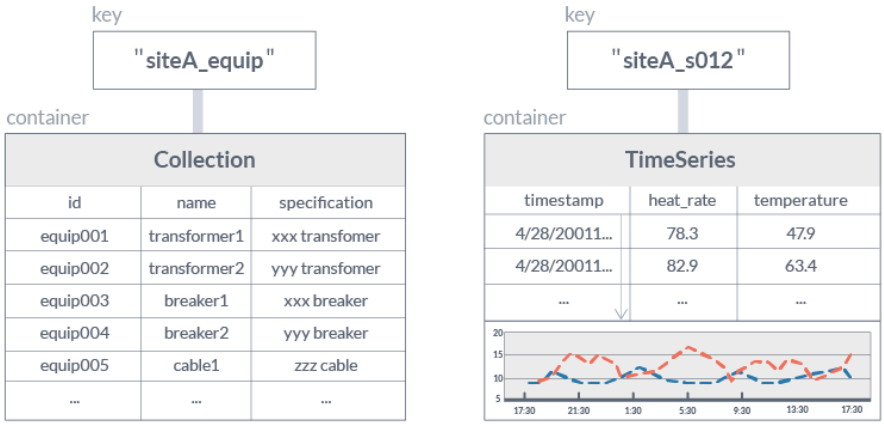
To make these concepts concrete, let's consider a practical scenario. Imagine you are tasked with building the data backend for a smart factory with 5,000 machines, each equipped with a temperature sensor. Each sensor reports its reading once per second.

The Scale of the Problem:

- 5,000 writes per second
- 18,000,000 writes per hour
- 432,000,000 writes per day

How would we model this in GridDB? We would use two types of containers:

1. A **TIME_SERIES Container for each sensor.** We could create containers named temp_sensor_0001, temp_sensor_0002, and so on. Each container would have a simple schema like (timestamp TIMESTAMP, temperature DOUBLE). This one-container-per-sensor model is highly efficient for write-heavy workloads, as each stream of data is written to its own dedicated location.
2. A **COLLECTION Container for metadata.** A COLLECTION container is suited for storing static or infrequently updated data about the devices. We could create a container named machine_metadata with a schema like (machine_id STRING, location STRING, install_date TIMESTAMP).



2 types of container, collection and time series
A container is like the RDB table of a row/column

This design cleanly separates the high-velocity event data from the low-velocity descriptive data. With this structure, you get the best of both worlds: the speed and flexibility of a **NoSQL** database for data ingest, combined with a powerful and familiar **SQL** query language for analysis.

For example, to find the average temperature for a specific sensor over the last hour, the query is simple and intuitive:

```
SELECT AVG(temperature)
FROM temp_sensor_0001
WHERE timestamp > NOW() - 1 HOUR;
```

Note: Thinking in Containers: The "No JOINs" Philosophy
If you come from a relational database background, the most important mental shift is to embrace a **denormalized** data model. Though GridDB supports JOIN operations across containers, it is generally not the best practice when designing your application.

Instead of joining tables at query time, you model your data to avoid the need for joins. This often means storing related information together in the same container or, as in our example, separating data by its access pattern. This approach is fundamental to building high-performance systems in the world of distributed databases.

1.5 Summary

The Internet of Things represents a paradigm shift in how we interact with the physical world, driven by an unprecedented volume and velocity of data. Traditional databases were not designed for the unique demands of time-series data at this scale.

GridDB provides a purpose-built solution, offering a scalable, performant, and developer-friendly platform for IoT applications. Its key-container model, first-class time-series support, and distributed architecture make it an ideal foundation for turning sensor data into actionable insights.

In the next chapter, we will dive deeper into the core concepts of GridDB. We'll explore its architecture in more detail, get hands-on with its data model, and set up your first GridDB cluster.

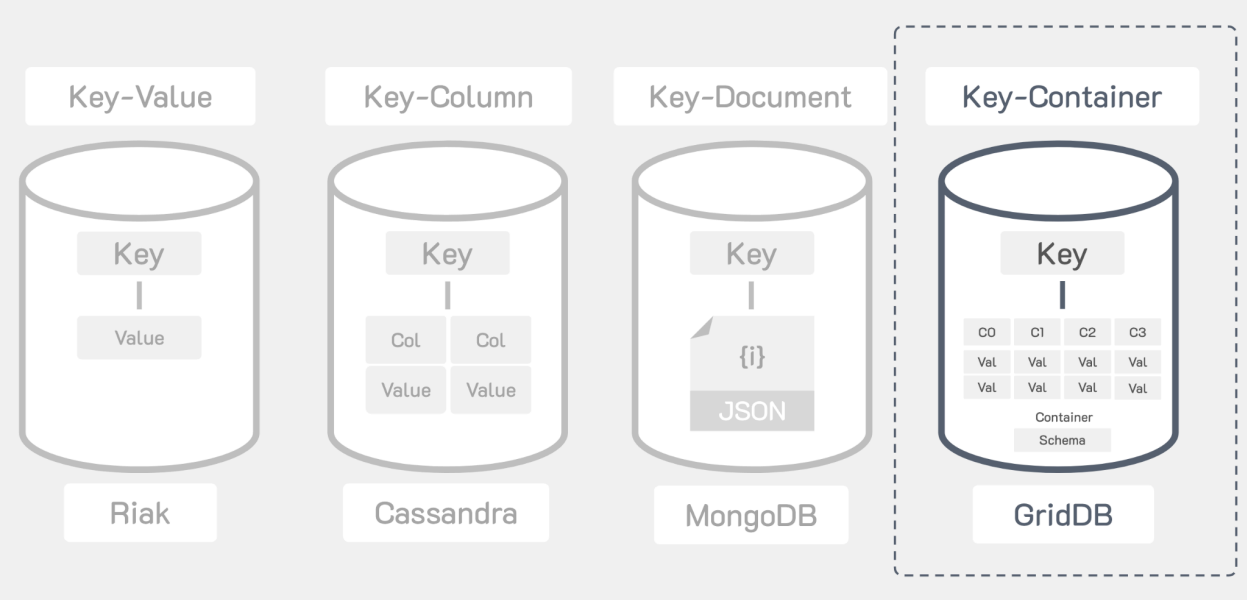
Chapter 2. Core Concepts: Containers and Data Modeling

In a traditional relational database, schema design often feels like an exercise in normalization—breaking data into distinct tables to eliminate redundancy, then reassembling it with JOINs. When working with a distributed time-series database like GridDB, you must unlearn this instinct. Here, the rules are different, and they are driven by the relentless demands of performance and scale.

Effective data modeling in GridDB isn't about abstract normalization rules; it's a practical discipline focused on one primary goal: **designing for your access patterns.** How you structure your data will directly determine how fast you can write it and how efficiently you can query it. This chapter introduces the core building blocks of a GridDB schema—Containers—and the fundamental principles for modeling data in a high-throughput IoT environment.

2.1 The Container: Your Basic Unit of Storage

The fundamental unit of organization in GridDB is the **Container**. You can think of a container as being analogous to a table in a relational database. It has a name, and it holds a collection of rows, where each row conforms to a predefined schema (i.e., a set of columns with specific data types).



GridDB has two primary types of containers, each optimized for a different kind of data:

- **TIME_SERIES Containers:** These are the workhorses of any IoT application. They are specifically designed to store timestamped event data. In a TIME_SERIES container, one column **must** be designated as the RowKey and be of the TIMESTAMP type. This structure allows GridDB to perform highly optimized time-based operations, such as rapidly ingesting new data points or querying for data within a specific time range.
- **COLLECTION Containers:** These are general-purpose containers suitable for storing data that isn't primarily defined by time. This is where you store metadata, device attributes, or configuration information. In a COLLECTION container, the RowKey can be of type STRING, INTEGER, LONG, or TIMESTAMP.

The most critical pattern in GridDB schema design is the clean separation of these two data types. High-velocity, time-stamped sensor readings belong in TIME_SERIES containers. Low-velocity, descriptive metadata belongs in COLLECTION containers.

2.2 The First Principle: Design for a JOIN-less World

As we noted in Chapter 1, GridDB *does* support JOIN operations across containers, but they are discouraged. Forgoing the use of JOINs can be a deliberate design choice that enables horizontal scalability and predictable query performance. A JOIN is an expensive operation in a distributed system, and by designing it out of the database engine, GridDB encourages you to adopt a more scalable data model.
This leads to a simple but powerful guideline: **denormalize your data**. Instead of striving to eliminate every piece of redundant data, you should intentionally duplicate data where it makes sense for your queries.

Let's revisit our smart factory example. We have sensor readings (time-series data) and machine information (metadata). In a relational world, you might have a `readings` table and a `machines` table, and you would JOIN them on machine_id to find out which location a particular reading came from.

In GridDB, you handle this differently. The sensor data goes into a TIME_SERIES container. The machine metadata (ID, location, installation date) goes into a COLLECTION container. If a query needs to know the location of a sensor that triggered an alert, your application would perform two separate, simple queries:

1. Fetch the alert record, which contains the sensor_id.
2. Fetch the metadata for that sensor_id from the machine_metadata container.

These two fast, indexed lookups are far more scalable than a single, complex distributed join.

Warning: The Perils of a "God" Container
A common anti-pattern for developers new to GridDB is to create a single, massive TIME_SERIES container to hold readings from all sensors. While this seems simple initially, it creates a major performance bottleneck. Because GridDB provides ACID guarantees at the container level, all writes to this single container become serialized, creating contention and limiting your ingestion throughput. The key to performance is to spread the load across many containers.

2.3 The "One Container Per Sensor" Model

The most effective and scalable pattern for modeling high-frequency IoT data in GridDB is the **one container per sensor** model. Instead of a single container for all temperature readings, you create a separate TIME_SERIES container for *each individual temperature sensor*.

For our factory with 5,000 sensors, this means you would create 5,000 TIME_SERIES containers. This might sound like a lot, but it is the key to unlocking GridDB's performance. This approach provides several major advantages:

- **Massive Write Parallelism:** With each sensor writing to its own dedicated container, writes can happen in parallel across the cluster without contention. This is how you achieve ingestion rates of millions of data points per second.
- **Query Isolation:** When you query for data from a single sensor, you are reading from a small, dedicated container. This is significantly faster than searching for that sensor's data within a massive, shared container.
- **Schema Flexibility:** If a new version of a sensor starts reporting an additional data field, you only need to define a new schema for that sensor's new container. It has no impact on the thousands of other existing containers.

To manage this many containers, a consistent **naming convention** is essential. For example, you might name your containers using a pattern like DeviceType_FacilityID_DeviceID, which would result in names like:

- temp_F01_A34C
- vibration_F01_B81A
- pressure_F02_A35B

2.4 A Concrete Schema Example

Let's formalize the schema for our industrial factory. We'll define the schemas using Java class syntax, which is how the GridDB Java client maps objects to container rows.

First, the schema for our individual sensor data containers. This is a "narrow" model, with one row per reading.

```java
// Schema for a TIME_SERIES container, e.g., "temp_F01_A34C"
public class SensorReading {
    @RowKey Date timestamp; // The time of the reading is the RowKey
    double value;   // The sensor's value
    String status;  // e.g., "NORMAL", "WARNING"
}
```

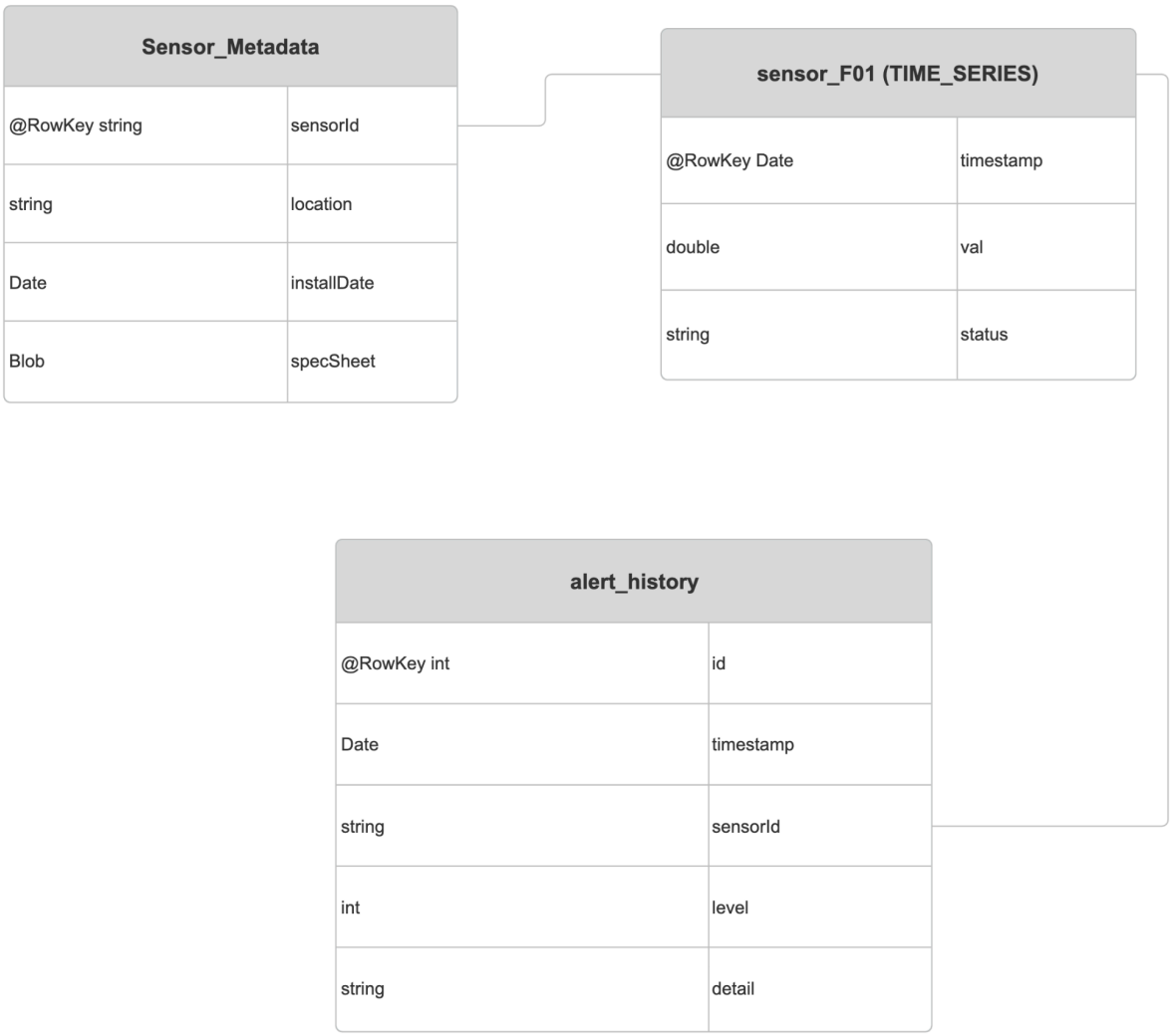
Next, the schema for our COLLECTION container that stores metadata about each machine or sensor.

```
// Schema for a COLLECTION container
public class Sensor_Metadata {
    @RowKey String sensorId; // e.g., "temp_F01_A34C"
    String location;  // e.g., "Building 3, Line 2"
    Date installDate;
    Blob specSheet;   // Store binary data like a PDF spec sheet
}
```

Finally, we might have a COLLECTION container for storing discrete alert events.

```
// Schema for a COLLECTION container named "alert_history"
public class Alert {
    @RowKey int id;
    Date timestamp;
    String sensorId;    // Which sensor triggered the alert
    int level;          // e.g., 1=Warning, 2=Critical
    String detail;      // Description of the alert
}
```

This multi-container, denormalized design is the blueprint for a scalable IoT application in GridDB.



2.5 Summary

Effective schema design in GridDB requires a shift in thinking away from the relational model. By embracing a JOIN-less, denormalized approach and leveraging the "one container per sensor" pattern, you can build a data architecture that is optimized for the extreme write and query loads of a large-scale IoT system.
The key takeaways are:

Practice	Benefit
One TIME_SERIES container per sensor	Enables massive write parallelism and fast, isolated queries.
Use COLLECTION containers for metadata	Cleanly separates high-velocity data from slow-changing attributes.
Denormalize to avoid JOINs	Designs for scalability by favoring multiple simple lookups over one complex operation.
Use clear naming conventions	Makes managing thousands of containers programmatically feasible.

In the next chapter, we'll move from theory to practice. You'll learn how to install GridDB, connect to it with a client, and perform your first create, read, update, and delete (CRUD) operations.

Chapter 3. Working with Time-Series Data

With a solid data model in place, it's time to focus on the heart of any IoT application: the time-series data itself. GridDB isn't just a place to store timestamped records; it's an engine built to manage and analyze that data over its entire lifecycle. This involves more than just writing and reading—it includes handling data retention, performing complex time-based queries, and running real-time analytics.

In this chapter, we'll explore the specialized features that GridDB provides for working with TIME_SERIES containers. You'll learn how to automatically manage data storage with expiry policies and how to leverage GridDB's powerful time-series functions to sample, aggregate, and analyze your data efficiently.

3.1 Managing the Data Lifecycle: Partitioning and Expiry

In many IoT systems, the value of a data point diminishes over time. A temperature reading from 10 seconds ago is critical, but a reading from two years ago is often less relevant. Storing every data point forever is expensive and degrades query performance.

GridDB solves this problem by integrating data expiry directly with its **partitioning** feature. Instead of deleting row-by-row, GridDB drops an entire *partition* (a chunk of data) once it's considered old. This is an essential feature for managing storage and maintaining high performance.

To use expiry, you must also define a partitioning rule. This is done via SQL when you first create your container:

```sql
SQL
CREATE TABLE IF NOT EXISTS temp_sensors (
  timestamp TIMESTAMP NOT NULL PRIMARY KEY,
  value FLOAT
)
WITH (
  expiration_type = 'PARTITION',
  expiration_time = 10,
  expiration_time_unit = 'DAY'
)
PARTITION BY RANGE (timestamp)
EVERY (1, DAY);
```

Let's break down the new settings in this `CREATE TABLE` command:

- `PARTITION BY RANGE (timestamp) EVERY (1, DAY)` This command tells GridDB to create a new data partition for each day's worth of data (e.g., "Jan 1", "Jan 2", "Jan 3"). This is the "partitioning rule."
- `WITH ( ... )` This block defines the "expiry rule" that works on those partitions.
    - `expiration_time = 10` & `expiration_time_unit = 'DAY'`: This sets the retention period. We want to keep data for 10 days.
    - `expiration_type = 'PARTITION'`: This is the key. It tells GridDB to expire the *entire partition* at once, not individual rows.

How it works:

With this setup, the entire "Jan 1" partition will only be dropped from the database when *all* of its data (including the data from 23:59 on Jan 1) is older than 10 days. This means on "Jan 12", the "Jan 1" partition becomes eligible for deletion.

This simple SQL setting offloads a massive operational burden. You no longer need complex cleanup scripts. GridDB handles the data pruning for you, ensuring your containers remain lean and performant.

3.2 Querying with Time: SQL Time-Series Functions

Standard SQL can be clumsy for time-series queries. Answering "What was the average temperature every 5 minutes?" often requires complex window functions.

GridDB's **SQL** interface extends familiar syntax with powerful functions designed specifically for time-series analysis. These allow you to express complex time-based operations in a clear, concise, and standard way.

Downsampling with GROUP BY RANGE

One of the most common IoT tasks is downsampling—reducing high-frequency data to analyze or visualize it. Instead of a proprietary function, GridDB uses the elegant `GROUP BY RANGE` clause.

Imagine a sensor reports every second, but you only need the *maximum* value every five minutes for a dashboard. You can do this directly in the database:

```sql
-- Select the maximum temperature for every 5-minute interval
-- over the past 24 hours.
SELECT MAX(value)
FROM temp_F01_A34C
WHERE timestamp > NOW() - 1 DAY
GROUP BY RANGE timestamp EVERY (5, MINUTE);
```

This single query replaces a significant amount of application-level code and is much more intuitive than traditional SQL.

Note: Interpolation for Missing Data
Real-world sensors are unreliable. They drop connections or fail to report, leaving gaps in your timeline. GridDB handles this by adding a `FILL` clause to the `GROUP BY RANGE` query.

You no longer need a separate `TIME_INTERPOLATED` function. You can perform linear interpolation to fill gaps and create a continuous data set, which is perfect for smooth graphs or feeding ML models.

```sql
-- Get the average value every 10 seconds,
-- and fill any missing 10-second intervals using linear interpolation.
SELECT AVG(value)
FROM temp_sensor_0001
WHERE timestamp > NOW() - 1 HOUR
GROUP BY RANGE timestamp EVERY (10, SECOND)
FILL (LINEAR);
```

Other `FILL` options include `NULL` (fill with NULL) and `PREVIOUS` (carry the last known value forward).

3.3 Aggregation and Real-Time Analytics

Beyond sampling, TQL provides a full set of standard aggregation functions that work seamlessly with time-series data. You can calculate averages, find minimums and maximums, compute standard deviations, and more, all on the server side.

These functions are the building blocks of real-time monitoring and analytics. For example, you could build a dashboard that monitors the health of all machines in a facility by running queries like these:

```sql
-- Get the average temperature across all sensors in the last 10 minutes
-- (This would be run in a loop for each sensor container by the application)
SELECT AVG(value) FROM temp_F01_A34C WHERE timestamp > NOW() - 10 MINUTE;

-- Find the highest vibration reading from a specific machine today
SELECT MAX(value) FROM vibration_F02_B81A WHERE timestamp > NOW() - 1 DAY;

-- Calculate the standard deviation of pressure to detect instability
SELECT STDDEV(value) FROM pressure_F01_A35B WHERE timestamp > NOW() - 1 HOUR;
```

By performing these calculations inside the database, you minimize the amount of data that needs to be transferred over the network. Instead of pulling millions of raw data points into your application to compute an average, you send a simple query and get back a single number. This is a fundamental principle of scalable IoT architecture.

3.4 Optimizing for Bulk Operations

When interacting with GridDB, especially during high-throughput ingestion or large data exports, it's always more efficient to work with data in batches. Sending one row at a time incurs significant network overhead.

All GridDB client libraries provide methods for bulk operations. For example, the Java client has:

- **multiPut()**: Puts a list of rows into a container in a single call. This is essential for achieving high ingestion rates from your sensor data collectors.
- **multiGet()**: Retrieves multiple rows based on a list of row keys.

Always favor these batch operations over single-row operations in a loop. A simple change from put() in a for loop to a single multiPut() call can improve your write performance by an order of magnitude.

3.5 Summary

Working effectively with time-series data goes beyond simple storage. A robust IoT platform requires tools to manage the data lifecycle, perform complex time-based queries, and run analytics in real time. GridDB provides these capabilities as first-class features of the database.

Key takeaways from this chapter include:

Feature	Benefit
Data Expiry Policies	Automatically manages storage, saving cost and maintaining performance.
TIME_SAMPLING Function	Simplifies downsampling and data resolution reduction directly in a query.
Server-Side Aggregations	Minimizes data transfer and enables efficient, real-time analytics.
Bulk Operations (multiPut)	Maximizes ingestion throughput by reducing network overhead.

By mastering these features, you can build applications that are not only scalable and performant but also capable of extracting meaningful insights from the relentless flow of IoT data.

In the next chapter, we will get our hands dirty with code. We'll walk through detailed examples in Java and Python, showing you how to connect to GridDB and perform the fundamental write and read operations that form the foundation of any IoT application.

Chapter 4. Writing Data to GridDB

With your schema designed and your time-series management strategy in place, you're ready for the most fundamental operation in any IoT system: writing data. For most IoT applications, the write throughput—often called the **ingestion rate**—is the single most critical performance metric. The system must be able to absorb a relentless stream of data from potentially millions of devices without faltering.

In this chapter, we'll get hands-on with code. You will learn the primary methods for writing data to GridDB, focusing first on the high-performance native Java client. We will then explore how to achieve the same results using the Python client, the JDBC driver, and the language-agnostic Web API.

4.1 The Core Pattern: Writing with the Java Client

The native Java client offers the highest performance and the most features. It's the recommended choice for building the core data ingestion services in a demanding production environment.

Connecting to the GridDB Cluster

The first step in any application is to establish a connection to the GridDB cluster. This is done by creating a GridStore instance, configured with the properties of your cluster.

```java
import com.toshiba.mwcloud.gs.GridStore;
import com.toshiba.mwcloud.gs.GridStoreFactory;
import java.util.Properties;

// Set up the connection properties
Properties props = new Properties();
props.setProperty("notificationMember", "127.0.0.1:10001"); // Address of a cluster
node
props.setProperty("clusterName", "myCluster");
props.setProperty("user", "admin");
props.setProperty("password", "admin");

// Get an instance of the GridStore
GridStore store = GridStoreFactory.getInstance().getGridStore(props);
```

Defining a Schema with a Java Class

As we saw in Chapter 2, you can map a Java class directly to a GridDB container schema. This object-oriented approach is intuitive and type-safe. Let's use our SensorReading class again.

```java
// This class defines the schema for our container
public class SensorReading {
    @RowKey Date timestamp;
    double value;
    String status;
}
```

Putting a Single Row

To write data, you first get a reference to a container and then put an object into it. The put operation is transactional. You must call commit() to save the changes to the database.

```java
// Get a reference to a TIME_SERIES container.
TimeSeries<SensorReading> container = store.getTimeSeries("temp_F01_A34C",
SensorReading.class);

// Create an instance of our data object
SensorReading reading = new SensorReading();
reading.timestamp = new Date(); // Use the current time
reading.value = 21.5;
reading.status = "NORMAL";

// Put the row into the container
container.put(reading);

// IMPORTANT: Commit the transaction to persist the data
container.commit();
```

Note: Auto-Commit
The container object has an setAutoCommit(true) method. While this can be convenient for simple scripts, it is **not recommended** for high-performance applications. Explicitly controlling your transaction boundaries with commit() gives you finer control and is essential for batch operations.

4.2 The Key to Performance: Batch Inserts with multiPut()

Putting one row at a time is inefficient for high-frequency data streams. Each put and commit cycle involves network communication and transaction overhead. To achieve high ingestion rates, you must batch your writes.

The multiPut() method allows you to insert a list of rows into a container in a single, highly optimized operation. This dramatically reduces network overhead and is the standard practice for production data collectors.

```java
// Assume 'container' is the same TimeSeries object from the previous example

// Create a list to hold multiple readings
List<SensorReading> readings = new ArrayList<>();

// Populate the list with new data points
// In a real application, this data would come from a message queue or sensor stream
for (int i = 0; i < 1000; i++) {
    SensorReading reading = new SensorReading();
    reading.timestamp = new Date(System.currentTimeMillis() + i); // Ensure unique
timestamps
    reading.value = 20.0 + (Math.random() * 5); // Random value between 20 and 25
    reading.status = "NORMAL";
    readings.add(reading);
}

// Put all 1,000 rows in a single network call
container.multiPut(readings);

// Commit the transaction for the entire batch
container.commit();
```

4.3 Alternative Methods for Writing Data

While the Java client is ideal for high-performance services, GridDB provides several other interfaces to suit different languages, tools, and development styles.

Writing with Python

The Python client is excellent for data science, machine learning tasks, and rapid prototyping. The syntax is clean and idiomatic.

Python

```python
import griddb_python as griddb
from datetime import datetime

# Connect to the store
factory = griddb.StoreFactory.get_instance()
gridstore = factory.get_store(
    notification_member="127.0.0.1:10001",
    cluster_name="myCluster",
    username="admin",
    password="admin"
)

# Define the container schema and create the container
con_info = griddb.ContainerInfo(
    name="temp_F01_A34C",
    column_info_list=[
        ["timestamp", griddb.Type.TIMESTAMP],
        ["value", griddb.Type.DOUBLE],
        ["status", griddb.Type.STRING]
    ],
    type=griddb.ContainerType.TIME_SERIES,
    row_key=True
)
container = gridstore.put_container(con_info)

# Put a single row. The list must match the column order.
# The Python client is auto-committed by default.
container.put([datetime.now(), 20.8, "NORMAL"])
```

Writing with the JDBC Driver

If your organization has existing infrastructure or expertise built around SQL, you can use GridDB's JDBC driver. This allows you to interact with GridDB using standard SQL INSERT statements.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;

// Set up the JDBC connection URL
String jdbcUrl = "jdbc:gs://127.0.0.1:20001/myCluster/public";
Properties props = new Properties();
props.setProperty("user", "admin");
props.setProperty("password", "admin");

// Establish the connection
Connection con = DriverManager.getConnection(jdbcUrl, props);
Statement stmt = con.createStatement();

// Create a table if it doesn't already exist
stmt.executeUpdate("CREATE TABLE IF NOT EXISTS SensorLog (timestamp TIMESTAMP PRIMARY
KEY, value DOUBLE, status STRING)");

// Insert a row using a standard SQL statement
stmt.executeUpdate("INSERT INTO SensorLog VALUES (NOW(), 22.1, 'NORMAL')");

con.close();
```

Writing with the Web API

For ultimate flexibility, GridDB provides a RESTful Web API. This allows any language or tool capable of making HTTP requests to write data, making it a perfect fit for microservices written in languages like Go, Rust, or Node.js.

Here is an example using curl to insert a row into our temp_F01_A34C container.

Shell

```
# The --data payload is a JSON array of rows.

# Each inner array represents a row, with values in the correct column order.
curl --location --request PUT \
  'http://127.0.0.1:8080/griddb/v2/myCluster/dbs/public/containers/temp_F01_A34C/rows'
\
  --header 'Content-Type: application/json' \
  --header 'Authorization: Basic YWRtaW46YWRtaW4=' \
  --data '[ ["2025-08-12T15:16:00.000Z", 23.3, "NORMAL"] ]'
```

4.4 Summary

GridDB offers multiple pathways for writing data, allowing you to choose the tool that best fits the task. While each method has its place, the core principles of high-speed ingestion remain the same: batch your writes and use the right tool for the job.

Method	Technology	Primary Use Case
Native API	Java / C	High-performance ingestion services, maximum control.
Python API	Python	Data science, scripting, and rapid application development.
JDBC Driver	SQL	Integration with existing BI tools and SQL-based applications.
Web API	REST/HTTP	Language-agnostic access, microservices architecture.

Now that you know how to get data *into* GridDB, the next step is to get it *out*. In the next chapter, we will explore the powerful query capabilities of GridDB, from simple lookups to complex aggregations and time-series analysis.

Chapter 5. Querying Data from GridDB

Now that you have data flowing into your GridDB cluster, the next step is to retrieve it. Whether you're powering a real-time dashboard, running an analytical job, or triggering an alert, efficient querying is the key to unlocking the value of your IoT data. GridDB provides a flexible query model that combines the power of a SQL-like language with the performance of a native NoSQL API.

In this chapter, you will learn the primary methods for querying data. We'll start with GridDB's native TQL using the Java client, then explore how to perform the same operations using Python, the JDBC driver, and the Web API.

5.1 The Primary Method: Querying with SQL

For most applications, the most direct and standard-compliant way to query GridDB is by using its **JDBC Driver**. This approach allows any Java application to connect and run standard SQL, just as it would with any other SQL database like PostgreSQL or MySQL.

This requires the GridDB JDBC driver (`gridstore-jdbc.jar`) to be in your application's classpath and uses the standard `java.sql.*` package.

Fetching a Set of Rows with JDBC

Let's start with our common task: retrieving all sensor readings from the last hour that exceed a certain threshold. With JDBC, you use a `Connection` and `Statement` to execute a full SQL query, then iterate over the `ResultSet`.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

// (Inside your application method)
String containerName = "temp_F01_A34C";
String jdbcUrl =
"jdbc:gs://127.0.0.1:20001/myCluster/public?user=admin&password=admin";
```

```
// Create a standard SQL query.
String sql = String.format(
    "SELECT * FROM %s WHERE status = 'CRITICAL' AND timestamp > NOW() - 1 HOUR",
    containerName
);

// Use a try-with-resources block to manage the connection
try (Connection conn = DriverManager.getConnection(jdbcUrl);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {

    // Iterate through the results just like any other database
    while (rs.next()) {
        // Manually map columns from the ResultSet
        java.sql.Timestamp ts = rs.getTimestamp("timestamp");
        double val = rs.getDouble("value");
        String status = rs.getString("status");

        // Process the data (e.g., send an alert)
        System.out.println(
            "Alert! High reading at " + ts + ": " + val
        );
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

This pattern is the foundation of all standard database operations. The SQL query is executed on the server, and only the matching rows are returned to the client.

Server-Side Aggregations

As we discussed in Chapter 3, performing aggregations in the database is critical for performance. With JDBC, this is simply another SQL query.

Instead of pulling thousands of raw data points to your application to calculate an average, you ask GridDB to do it for you.

```java
// Assume 'jdbcUrl' and 'containerName' are defined
// Create an aggregation query using a SQL alias 'avg_val' for clarity
String sql = String.format(
"SELECT AVG(value) AS avg_val FROM %s WHERE timestamp > NOW() - 1 DAY",
 containerName
);
try (Connection conn = DriverManager.getConnection(jdbcUrl);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql)) {

     // An aggregation query almost always returns exactly one row
     if (rs.next()) {
        // Fetch the result by its alias or column index (1)
        double averageValue = rs.getDouble("avg_val");
        System.out.println( "Average value over the last 24 hours: " + averageValue );
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
```

This approach dramatically reduces network traffic and client-side processing, making it ideal for powering dashboards and analytical reports.

5.2 Querying from Python: The Data Scientist's Toolkit

The GridDB Python client provides the same powerful TQL capabilities but with an interface tailored for data analysis and scripting.

Basic Queries in Python

The query process is very similar to Java: you get a container, run a query, and iterate through the results.

```python
import griddb_python as griddb

# ... (connect to gridstore as shown in Chapter 4) ...

try:
    # Get the container
    container = gridstore.get_container("temp_F01_A34C")
    if container is None:
        print("Container not found")
        exit()

    # Run a TQL query
    query = container.query("SELECT * WHERE value > 25.0 ORDER BY timestamp DESC LIMIT
10")
    rs = query.fetch()

    # Iterate through the results
    while rs.has_next():
        # The result 'row' is a list matching the column order
        row = rs.next()
        print(f"Timestamp: {row[0]}, Value: {row[1]}, Status: {row[2]}")

except griddb.GSException as e:
    print(f"An error occurred: {e}")
```

The Power of Pandas DataFrames

One of the standout features of the Python client is its seamless integration with the **Pandas** library, the de facto standard for data manipulation in Python. With a single command, you can fetch the entire result set of a query directly into a Pandas DataFrame.

```python
# Assume 'container' and 'query' are the same as the previous example

df = rs.fetch_rows() #fetch_rows fetches a dataframe

# Now you can use all the power of Pandas
print("Query results as a DataFrame:")
print(df.head())
```

```
print("\nBasic statistics:")
print(df['value'].describe())
```

This feature makes GridDB an excellent backend for data science, machine learning, and visualization workloads.

Python And Apache Arrow

As of version 5.8.0, the Python client now utilizes the native Java API, enabling the retrieval of data rows as Apache Arrow objects. Apache Arrow is highly beneficial due to its support for "zero-copy reads and fast data access and interchange without serialization overhead between these languages and systems" (https://en.wikipedia.org/wiki/Apache_Arrow). This capability allows for seamless and direct data transfer between languages like Python and Node.js, for instance, without any performance drawbacks.

Here's an example:

Python
```
import pyarrow as pa
import pandas as pd

ra = griddb.RootAllocator(sys.maxsize)
df = pd.read_csv("data.csv")
# record batch returns an Apache Arrow data obj
rb = pa.record_batch(df)
col = gridstore.get_container("device")
col.multi_put(rb, ra)

q = col.query("select *")
q.set_fetch_options(root_allocator=ra)
rs = q.fetch()
rb = rs.next_record_batch()
# can convert directly to dataframe
df2 = rb.to_pandas()
```

5.3 Alternative Query Interfaces

For different architectures and use cases, GridDB provides other ways to access your data.

Using SQL with the JDBC Driver

If you're integrating with existing business intelligence (BI) tools or have an application already built on SQL, the JDBC driver is a perfect choice. It allows you to run standard SQL SELECT statements against your containers (which appear as tables).

```java
// Assume 'con' is a java.sql.Connection object from Chapter 4

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
    "SELECT * FROM SensorLog WHERE value > 22.0 AND status = 'NORMAL'"
);

while (rs.next()) {
    Timestamp ts = rs.getTimestamp("timestamp");
    double val = rs.getDouble("value");
    String status = rs.getString("status");
    System.out.println("time=" + ts + ", value=" + val + ", status=" + status);
}
rs.close();
stmt.close();
```

Using the Web API

The RESTful Web API provides a language-agnostic way to query data, suitable for microservices or web frontends. You simply send your TQL query as part of a JSON payload.
Execute a TQL query via a curl POST request

```
Java
curl --location --request POST \
  'http://127.0.0.1:8080/griddb/v2/myCluster/dbs/public/tql' \
  --header 'Content-Type: application/json' \
  --header 'Authorization: Basic YWRtaW46YWRtaW4=' \
  --data '[{
    "container": "temp_F01_A34C",
    "stmt": "SELECT * WHERE value > 25.0 LIMIT 10"
  }]'
```

The server will respond with a JSON object containing the query results.

5.4 Summary

GridDB offers a rich set of query capabilities designed to fit multiple architectural patterns. Whether you need the raw performance of the native Java client, the analytical power of the Python integration, or the compatibility of SQL, you have the right tool for the job.

Query Method	Language/Tech	Strengths
TQL API	Java / Python	High performance, schema-aware, full access to time-series functions.
JDBC Driver	Java / SQL	Familiar SQL syntax, compatibility with existing BI and reporting tools.
Pandas Integration	Python	Ideal for data science, machine learning, and advanced analytics.
Web API	REST/HTTP	Language-agnostic, perfect for microservices and web applications.

Now that you can write and read data, you're equipped to build the core of a complete IoT application. In the next chapter, we'll look at a key architectural pattern: integrating GridDB with a streaming platform like Apache Kafka to build a robust, real-time data pipeline.

Chapter 6. Performance Tuning and Optimization Strategies

As IoT systems scale, the sheer volume and velocity of data can push any database to its limits. While GridDB is architected for high performance, achieving maximum throughput requires developers to adopt best practices that align with the database's design principles. The most significant factor in the performance of a distributed database is not always CPU or memory, but rather the time spent communicating over the network.

This chapter focuses on fundamental tuning strategies for GridDB, centering on the single most important principle for optimizing performance: minimizing network latency. We will explore how batching operations for both writing and reading data can dramatically increase the efficiency and speed of your application.

6.1 The Primary Bottleneck: Network Latency

In a distributed system, your application (the client) communicates with the GridDB server over a network. Every command you send—whether it's to insert a single row or query a piece of data—incurs a small delay. This delay, known as network latency, is the time it takes for your request to travel to the server and for the response to travel back.

While the latency for a single operation might be milliseconds, in an IoT application processing thousands of data points per second, these milliseconds add up catastrophically. The primary goal of performance tuning in this context is to reduce the total number of round trips between the client and the server.

6.2 Optimizing Write Operations: The Power of multi_put()

A common but inefficient pattern for writing data is to insert records one at a time within a loop.

The Inefficient Approach: Row-by-Row put()

Consider an application that needs to insert 1,000 new sensor readings. A naive approach would be to loop through the readings and call the put() function for each one.

Python

```
# ANTI-PATTERN: Do not do this in performance-critical code!
for i in range(1000):
    row = [i, "sensor_reading", 35.5 + i]
    container.put(row) # A separate network call is made for each row
```

While functionally correct, this code is extremely inefficient. It makes 1,000 separate network round trips. The majority of the execution time is spent waiting for the network, not on the database's processing. It's analogous to going to the supermarket 1,000 times to buy 1,000 individual items.

The Optimized Approach: Batching with multi_put()

The correct and highly optimized method is to batch these insertions into a single operation using the multi_put() function. This involves preparing a list of all the rows you want to insert and then sending them to GridDB in one command.

```python
# BEST PRACTICE: Batch writes using multi_put()
row_list = []
for i in range(1000):
    row = [i, "sensor_reading", 35.5 + i]
    row_list.append(row)

container.multi_put(row_list) # All 1000 rows are sent in a single network call
```

This approach reduces 1,000 network round trips to just one. The performance improvement is not incremental; it's often a difference of one or more orders of magnitude. The database can ingest the bulk data far more efficiently, and the time spent on network latency becomes negligible.

6.3 Optimizing Read Operations: Querying in Bulk

The same principle of batching applies to reading data. Attempting to fetch multiple specific records by querying them one by one in a loop is a significant performance anti-pattern.

The Inefficient Approach: Single-Key Queries in a Loop

Imagine you need to retrieve the records for 100 specific device IDs. Looping and executing a query for each ID will once again result in excessive network calls.

```python
# ANTI-PATTERN: Do not query one key at a time in a loop!
results = []
for i in range(100):
    query = container.query(f"SELECT * WHERE device_id = {i}")
    rows = query.fetch()
    results.extend(rows) # A separate network call for each device ID
```

The Optimized Approach: MULTI_GET and IN Clauses

To optimize this, you should retrieve all the desired records in a single database query. GridDB provides two effective ways to do this:

1. Using MULTI_GET: This function is specifically designed to fetch multiple rows based on a list of row keys, making it ideal for targeted lookups.
2. Using a SQL IN Clause: For more complex queries, you can construct a single SQL query that uses the IN operator to specify all the keys or values you want to retrieve.

```python
# BEST PRACTICE: Use a single query to fetch multiple records
query_string = "SELECT * WHERE device_id IN (0, 1, 2, ..., 99)"
query = container.query(query_string) # Only one network call is made
rows = query.fetch()
```

This method consolidates 100 potential network calls into a single, efficient operation, allowing the database to perform a coordinated lookup and return all the results in one response.

In summary, the core principle of GridDB performance tuning is simple yet profound: always prefer batch operations over iterative, single-row operations. By batching your writes with multi_put() and your reads with MULTI_GET or IN clauses, you directly address the primary bottleneck of network latency, ensuring your application remains fast, efficient, and scalable.

Chapter 7. Real-Time Streaming with Apache Kafka

In a modern IoT architecture, the database rarely lives in isolation. It's part of a larger ecosystem of services that produce, transport, and consume data. For handling the massive, continuous flow of data from devices to your database, the industry standard is a **streaming platform**, and the undisputed leader in this space is Apache Kafka.

Think of Kafka as the central nervous system for your data. It's a highly scalable, durable, and fast message bus designed to handle real-time data feeds. By placing Kafka between your IoT devices and GridDB, you create a robust, decoupled architecture that can handle unpredictable data bursts and scale to millions of devices. In this chapter, you'll learn how to integrate GridDB into a Kafka-based data pipeline using the GridDB Kafka Connectors.

7.1 The Role of Kafka in an IoT Architecture

At its core, Apache Kafka allows different systems to communicate without being directly connected to each other.

- **Producers** (like your IoT data collection service) publish streams of data.
- This data is organized into **Topics** (e.g., temperature_readings, gps_locations).
- **Consumers** (like the GridDB Kafka Connector) subscribe to these topics to receive the data in real time.

This decoupling is the key benefit. If GridDB needs to go down for maintenance, or if there's a sudden spike in traffic from your devices, Kafka acts as a massive, persistent buffer. It holds the data safely until GridDB is ready to consume it, ensuring no data is lost.

7.2 The GridDB Kafka Connect Framework

To make integration seamless, GridDB provides connectors for **Kafka Connect**, a framework for building and running reusable connectors that move data between Kafka and other systems. You don't need to write custom consumer or producer code; you simply configure the pre-built GridDB connectors.

There are two primary connectors:

- **The Sink Connector:** This connector "sinks" data from Kafka into GridDB. It subscribes to one or more Kafka topics and writes the messages it receives into GridDB containers. This is the primary method for data ingestion.

- **The Source Connector:** This connector "sources" data from GridDB into Kafka. It polls containers for new or updated rows and publishes them to a Kafka topic. This is ideal for feeding data from GridDB to other downstream systems, like a real-time analytics engine or an alerting service.

7.3 Ingesting Data: The GridDB Sink Connector

Let's walk through the most common use case: streaming sensor data from a Kafka topic into a GridDB container.

Setup and Configuration

First, you need a running Kafka cluster and the GridDB Kafka Connector JAR file placed in the appropriate Kafka plugin directory. The connector is configured with a simple properties file. Here's an example configuration to stream data from a topic named iot_telemetry into GridDB.

```shell
# A unique name for this connector instance
name=griddb-iot-sink

# The connector class for the GridDB Sink
connector.class=com.github.griddb.kafka.connect.GriddbSinkConnector

# The Kafka topic(s) to consume messages from
topics=iot_telemetry

# --- GridDB Connection Properties ---
cluster.name=myCluster
notification.member=127.0.0.1:10001
user=admin
password=admin

# --- Data Transformation ---
# Use Kafka Connect's built-in transform to convert a string field to a timestamp
transforms=TimestampConverter
transforms.TimestampConverter.type=org.apache.kafka.connect.transforms.TimestampConver
ter$Value
transforms.TimestampConverter.field=ts
transforms.TimestampConverter.format=yyyy-MM-dd'T'HH:mm:ss.SSS'Z'
transforms.TimestampConverter.target.type=Timestamp
```

Once you load this configuration into your Kafka Connect worker, it will immediately start listening for messages on the iot_telemetry topic.

Sending Data

Now, any producer can send data to the iot_telemetry topic. For example, using Kafka's command-line producer, you could send a simple JSON payload representing a sensor reading:

```
# Start the Kafka console producer
kafka-console-producer.sh --topic iot_telemetry --bootstrap-server 127.0.0.1:9092

# Paste the following JSON message and press Enter
> {"ts": "2025-08-12T15:30:00.123Z", "sensorId": "temp_F01_A34C", "value": 22.7, "status": "NORMAL"}
```

The Sink Connector receives this message. It uses the topic name (iot_telemetry) or a field within the message (e.g., sensorId) to determine the target container name in GridDB. It maps the JSON fields to the columns in the container and performs the insert. The TimestampConverter transform we configured automatically converts the ts string field into a proper TIMESTAMP type for the row key.

7.4 Exporting Data: The GridDB Source Connector

To move data in the other direction—from GridDB to Kafka—you use the Source Connector. This is useful for feeding a stream of database changes to other applications.

The configuration is similarly straightforward. This example polls the alert_history container every 10 seconds for new rows.

Shell

```
# A unique name for this connector instance
name=griddb-alert-source

# The connector class for the GridDB Source
connector.class=com.github.griddb.kafka.connect.GriddbSourceConnector
```

```
# The container to read data from
container.name=alert_history
# The Kafka topic to publish data to
topic.name=alerts

# --- GridDB Connection Properties ---
cluster.name=myCluster
notification.member=127.0.0.1:10001
user=admin
password=admin

# --- Polling Configuration ---
# Check for new rows every 10 seconds
poll.interval.ms=10000
# Use the timestamp column to find new rows
mode=timestamp
timestamp.column.name=timestamp
```

With this connector running, whenever a new row is added to the alert_history container in GridDB, the Source Connector will detect it on its next poll, convert it to a JSON message, and publish it to the alerts topic in Kafka. From there, another service (like a Slack notifier or an email gateway) can consume the message and take action.

7.5 Summary

Integrating GridDB with Apache Kafka creates a powerful, scalable, and resilient architecture for modern IoT applications. Kafka acts as the data backbone, decoupling your devices from your database and enabling robust, event-driven workflows.

- Use the **Sink Connector** to stream data *from* Kafka *into* GridDB for durable storage.
- Use the **Source Connector** to stream data *from* GridDB *into* Kafka to feed downstream analytics, alerting, or machine learning systems.

By leveraging the Kafka Connect framework, you can build these complex data pipelines with simple configuration, allowing you to focus on your application logic instead of the plumbing.
In the next chapter, we'll explore another key integration point in the IoT ecosystem: connecting GridDB with cloud platforms like Microsoft Azure to build globally distributed applications.

Chapter 8. Building a Secure API with Flask

Your IoT data is valuable. Whether it's powering an internal dashboard, a mobile application for customers, or a machine learning pipeline, you need a secure and reliable way to expose it. The most common way to do this is by building a **RESTful API**, and Python's lightweight web framework, **Flask**, is an excellent tool for the job.

However, simply exposing data isn't enough; you must protect it. In this chapter, we'll go beyond basic data retrieval and build a complete, production-ready API service. You will learn how to use GridDB not just as a time-series database but also as the backbone for your application's security. We will build a full authentication system using the industry-standard **OAuth2** protocol, storing user credentials and access tokens directly in GridDB containers.

8.1 The Architecture: GridDB as a Unified Data and Identity Store

By combining Flask with GridDB, we can create a powerful and elegant architecture.

- **Flask** provides the web server and routing, handling incoming HTTP requests and sending back JSON responses.
- **GridDB** serves two critical roles:
    1. It's the **data store** for our IoT time-series data (e.g., temp_F01_A34C).
    2. It's the **identity store** for our API, holding user credentials and security tokens.

This unified approach simplifies your architecture. You don't need a separate database for your application's users; GridDB can manage it all.

8.2 Project Setup and Dependencies

To build our secure API, we'll need a few key Python libraries:

- **Flask**: The core web framework.
- **griddb_python**: The GridDB client library.
- **Authlib**: A powerful library for implementing OAuth2 clients and providers.
- **bcrypt**: A library for securely hashing passwords.

You can install them all with pip:

```
pip install Flask Authlib bcrypt griddb_python
```

8.3 Schema Design for Authentication

Before we write any API code, we need to design the containers to hold our security data. We'll create two COLLECTION containers for this purpose.

First, a container to store user information. It's critical that we **never store passwords in plain text**. We will store a securely hashed version of the password.

Python

```
# Schema for the "users" container
users_container_info = griddb.ContainerInfo(
    name="users",
    column_info_list=[
        ["username", griddb.Type.STRING], # RowKey
        ["password_hash", griddb.Type.BLOB] # Store the bcrypt hash as bytes
    ],
    type=griddb.ContainerType.COLLECTION,
    row_key=True
)
gridstore.put_container(users_container_info)


Second, a container to store the OAuth2 access tokens that we issue to users after
they log in.
# Schema for the "tokens" container
tokens_container_info = griddb.ContainerInfo(
    name="tokens",
    column_info_list=[
        ["access_token", griddb.Type.STRING], # RowKey
        ["username", griddb.Type.STRING],
        ["expires_at", griddb.Type.TIMESTAMP],
        ["issued_at", griddb.Type.TIMESTAMP]
    ],
    type=griddb.ContainerType.COLLECTION,
    row_key=True
)
gridstore.put_container(tokens_container_info)
```

8.4 Implementing the Authentication Flow

Our API will have two main authentication endpoints: one for creating a new user and one for logging in and issuing a token.

User Registration

This endpoint will accept a new username and password, hash the password using bcrypt, and store the new user in our users container.

```python
import bcrypt
from flask import request, jsonify

@app.route('/register', methods=['POST'])
def register():
    username = request.form.get('username')
    password = request.form.get('password')
    if not username or not password:
        return jsonify({"error": "Username and password are required"}), 400

    # Hash the password for secure storage
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

    try:
        users_container = gridstore.get_container("users")
        # The BLOB type expects a bytearray
        users_container.put([username, bytearray(hashed_password)])
        return jsonify({"message": "User created successfully"}), 201
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Token Issuance (Login)

This is the core of the OAuth2 flow. When a user provides their correct username and password, our server will generate a temporary, secure **Bearer Token** and return it to them. The user will then include this token in the Authorization header of all future requests to prove their identity. The Authlib library handles the complexity of generating these tokens.
We configure Authlib to use our GridDB containers to validate user credentials and save the issued tokens.

Note: The Role of Authlib
Implementing the full OAuth2 specification is complex and error-prone. Libraries like Authlib are essential because they handle the protocol's intricacies, such as token generation, expiry, and validation, allowing you to focus on your application logic. We simply need to provide the functions that let Authlib talk to our GridDB backend.

8.5 Protecting API Endpoints

With the authentication flow in place, we can now protect our data endpoints. We create a require_oauth decorator from Authlib that will automatically validate the Bearer Token on incoming requests.

To do this, we provide Authlib with a custom token validator that knows how to look up tokens in our GridDB tokens container.

```
Python
from authlib.integrations.flask_oauth2 import ResourceProtector
from authlib.oauth2.rfc6750 import BearerTokenValidator

class GridDBTokenValidator(BearerTokenValidator):
    def authenticate_token(self, token_string):
        # Look for the token in our GridDB container
        tokens_container = gridstore.get_container("tokens")
        token_row = tokens_container.get(token_string)
        if token_row:
            # Here you would check if the token is expired
            return token_row # Authlib uses this object to validate the request
        return None

require_oauth = ResourceProtector()
require_oauth.register_token_validator(GridDBTokenValidator())


Now, protecting an endpoint is as simple as adding a decorator:
@app.route('/api/sensor_data/<string:sensor_id>')
@require_oauth() # This line locks down the endpoint
def get_sensor_data(sensor_id):
    try:
        container = gridstore.get_container(sensor_id)
        if not container:
            return jsonify({"error": "Sensor not found"}), 404
```

```
        # Query the last 10 data points
        query = container.query("SELECT * ORDER BY timestamp DESC LIMIT 10")
        rs = query.fetch()

        results = []
        while rs.has_next():
            row = rs.next()
            results.append({
                "timestamp": row[0].isoformat(),
                "value": row[1],
                "status": row[2]
            })

        return jsonify(results)
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

If a user tries to access this endpoint without a valid token, they will receive a 401 Unauthorized error.

8.6 Summary

You have now built a complete, secure REST API using Flask and GridDB. This architecture provides a robust foundation for any application that needs to serve IoT data. By using GridDB as both the data and identity store, you've created a clean, unified system.

The key principles are:

- **Never store plain-text passwords.** Always use a strong, one-way hashing algorithm like bcrypt.
- **Use an industry-standard protocol like OAuth2** for authentication. Don't invent your own.
- **Leverage libraries like Authlib** to handle the complexities of security protocols.
- **Protect every data endpoint** that exposes sensitive information.

With these patterns, you are well-equipped to build secure, scalable, and production-ready IoT applications.

Chapter 9. Introduction to GridDB Cloud

In the previous chapters, we've explored the architecture and features of GridDB, focusing on deployments you would manage yourself on-premises or on your own cloud infrastructure. This approach offers maximum control, but it also comes with the significant responsibility of provisioning hardware, managing operating systems, patching software, and planning for scale. For many modern IoT applications, a more agile and operationally efficient model is needed.

This chapter introduces GridDB Cloud, the fully managed database-as-a-service (DBaaS) offering. We will explore what GridDB Cloud is, why a managed cloud database is often the superior choice for IoT projects, and how to perform fundamental interactions with it using its universal Web API. This will lay the groundwork for building sophisticated, cloud-native data pipelines in the chapters to come.

You can sign up for GridDB Cloud for free here:
https://form.ict-toshiba.jp/download_form_griddb_cloud_freeplan_e

It is also available on Azure's Marketplace:
https://marketplace.microsoft.com/en-us/marketplace/apps?search=griddb&page=1&referer=https%3A%2F%2Fmarketplace.microsoft.com%2Fen-us%2Fmarketplace%2Fapps%3Fsearch%3Dgriddb%26page%3D1&correlationId=f6801d7b-bfac-40a9-9041-60a49f002339

9.1 What is GridDB Cloud?

GridDB Cloud is a high-performance, scalable NoSQL database delivered as a fully managed service. It is built upon the same powerful, in-memory architecture and time-series optimizations as the GridDB platform you are already familiar with. The key difference is the service model: instead of you managing the database, the GridDB team handles all the underlying operational work.

This includes:

- **Infrastructure Provisioning:** No need to select and configure virtual machines or servers.
- **Installation and Configuration:** The database cluster is deployed and configured for you based on best practices.
- **Maintenance and Patching:** All software updates and security patches are applied automatically.
- **Backups and Recovery:** Automated backup schedules and disaster recovery plans are built-in.
- **Scaling and High Availability:** The service can scale resources up or down to meet demand and is architected for high availability to ensure your application remains online.

By abstracting away this operational complexity, GridDB Cloud allows your team to focus exclusively on what matters most: building your application and deriving value from your IoT data.

9.2 Why Use the Cloud? On-Premises vs. Managed Service

Choosing between a self-hosted (on-premises) database and a managed service like GridDB Cloud involves a trade-off between control and convenience.

- **On-Premises Deployment:** This model gives you complete control over every aspect of your environment, from the hardware specifications to the network topology. It can be advantageous for organizations with strict data residency requirements or those with existing data centers and a dedicated operations team. However, it carries a high operational cost in terms of staffing, maintenance, and the "heavy lifting" required to ensure reliability and scale.
- **GridDB Cloud (DBaaS):** This model prioritizes operational efficiency and speed. Deploying a new cluster takes minutes, not days. Scaling to handle a surge in device connections is a simple configuration change, not a complex hardware procurement process. This agility is critical in the fast-moving world of IoT, where project requirements can evolve rapidly. For most new projects, the reduction in total cost of ownership (TCO) and the ability to focus on application logic make a managed service the clear winner.

9.3 Interacting with GridDB Cloud: The Web API

While GridDB supports various native clients and connectors, GridDB Cloud is designed around a powerful and secure RESTful Web API. This API provides a universal, language-agnostic interface for interacting with your database over HTTPS. Its platform independence makes it the perfect choice for the modern, distributed architectures we see in IoT, including serverless functions, web applications, and microservices written in different languages.

Interaction with the API is straightforward:

1. **Authentication:** You authenticate your requests using a secure API key, which is passed in the HTTP Authorization header.
2. **Endpoints:** The API provides logical endpoints for managing and accessing your data (e.g., `/containers/`, `/rows/`).
3. **HTTP Verbs:** You use standard HTTP verbs to perform actions: `PUT` to insert data, `GET` to query data, `POST` to create resources, and `DELETE` to remove them.

The following Node.js example uses `axios` to demonstrate a basic operation: inserting a new row of sensor data into a container named `device-001`.

JavaScript

```
const axios = require("axios");

// These credentials should be loaded securely from environment
variables or a secret manager
```

```
const GRIDDB_API_ENDPOINT =
"[https://your-cluster-endpoint.api.griddb.net](https://your-cluster-endpoint.api.griddb.net)";
const GRIDDB_BASIC_AUTHENTICATION = "base64encode-of-user-pass";
const containerName = "device-001";

// The row data we want to insert, matching the container schema
// Let's assume the schema is [timestamp, temperature, humidity]
const rowData = [
    "2023-10-27T10:00:00Z", // timestamp (as an ISO 8601 string)
    22.5,                   // temperature
    45.1                    // humidity
];

async function insertSensorData() {
    const url =
`${GRIDDB_API_ENDPOINT}/v2/myCluster/dbs/public/containers/${containerName}/rows`;

    try {
        const response = await axios.put(url,
            [ rowData ], // The payload must be an array of one or more rows
            {
                headers: {
                    'Authorization': `BASIC ${GRIDDB_BASIC_AUTHENTICATION}`,
                    'Content-Type': 'application/json'
                }
            }
        );
        console.log("Successfully inserted data:", response.status);

    } catch (error) {
        console.error("Failed to insert data:", error.response ? error.response.data : error.message);
    }
```

```
}

insertSensorData();
```

This simple pattern of constructing a request and sending it to a secure endpoint is the fundamental building block we will use for more advanced integrations.

9.4 A Look Ahead: The Broader GridDB Ecosystem

Interacting directly with the Web API is just the starting point. GridDB Cloud is designed to be the core of a larger data ecosystem. In the upcoming chapters, we will explore how to connect it with other best-in-class tools to build a complete end-to-end solution:

- **Data Visualization with Grafana:** We'll show you how to connect GridDB Cloud as a data source in Grafana to build beautiful, real-time dashboards that provide instant insight into your IoT device fleet.
- **Streaming Data Pipelines with Apache Kafka:** We will discuss architectures that use Apache Kafka to handle massive streams of incoming data, enabling you to perform real-time processing and analytics before the data is persisted to GridDB.
- **Simplified Management with a CLI:** To streamline administrative tasks, we will introduce a purpose-built Command Line Interface (CLI) tool that simplifies container creation, user management, and cluster monitoring directly from your terminal.

9.5 Summary

GridDB Cloud transforms a powerful database into a flexible, scalable, and easy-to-use service, allowing you to build robust IoT applications without the burden of infrastructure management. Its Web API provides a simple yet powerful integration point for any application, on any platform.

Now that we understand the fundamentals of GridDB Cloud and how to communicate with it, let's explore a powerful, real-world application: building a serverless data ingestion pipeline using Microsoft Azure.

Chapter 10. Serverless Integration with Microsoft Azure IoT Hub

So far, we've focused on architectures where you manage the servers running your database and applications. However, the cloud offers a powerful alternative: **serverless computing**. A serverless approach allows you to build and run applications without thinking about servers at all. You write the code, and the cloud platform handles the provisioning, scaling, and management of the underlying infrastructure.

For IoT, Microsoft's **Azure IoT Hub** is a premier managed service that provides the foundation for a serverless architecture. It handles device connectivity, security, and messaging at a massive scale. When you combine Azure IoT Hub with a high-performance database-as-a-service like **GridDB Cloud**, you can create an incredibly powerful, scalable, and cost-effective data ingestion pipeline. In this chapter, you'll learn how to build this event-driven, serverless bridge between the cloud and your database.

10.1 The Serverless IoT Architecture

In this model, we move away from running a dedicated data ingestion service and instead use a chain of managed cloud services that are triggered by events. The flow of data is elegant and efficient:

1. **Device to IoT Hub:** An IoT device, securely provisioned, sends a telemetry message (e.g., a JSON payload) to its dedicated endpoint in Azure IoT Hub.
2. **IoT Hub to Event Grid:** IoT Hub natively integrates with **Azure Event Grid**, a service that routes events from any source to any destination. When a new message arrives at IoT Hub, it publishes an event to Event Grid.
3. **Event Grid to Azure Function:** We configure an **Azure Function**—a small, independent piece of code—to subscribe to these events from Event Grid. When a new event arrives, Event Grid automatically triggers our function, passing along the event data.
4. **Azure Function to GridDB Cloud:** The Azure Function contains the logic to parse the device message from the event payload and write it to the appropriate container in GridDB Cloud, typically via a secure REST API call.

This entire pipeline operates without a single virtual machine that you need to manage, patch, or scale. If you suddenly have one million devices sending data instead of one thousand, the Azure services and GridDB Cloud scale automatically to handle the load.

10.2 Device Provisioning and Security

Before a device can send data, it must be registered with IoT Hub to establish a secure identity. While IoT Hub supports several authentication methods, a common pattern for large fleets is to use symmetric keys derived from a group enrollment key. This allows you to provision devices programmatically without having to manage individual credentials for each one.
The following Python code shows how a device could derive its unique key and register itself with the Azure IoT Device Provisioning Service (DPS), which then registers it with IoT Hub.

```python
import base64
import hmac
import hashlib
from azure.iot.device.aio import ProvisioningDeviceClient

# These values would be configured on your device
PROVISIONING_HOST = "your-provisioning-host.azure-devices-provisioning.net"
ID_SCOPE = "your-id-scope"
GROUP_SYMMETRIC_KEY = "your-group-primary-key" # Never hardcode in production!
REGISTRATION_ID = "device-001" # The unique ID for this device

def derive_device_key(registration_id, group_key):
    """Derives a unique device key from the group key."""
    message = registration_id.encode("utf-8")
    signing_key = base64.b64decode(group_key.encode("utf-8"))
    signed_hmac = hmac.HMAC(signing_key, message, hashlib.sha256)
    return base64.b64encode(signed_hmac.digest())

# On the device, derive the key and register
async def register_device():
    device_key = derive_device_key(REGISTRATION_ID, GROUP_SYMMETRIC_KEY)

    provisioning_client = ProvisioningDeviceClient.create_from_symmetric_key(
        provisioning_host=PROVISIONING_HOST,
        registration_id=REGISTRATION_ID,
        id_scope=ID_SCOPE,
        symmetric_key=device_key.decode("utf-8"),
    )

    return await provisioning_client.register()
```

10.3 The Ingestion Engine: An Azure Function

The core of our serverless pipeline is the Azure Function. This function acts as the lightweight data processor. Its sole job is to receive an event, parse it, and forward the data to GridDB Cloud.
Here is an example of an Azure Function written in Node.js. It's configured to be triggered by Azure Event Grid.

JSON

```
// function.json (configuration file)
{
  "bindings": [
    {
      "type": "eventGridTrigger",
      "name": "eventGridEvent",
      "direction": "in"
    }
  ]
}
```

JavaScript

```
// index.js (the function code)
const axios = require("axios");

// Securely load these from environment variables
const GRIDDB_API_ENDPOINT = process.env.GRIDDB_API_ENDPOINT;
const GRIDDB_API_KEY = process.env.GRIDDB_API_KEY;

module.exports = async function (context, eventGridEvent) {
    context.log("Event Grid trigger function processed an event.");
    // The actual device message is Base64 encoded in the event body
    const messageBody = Buffer.from(eventGridEvent.data.body, 'base64').toString();
    const telemetry = JSON.parse(messageBody);

    // The device ID is part of the event's subject
    const deviceId = eventGridEvent.subject.split('/')[1];

    // The target container name could be the device ID or another field
```

```
    const containerName = deviceId;

    // Construct the URL for the GridDB Cloud REST API
    const url =
`${GRIDDB_API_ENDPOINT}/v2/myCluster/dbs/public/containers/${containerName}/rows`;

    try {
        // Use axios to make a PUT request to insert the row
        await axios.put(url, [
            // The payload must be an array of rows
            [telemetry.timestamp, telemetry.value, telemetry.status]
        ], {
            headers: {
                'Authorization': `Bearer ${GRIDDB_API_KEY}`, // Use a secure auth token
                'Content-Type': 'application/json'
            }
        });

        context.log(`Successfully wrote data for device: ${deviceId}`);

    } catch (error) {
        context.log.error(`Failed to write to GridDB: ${error.message}`);
        // Implement retry logic or send to a dead-letter queue
        throw error;
    }
};
```

10.4 Best Practices for Serverless Integration

- **Secure Your Endpoint:** Never expose your GridDB Cloud API without authentication. Use a robust method like OAuth 2.0 or an API key system, and store credentials securely in Azure Key Vault, not in your function's code.
- **Filter Upstream:** If devices send telemetry you don't need to store, filter it out in the Azure Function. This saves on database write costs and keeps your dataset clean.
- **Implement Retries and Dead-Letter Queues:** Network calls can fail. Your Azure Function should have a retry policy (e.g., exponential backoff) for transient errors when calling the GridDB API. For persistent failures, configure a dead-letter queue to store the failed messages for later inspection.

10.5 Summary

By combining Azure IoT Hub's managed device connectivity with the event-driven power of Azure Functions and the performance of GridDB Cloud, you can build a state-of-the-art IoT data platform. This serverless model provides immense scalability and operational efficiency, freeing you from managing infrastructure and allowing you to focus on building features for your application.

In the next chapter, we'll dive deeper into a topic we just touched on: securing your database endpoints. We'll explore authentication and authorization patterns to ensure your GridDB APIs are production-ready and protected from unauthorized access

Chapter 11. Cloud-Native Visualization: Pairing GridDB Cloud with Grafana Cloud

As organizations increasingly adopt cloud-native infrastructure, the need for seamless, scalable, and fully-managed data solutions has become paramount. While self-hosting a database and visualization platform offers granular control, it also introduces significant operational overhead. A cloud-native approach, leveraging managed services for both data storage and visualization, abstracts away the complexities of deployment, maintenance, and scaling, allowing engineers to focus on deriving value from their data. ☁️

This chapter details the modern, streamlined workflow for connecting **GridDB Cloud**, a fully-managed time-series Database-as-a-Service (DBaaS), with **Grafana Cloud**, the managed offering from the creators of Grafana. This pairing provides a powerful, enterprise-grade solution for IoT analytics with minimal setup and maintenance.

11.1 The Cloud Advantage: Simplicity and Scalability

The primary advantage of using GridDB Cloud and Grafana Cloud in tandem is the elimination of infrastructure management. Both platforms are designed to be provisioned and configured in minutes. Key benefits include:

- **Rapid Deployment:** Launch a new GridDB cluster and a Grafana instance without provisioning servers, configuring networks, or managing software updates.
- **Elastic Scalability:** Both services can scale on demand to handle fluctuating data loads, from small-scale prototypes to massive, production-level IoT deployments.
- **Enhanced Security:** Cloud providers handle security at the infrastructure level. Configuration is simplified to managing access controls, such as IP allowlists.
- **High Availability:** Managed services come with built-in redundancy and uptime guarantees, ensuring your data pipeline is resilient and reliable.

11.2 The Integration Architecture: Connecting via the Infinity Plugin

Direct communication between cloud services requires secure and standardized protocols. GridDB Cloud exposes its powerful query capabilities through a Web API, allowing for interaction via standard HTTP requests. To bridge this API with Grafana Cloud, we utilize a versatile third-party connector called the **Infinity plugin**.

The Infinity plugin acts as a universal data source, capable of retrieving data from any JSON-based REST API and rendering it within Grafana panels. This architecture is both simple and powerful:

Grafana Cloud <--> Infinity Plugin <--> GridDB Cloud Web API

The implementation workflow involves three main stages: configuration, data ingestion, and querying.

1. Initial Configuration and Security

Before the services can communicate, a security handshake is required. The first step is to configure GridDB Cloud's firewall to accept incoming connections from Grafana Cloud. This is accomplished by adding Grafana's published IP addresses to the **IP allowlist** in the GridDB Cloud portal. This critical step ensures that only authorized Grafana instances can query your database.

2. Setting Up the Infinity Datasource

Within Grafana Cloud, the Infinity plugin must be installed and configured as a new data source. The key configuration parameters are:

- **URL:** The hostname of your GridDB Cloud portal (eg. https://cloud5197.griddb.com).
- **Authentication:** Basic authentication credentials (username and password) for your GridDB Cloud instance.
- **Allowed Hosts:** The GridDB Cloud hostname must be added to the plugin's list of allowed hosts for security.

A successful setup can be verified using the "Save & Test" feature, which performs a health check to confirm connectivity.

You can also click on the health check tab and enter in the `/checkConnection` endpoint to ensure connection is possible.

3. Querying and Visualization

With the connection established, you can begin to visualize your time-series data. The Infinity plugin allows you to construct queries directly within Grafana's panel editor. Since you are communicating with a Web API, the query is formulated as an HTTP POST request.

The core components of the query include:

- **Type:** Set to JSON.
- **Parser:** Set to Backend.
- **Request Body (JSON):** This is where you define your query using either **GridDB's SQL syntax** or its **API-based query format**.

For example, to retrieve temperature and humidity readings from a sensor container, you could use a SQL query in the request body:

```JSON
{
  "queries": [
    {
      "query": "SELECT timestamp, temperature, humidity FROM sensors
LIMIT 1000"
    }
  ]
}
```

This request is sent from Grafana Cloud to the GridDB Cloud API, which executes the SQL query and returns the results as a JSON object. The Infinity plugin then parses this response and transforms it into a time-series format that Grafana can plot on a graph. To properly parse the HTTP Response, you must set the parsing option and result field as ‘results’ if conducting a SQL query. You can then also set the columns based on index based on your structure.

More advanced SQL features, such as GROUP BY RANGE for data aggregation, are also fully supported, enabling sophisticated on-the-fly analysis directly from your dashboard. This powerful cloud-native integration provides a robust foundation for building modern, scalable IoT applications. More can be learned about this process directly from the GridDB developers’ website: https://griddb.net/en/blog/pairing-griddb-cloud-with-grafana-cloud/

Chapter 12: Streamlining Cloud Management with the GridDB Cloud CLI

In the lifecycle of an IoT application, developers and administrators interact with the database in various ways. While a graphical web interface is invaluable for visual exploration and initial setup, the demands of modern software development—automation, scripting, and operational efficiency—often call for a more direct and programmatic method of control. For these scenarios, a Command-Line Interface (CLI) is the quintessential tool.

This chapter introduces the **GridDB Cloud CLI**, a powerful yet straightforward utility designed to simplify and accelerate interactions with your GridDB Cloud instance. We will explore its purpose, setup, and core functionalities, demonstrating how it serves as a vital bridge between your development environment and your cloud database, enabling automation and enhancing productivity.

12.1 The "Why" of a Command-Line Interface

The GridDB Cloud portal provides a comprehensive web-based user interface for managing databases, containers, and security settings. However, many essential developer and operations (DevOps) workflows are performed from the command line. A CLI provides several distinct advantages:

- **Speed and Efficiency:** For experienced users, executing a command is often much faster than navigating through multiple web pages and menus.
- **Scriptability:** CLI commands can be easily integrated into shell scripts (e.g., Bash) to automate repetitive tasks, such as nightly data exports, health checks, or routine maintenance.
- **Automation:** A CLI is the cornerstone of CI/CD (Continuous Integration/Continuous Deployment) pipelines. It allows automated systems to perform database operations, like provisioning a new container for a feature test or running a data migration as part of a new software release.
- **Composability:** Command-line tools are designed to work together. The output of the GridDB Cloud CLI can be "piped" into other standard utilities like jq for JSON parsing, grep for filtering, or awk for text processing, creating powerful one-line commands.

In essence, the GridDB Cloud CLI is a purpose-built wrapper around the GridDB Cloud Web API. It abstracts away the complexities of handling HTTP requests, managing authentication tokens, and formatting API calls, presenting a clean and intuitive set of commands for direct database interaction.

12.2 Installation and Initial Configuration

The CLI is distributed as a single, self-contained binary, making installation a trivial process. Once downloaded, the first and most critical step is configuring the tool to securely connect to your GridDB Cloud account.

The tool relies on reading a `.griddb.yaml` file in your home directory. So go ahead and make one of those:

```shell
$ vim ~/.griddb.yaml
cloud_url:
"https://cloud5327.griddb.com:443/griddb/v2/gs_clustermfcloud5237/dbs/
ZV8LOPl8"
cloud_username: "M0gaXVkFEDG-user"
cloud_pass: "password"
```

This interactive prompt asks for three key pieces of information:

1. **GridDB Cloud FQDN:** The fully qualified domain name (hostname) of your GridDB Cloud portal.
2. **Username:** The username for your GridDB account.
3. **Password:** Your account password.

This local configuration file is responsible for securely storing your connection details and handling the underlying authentication tokens for all subsequent commands. This one-time setup means you do not need to provide your credentials for every operation.

12.3 Core Operations: Querying and Introspection

With the CLI configured, you can immediately begin interacting with your database. The primary function for data retrieval is the sql command. This command allows you to run any valid GridDB SQL query directly from your terminal.

For instance, to retrieve the 10 most recent records from a device_readings container, the command would be:

```shell
$ griddb-cloud-cli sql query -s "SELECT * FROM device_readings ORDER
BY timestamp DESC LIMIT 10" --pretty
```

The CLI sends this query to your GridDB Cloud instance and prints the result to the standard output as a JSON object. This raw JSON output is machine-readable and perfect for automation. For human readability, it can be used with the `pretty` flag to add indents and line breaks, or it can be piped to a tool like jq:

Shell

```
$ griddb-cloud-cli sql query -s "SELECT * FROM device1 ORDER BY ts
DESC LIMIT 10" | jq
{
  [
    {
      "Name": "ts",
      "Type": "TIMESTAMP",
      "Value": "2020-07-19T16:49:51.700Z"
    },
    {
      "Name": "co",
      "Type": "DOUBLE",
      "Value": 0.005918290620131231
    },
    {
      "Name": "humidity",
      "Type": "DOUBLE",
      "Value": 49.5
    },
    {
      "Name": "light",
      "Type": "BOOL",
      "Value": false
    },
    {
      "Name": "lpg",
      "Type": "DOUBLE",
      "Value": 0.008698836021386215
    },
    {
      "Name": "motion",
      "Type": "BOOL",
      "Value": false
    },
    {
      "Name": "smoke",
      "Type": "DOUBLE",
```

```
      "Value": 0.02341023929228802
    },
    {
      "Name": "temp",
      "Type": "DOUBLE",
      "Value": 21.9
    }
  ]
    ...
  ]
}
```

Beyond executing queries, the CLI provides essential "introspection" commands for exploring your database structure:

- **show <container>**: shows schema of container
- **list**: Lists all containers (the equivalent of tables) within a specified database.

These commands are fundamental for scripting, as they allow a script to dynamically discover database or container names before performing operations on them.

12.4 The CLI in Practice: A Tool for Automation

The true power of the GridDB Cloud CLI is realized when it is integrated into automated workflows. By removing the need for manual intervention, it enables robust, repeatable processes essential for modern IoT application management.

Consider a simple daily reporting script. A Bash script could use the CLI to execute a SQL query that aggregates the previous day's sensor data, formats the JSON output with jq, and emails the results to the operations team.

```shell
#!/bin/bash

# Fetch daily summary from GridDB Cloud
SUMMARY_DATA=$(griddb-cloud-cli sql query -s "SELECT MAX(temperature)
FROM device_readings WHERE timestamp > NOW() - 1 DAY")

# Format and send the report
echo "Daily Max Temperature Report:" > report.txt
echo $SUMMARY_DATA | jq . >> report.txt
# mail -s "Daily Report" ops-team@example.com < report.txt
```

In a CI/CD context, the CLI could be used in a pipeline to automatically verify that a database schema change was applied correctly or to load a set of test data into a container before running integration tests.

By providing a simple, scriptable interface for powerful database operations, the GridDB Cloud CLI becomes an indispensable tool in the IoT developer's toolkit, promoting efficiency, reliability, and automation.