

# IrregularDB: A model-based time series DBMS for regular & irregular time series data



**AALBORG UNIVERSITET**  
STUDENTERRAPPORT

MASTER PROJECT (P10)  
GROUP CS-22-DT-10-02  
SOFTWARE  
AALBORG UNIVERSITY  
JUNE 2022



**AALBORG UNIVERSITET**  
STUDENTERRAPPORT

**5th Year Software**  
Software  
Selma Lagerlöfs Vej 300  
9220 Aalborg Øst  
<https://www.aau.dk/>

**Title**

**IrregularDB:** A model-based time series DBMS for regular & irregular time series data

**Theme**

Time series databases

**Project period**

February 2022 - June 2022

**Project group**

cs-22-dt-10-02

**Participants**

Esben Kaa Nedergaard  
Kenneth Ljunggren Nørholm  
Simon Teodor Manojlovic

**Supervisor**

Christian Thomsen

**Amount of pages:** 73

**Appendices:** 4 (13 pages)

**Project finished:** 17-06-2022

**Abstract:**

Given the vast amount of time series data produced by sensors in different domains, specialized time series DBMSs are needed to ingest, compress, store, and analyze the time series data. Many time series DBMSs have already been developed and proven to be efficient but with the ever increasing requirements for collecting time series data new approaches should be explored such as systems that can efficiently handle both regular and irregular time series data.

In this project we design, implement, and test a new time series DBMS, IrregularDB. Tests show that IrregularDB achieves 1.03-2.69 times better compression, and 0.61-27.39 times the ingestion speed of existing open-source systems: InfluxDB and ModelarDB while supporting both regular and irregular time series data.

In order to achieve these results IrregularDB uses a novel approach we name Multi-Timestamp Multi-Value Model Compression (MTVMC). The approach applies one of multiple timestamp compression models to the timestamp data and one of multiple value compression models on the measured values. The timestamp- and value models are then stored as *segments* in a relational DBMS. From here the data can be queried with full SQL support directly on the DBMS.

# Summary

The amount of sensors that are used to capture time series data is increasing every year. The sensor data can be information such as temperatures or energy production from a wind turbine. No matter what specific information is measured most of these recorded values need to be stored, and depending on the use case some need to be stored long term. The long term data storage can become a challenge with the ever-increasing amount of data collected. Therefore, systems that are optimized for storing and processing time series data are a necessity.

Existing purpose-built time series DBMSs have among other strategies used *model-based compression* to handle these vast quantities of data. In this project we differentiate between two types of model-based compression: *timestamp-model-based compression* and *value-model-based compression*. Timestamp-model-based compression means that models are created to compress timestamps of a time series. Value-model-based compression is the same but for values. An example of value-model-based compression could be to fit the values of a time series interval to a linear function, e.g.  $f(x) = ax + b$ . Then this linear function's parameters can be saved instead of storing the individual values.

Model-based compression can be extended to *multi-model-based compression*. This is done by supporting multiple different model types and then selecting only the best model for a given time series interval. Multi-model-based compression can then be utilized for both timestamp-model-based compression and value-model-based compression by for example supporting multiple different value model types leading to *multi-value-model-based compression*.

Systems like *ModelarDB* [1] have applied multi-value-model-based compression. However, ModelarDB only supports regular time series data i.e. data points that are always measured with a fixed interval. The purpose of this project is to create a system inspired by ModelarDB but with support for both regular and irregular time series data, thereby increasing the scope of data that can be compressed with the strengths of multi-value-model-based compression. Other systems, e.g. Informix [2], have applied *multi-timestamp-model-based compression* to have better handling for both regular and irregular time series. The aforementioned systems have inspired this project to utilize both multi-value-model-based compression and multi-timestamp-model-based compression by creating the, to our knowledge, novel approach that we call Multi-Timestamp Multi-Value Model Compression (MTVMC).

The system produced in this project is called *IrregularDB*. IrregularDB can ingest data from either a TCP connection or CSV files. The system separates the ingestion from the processing of the data points such that the ingestion endpoints will always be responsive to the devices offloading their data. The MTVMC processing is performed by background worker threads, using several user-configurable parameters such as error bound and threshold. This leads to a time series being divided into segments of various lengths. Each segment represents a time series interval, a part of the time series, by storing model representations of the timestamps and the values. These models can then be used to recreate the time series interval represented by the given segment. The segments are sent in batches to a PostgreSQL database. The segments can then be queried using full SQL support from the database and a proof-of-concept user-defined PL/Java function can be used to decompress them.

IrregularDB was benchmarked against ModelarDB and *InfluxDB* for comparison. ModelarDB was chosen as it is the system that IrregularDB is most heavily inspired from and thus it is a

natural choice for performance evaluation. InfluxDB was chosen as it is a broadly used system [3] and is also often used by papers for comparison. For evaluation, the real-life dataset REDD [4] as well as a dataset created using the Time Series Benchmarking suite(TSBS) data generator, was used. The tests performed cover both compression ratio, ingestion speed, and query speed.

The test results show that the ideas utilized in IrregularDB to support MTVMC are worth investigating further, as IrregularDB shows promising results for both compression and ingestion. As for querying IrregularDB lacked behind the other two systems. But on the other hand, IrregularDB offers full SQL-query support, and handling of irregular time series, which ModelarDB does not.

Two ways have been identified, which should increase the query performance to be competitive with the other systems. The first way to increase query performance is to replace the proof-of-concept decompress segment function implemented in Java with a C implementation. The second improvement is to include additional summary information for segments since the summary information already implemented adds significant speed ups to relevant queries.

# Preface

This project is the result of three Aalborg University students' master's project under the supervision of Christian Thomsen. We would like to thank Mr. Thomsen for his continual support and expertise that has contributed to the outcome of the project. We also thank Søren Skejser Jensen for providing support and hotfixes for *ModelarDB* to get the desired query results. Lastly, we want to thank AAU for providing access to a server suitable for obtaining test results.

The first time glossaries appear in each chapter they will appear as follows: *IrregularDB*. Then afterward they will have normal formatting: IrregularDB. Similarly for definitions the first time they appear in a chapter they will look this: SEGMENT. Then afterward they will be plain text: segment.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Domain</b>	<b>2</b>
2.1	Time series databases . . . . .	2
2.2	Model-based compression . . . . .	3
2.3	Definitions . . . . .	5
2.4	Related work . . . . .	7
2.5	IrregularDB’s main idea . . . . .	11
<b>3</b>	<b>Algorithm Design</b>	<b>14</b>
3.1	Timestamp model types . . . . .	14
3.2	Value model types . . . . .	20
3.3	Fallback model types . . . . .	22
3.4	Segment generation . . . . .	22
3.5	Keeping the order of the data points . . . . .	26
<b>4</b>	<b>Design</b>	<b>30</b>
4.1	Architecture overview . . . . .	30
4.2	Data receiving . . . . .	32
4.3	Model-based Compression . . . . .	36
4.4	Component overview . . . . .	41
4.5	Database design . . . . .	42
<b>5</b>	<b>Implementation</b>	<b>47</b>
5.1	Overview . . . . .	47
5.2	Integration test . . . . .	47
<b>6</b>	<b>Evaluation</b>	<b>49</b>
6.1	Datasets . . . . .	49
6.2	Evaluation setup . . . . .	51
6.3	Storage usage . . . . .	54
6.4	Ingestion speed . . . . .	56
6.5	Query Speed . . . . .	60
<b>7</b>	<b>Future work</b>	<b>65</b>
7.1	Prettier querying . . . . .	65
7.2	Bucket sizes . . . . .	65
7.3	Parameter tuning . . . . .	66
7.4	Faster query times . . . . .	66

7.5	Not having to decompress models during ingestion . . . . .	66
7.6	Breaking threshold . . . . .	67
<b>8</b>	<b>Conclusion</b>	<b>68</b>
<b>A</b>	<b>Gorilla Time Stamp Model Type</b>	<b>74</b>
<b>B</b>	<b>Value model types</b>	<b>76</b>
B.1	PMC-mean Value Model Type . . . . .	76
B.2	Swing Filter Value Model Type . . . . .	78
B.3	Gorilla Value Model Type . . . . .	81
<b>C</b>	<b>User-configurable parameters</b>	<b>83</b>
<b>D</b>	<b>Initial Testing</b>	<b>85</b>
D.1	Length bound test . . . . .	85
D.2	Brute Force vs. Greedy Model Picker test . . . . .	85

# 1 Introduction

An increasing number of sensors are used to monitor devices. This has led to the generation of vast quantities of time series data. Time series data is often used for analysis to gain insight into the behavior of systems and their devices. This insight can help increase the reliability of a system by using the sensor data to detect anomalous behavior. Detecting anomalous behavior allows the system to correct its behavior before it has consequences e.g. downtime [5].

Time series data is not optimally stored and processed in traditional RDBMSs (Relational Database Management Systems). Because the RDBMSs are not well suited to handle the velocity and volume of the time series data produced by large sensor networks [6]. Time series data is better stored in Time Series Management Systems (TSMSs) as they are optimized towards quickly ingesting large amounts of time series data and storing the time series data compactly compared to the raw data. Furthermore, TSMSs are often optimized for answering typical time series queries.

The volumes of time series data produced by sensors can become large. For example, the sensors on a single Boeing 787 produce up to half a terabyte of data per flight [6]. The large volumes of data mean that storing the raw sensor data can, therefore, be infeasible or prohibitively expensive [1]. Compression is, therefore, often used to reduce the cost of storing the data. *ModelarDB* [1] has demonstrated *multi-model-based compression* of values to be effective at compressing time series data. Multi-model-based compression will be explained in more detail in **Section 2.2**. However, *ModelarDB* has the drawback that it only supports regular time series [1]. A regular time series is a time series where its data points are always measured with a fixed time interval between them. The main reason that *ModelarDB* only supports regular time series is that the domain *ModelarDB* was developed for (wind turbine monitoring data) consists of only regular time series.

It is far from all time series domains that only use regular time series data. For example, a system monitoring the stock market would only need to generate a data point whenever a stock changes its value. This results in irregular time series data as the data is not monitored with a fixed interval. Furthermore, regular measured time series can have imprecise readings resulting in almost regular time series data [7]. This means that systems often have to handle both regular time series and irregular time series, which is not possible in *ModelarDB*.

## Problem definition:

*ModelarDB* has shown that multi-model-based compression of values is effective at handling regular time series data. It is, therefore, interesting to research the possibilities of applying multi-model-based compression in a system that supports both regular and irregular time series data. The goal of this project is therefore to answer the following question:

*How should a time series management system be made such that it supports effective compression and efficient ingestion using multi-model-based compression of values for both regular and irregular time series?*

The time series management system developed in this project is named *IrregularDB* <sup>1</sup>.

---

<sup>1</sup><https://github.com/IrregularDB/IrregularDB>



## 2 | Domain

This chapter presents some necessary definitions and an understanding of the domain. **Section 2.1** describes some of the contexts wherein *time series databases* operate. **Section 2.2** introduces the concept of *model-based compression*. Some of the terms used will be formally defined in **Section 2.3**. In **Section 2.4** we will discuss the related works that use model-based compression for ingesting time series data. Lastly, the main idea of *IrregularDB* is presented in **Section 2.5**.

### 2.1 Time series databases

The domain of time series databases is characterized by having a large amount of data, requiring high throughput and availability. An example of a business domain that produces large amounts of sensor data is the energy production domain for wind turbines, which the *ModelarDB* paper [1] focuses on. The paper presents an example company with 16 windmill parks where each park has 12 wind turbines on average. Each turbine is monitored by 98 sensors that each measures every 100 ms. In total, this leads to:

- 18816 unique TIME SERIES
- $\sim 11.3$  million DATA POINTS per minute
- 5.5 TiB data per month if timestamps are stored using a 64-bit long and the value is stored using a 32-bit float.

It is not only the energy production domain that produces large-scale data. Large-scale web applications such as Facebook, TikTok, and Netflix that require sophisticated monitoring of e.g. servers to offer availability for their users also produce massive amounts of data. This monitoring produces vast quantities of data making it challenging to ingest and store the data efficiently while still offering acceptable query times. Considering the numbers mentioned in the Gorilla [8] paper from 2015, a system developed by Facebook to monitor their systems, Gorilla should be able to handle:

- 2 billion unique time series.
- 700 million data points per minute.
- $\sim 40,000$  queries per second at peak.

To handle these amounts of data the systems used need to provide high insertion throughput and low query latency with efficient storage based on compression. Storing that amount of data in a traditional SQL database as raw data is not feasible in practice. Therefore, specialized time series databases are used that efficiently ingest, compress, and offer acceptable query times [9].

## 2.2 Model-based compression

Time series databases typically implement a strategy to compress the time series data. One approach is to use model-based compression. The idea behind model-based compression is that instead of storing the raw data points one could simply store models that can be used as a ‘formula’ for how to reconstruct the original data points. Model-based compression can be implemented as either lossless or with an allowed deviation. Allowing deviations mean that the reconstructed data points are approximations. In general, a higher deviation will allow for a higher rate of compression at the cost of precision.

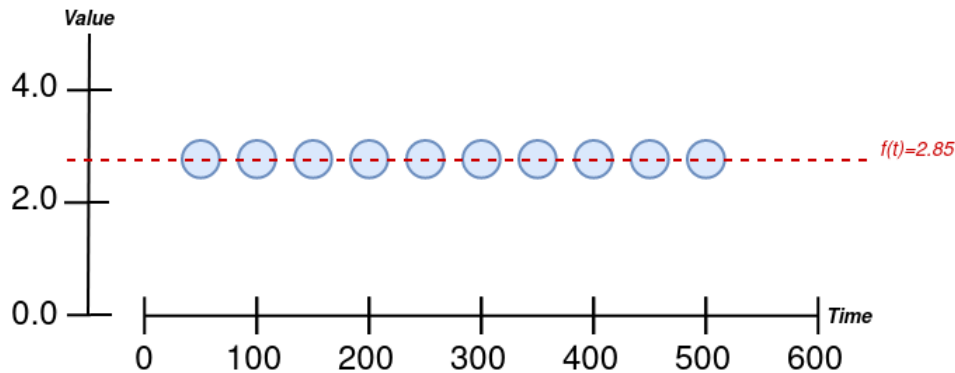
In this project we differentiate between two types of model-based compression:

- **Timestamp-model-based compression:** TIMESTAMP MODELS are created that compress the timestamps of the data points in a time series.
- **Value-model-based compression:** VALUE MODELS are created that compress the values of the data points in a time series.

This differentiation means that the data points are reconstructed by using the timestamp model to reconstruct their timestamps and the value model to reconstruct their values.

### 2.2.1 Value model-based compression example

**Figure 2.1** shows a subset of ten data points from a fictional time series. This subset is called a TIME SERIES INTERVAL as it only shows a portion of the entire time series. To keep the example simple we will only be focusing on constructing the value model using a single VALUE MODEL TYPE.



**Figure 2.1:** Simple example of a value model fitted to a set of data points from a time series.

The dotted red line represents the value model, which represents the values of the data points. In this example, the value model that fits the values of the time series interval could be  $m_v := f(t) = 2.85$  as all the values of the time series interval are equal to 2.85. This example compressed ten values to a single value resulting in a 10x compression ratio for the values.

The value model created in this example was created using the PMC-mean value model type, which is discussed in more detail in **Appendix B.1**. PMC-mean works by producing constant models such as  $m_v$  by trying to fit all values in the time series interval to a single constant value. This value model type is therefore highly dependent on the characteristics of the data points' value patterns.

If the data points do not all have the same value this value model type can not be used. However, if an error bound  $\varepsilon_v$  can be tolerated we can model data points with some variance in their values with the same model. If we allow an error of  $\varepsilon_v = 25\%$ , meaning the values of the reconstructed data points can deviate up to 25% of the original measured value, we can use the same value model type as before to construct a value model that represents the values of the data points in **Figure 2.2**.

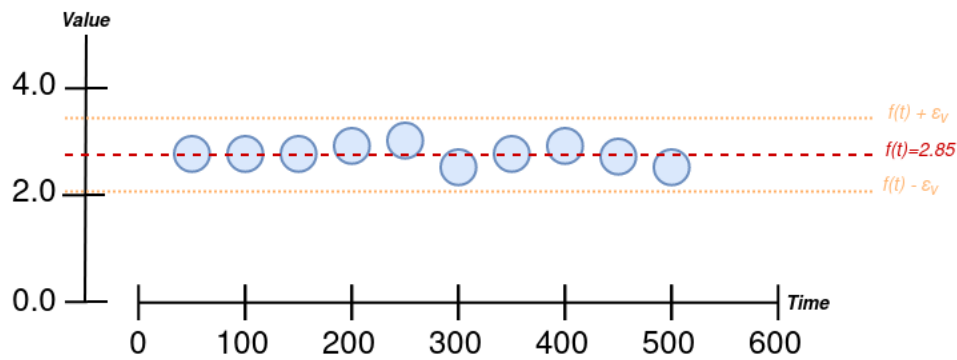


Figure 2.2: Model example with error bound.

### 2.2.2 Multi-model-based compression

The above example showed how model-based compression can lead to significant compression results as long as the data fits the model types used. A single model type cannot fit all data point patterns, which leads to the use of multiple model types, a technique known as *multi-model-based compression*. Multi-model-based compression attempts to fit the same data points to different model types and will then store the model of the model type that achieved the best compression ratio. By using a multi-model-based compression approach, ingestion time is sacrificed for an increase in compression ratio, which is the use case IrregularDB focuses on.

The differentiation between two types of model-based compression also leads to two types of multi-model-based compression:

- **Multi-timestamp-model-based compression:** the timestamps of the data points are fitted to multiple different `TIMESTAMP MODEL TYPES` and only the timestamp model achieving the best compression is stored for each time series interval.
- **Multi-value-model-based compression:** the values of the data points are fitted to multiple different value model types and only the value model achieving the best compression is stored for each time series interval.

## 2.3 Definitions

This section gives a set of definitions that forms a common foundation for the domain worked with in this project. The project focuses on handling data from sensors and monitoring devices, which produce data points. Our formal definition of a data point is shown in **Definition 2.3.1**.

**Definition 2.3.1** (Data point): A data point is a pair  $d = (t, v)$  consisting of a timestamp  $t \in \mathbb{N}$  and a value  $v \in \mathbb{R}$  recorded at time  $t$ .

For the timestamps, we use natural numbers as we choose to represent time using Unix epoch time meaning that each timestamp is represented by storing the number of milliseconds that have elapsed since January 1, 1970.

The data points are then combined into a sequence of data points, a time series, which is defined in **Definition 2.3.2**.

**Definition 2.3.2** (Time series): A time series  $TS = \langle (t_1, v_1), (t_2, v_2), \dots \rangle$  with a tag  $T$  is a sequence of data points that is uniquely identified by  $T$ . For each data point in the time series, its timestamp must be larger than the previous i.e.  $t_i < t_{i+1}$  for  $i \geq 1$ .

To identify which data points belong to which time series the data points are received as TIME SERIES READINGS (**Definition 2.3.3**).

**Definition 2.3.3** (Time series reading): A time series reading  $R$  is a pair  $R = (T, d)$  consisting of a tag  $T$  and a data point  $d$ , where  $T$  is a string used to identify which time series  $d$  belongs to.

The definition of a time series shown in **Definition 2.3.2** is unbounded, which means that the time series can contain an arbitrary amount of data points. However, when working with time series we often only work with fractions of them. We call these fractions time series intervals, which are defined in **Definition 2.3.4**. We denote the number of data points in a time series interval called  $TSI$  as  $|TSI|$ .

**Definition 2.3.4** (Time series interval): A time series interval  $TSI = \langle d_1, d_2, \dots, d_n \rangle$  is a subsequence of consecutive data points from a time series  $TS$ . A time series interval can also be seen as a bounded time series.

Model-based compression is used in IrregularDB to reduce the amount of storage space needed to store the time series intervals. This means that IrregularDB stores models for the different time series intervals instead of the raw data points. In IrregularDB we differentiate between two types of models: Timestamp models (see **Definition 2.3.5**) which represent timestamps and value models (see **Definition 2.3.6**) which represent values. As an example of a model consider the following time series interval  $TSI$  and the value model  $m_v$ , which is a linear function that can represent the values of  $TSI$ :

- $TSI = \langle (0ms, 50), (100ms, 150), (200ms, 250), (300ms, 350) \rangle$
- $m_v := f(t) = 1 \cdot t + 50$

Users can define allowed levels of imprecision in the constructed models because this can be used to further reduce the amount of storage used. Imprecisions can lead to further compression as it allows the models to fit more data points. This means that for the value models the users can specify a percentage error bound  $\varepsilon_v$ , which specifies the percentage allowed deviation. However, for the timestamp models it is not ideal to use percentage error, as discussed in more detail in **Section 3.1.1**, so instead a threshold value  $T_t$  is used, which specifies the allowed deviation in milliseconds.

**Definition 2.3.5** (Timestamp model): A timestamp model  $m_t$  with threshold  $T_t$  for a time series interval  $TSI = \langle (t_1, v_1), (t_2, v_2), \dots, (t_n, v_n) \rangle$  is a representation that can be used to reconstruct approximations  $\langle t_1^*, t_2^*, \dots, t_n^* \rangle$  of the timestamps  $\langle t_1, t_2, \dots, t_n \rangle$ , where it holds that  $|t_i - t_i^*| \leq T_t$  for  $1 \leq i \leq n$ .

**Definition 2.3.6** (Value model): A value model  $m_v$  with error bound  $\varepsilon_v$  for a time series interval  $TSI = \langle (t_1, v_1), (t_2, v_2), \dots, (t_n, v_n) \rangle$  is a representation that can be used to reconstruct approximations  $\langle v_1^*, v_2^*, \dots, v_n^* \rangle$  of the values  $\langle v_1, v_2, \dots, v_n \rangle$ , where it holds that  $\frac{|v_i - v_i^*|}{v_i} \leq \varepsilon_v$  for  $1 \leq i \leq n$ .

These models are constructed using timestamp model types that do timestamp-model-based compression, and value model types that do value-model-based compression. The definition of the model types are seen in **Definition 2.3.7** and **Definition 2.3.8**. As an example, a value model type could be  $M_V = \text{SWING}$ , because SWING as explained in **Appendix B.2** outputs linear functions that can be used as value models.

**Definition 2.3.7** (Timestamp model type): A timestamp model type  $M_T$  and its provided threshold  $T_t \geq 0$  is a function that takes a time series interval  $TSI = \langle (t_1, v_1), (t_2, v_2), \dots, (t_n, v_n) \rangle$  as input and returns a timestamp model  $m_t$ . The timestamp model  $m_t$  can be used to create approximations  $\langle t_1^*, t_2^*, \dots, t_n^* \rangle$  of the timestamps  $\langle t_1, t_2, \dots, t_n \rangle$ , where it holds that  $|t_i - t_i^*| \leq T_t$  for  $1 \leq i \leq n$ .

**Definition 2.3.8** (Value model type): A value model type  $M_V$  and its provided error bound  $\varepsilon_v \geq 0$  is a function that takes a time series interval  $TSI = \langle (t_1, v_1), (t_2, v_2), \dots, (t_n, v_n) \rangle$  as input and returns a value model  $m_v$ .  $m_v$  can be used to create approximations  $\langle v_1^*, v_2^*, \dots, v_n^* \rangle$  of the values  $\langle v_1, v_2, \dots, v_n \rangle$ , where it holds that  $\frac{|v_i - v_i^*|}{v_i} \leq \varepsilon_v$  for  $1 \leq i \leq n$ .

SEGMENTS are used to keep track of the timestamp models and value models used to represent the different time series intervals of a time series.

**Definition 2.3.9** (Segment): A segment is a 5-tuple  $S = (T, t_s, t_e, m_t, m_v)$  that represents data points for a time series interval  $TSI = \langle (t_s, v_s), (t_{s+1}, v_{s+1}), \dots, (t_e, v_e) \rangle$  where  $|TSI| \geq 1$  where:

- $T$ : is the tag of the time series that  $TSI$  is a subsequence of
- $t_s$ : is the start time of  $TSI$
- $t_e$ : is the end time of the  $TSI$
- $m_t$ : is a timestamp model approximating the timestamps of  $TSI$
- $m_v$ : is a value model approximating the values of  $TSI$

Another term used throughout the report is the SAMPLING INTERVAL, which can be used to describe certain time series intervals, where the time elapsed for each data point remains constant. The sampling interval is defined in **Definition 2.3.10**.

**Definition 2.3.10** (Sampling interval): For a time series interval  $TSI = \langle (t_1, v_1), (t_2, v_2), \dots, (t_n, v_n) \rangle$  where it holds that  $t_{i+1} - t_i = t_{i+2} - t_{i+1}$  for  $1 \leq i < n-1$  then the time elapsed between the timestamps in  $TSI$  is referred to as its sampling interval.

## 2.4 Related work

The paper primarily discussed in this section is the ModelarDB paper [1] as IrregularDB is inspired by ModelarDB.

### 2.4.1 ModelarDB

ModelarDB's [1] main contribution is to introduce a new category of model-based compression methods called *Multi-Model Group Compression (MMGC)*. MMGC is a combination of two other methods:

- **Multi-model-based compression (MMC)**: refers to selecting the best model from a set of models (as explained earlier in **Section 2.2.2**). It is worth noting that what is referred to as a model type in ModelarDB is what this project refers to as a value model type. This means that the multi-model-based compression that ModelarDB supports is what we call multi-value-model-based compression.
- **Model-based Group Compression (MGC)**: refers to grouping together time series with similar value patterns in a so-called *Time Series Group (TSG)*. The idea is then to only store models for one of the time series in the group and then apply a scaling factor to these models to get values for the other time series in the group. This approach can lead to a better compression ratio by having to store fewer segments compared to compressing each time series individually.

ModelarDB is used only for ingesting time series with a constant sampling interval. An overview of the system components can be seen in **Figure 2.3**. The system is a multi-node system running

in an Apache Spark cluster. A partitioner component splits time series into Time Series Groups and assigns each Time Series Group to a worker node. The worker node generates segments by using the MMGC algorithm and stores segments in persistent storage. A query processing component is responsible for reconstructing original data points from segments when the user queries data points.

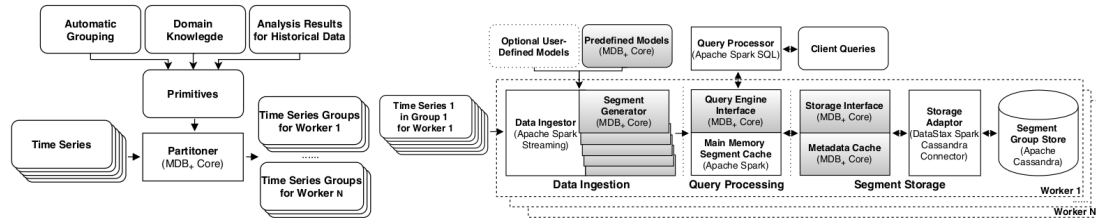


Figure 2.3: ModelarDB component overview [1].

ModelarDB offers an API to extend the system with user-defined model types by implementing an interface. The system only supports time series with constant sampling intervals, which is key for achieving a state-of-the-art compression and query performance. The reason for this choice is based on ModelarDB’s domain focus on energy production, namely wind turbine energy production where high-quality wired sensors are used to monitor and send data, which ensures the data is regular. Not all time series domains can safely make the same assumption about only processing regular time series data.

## 2.4.2 Other related work

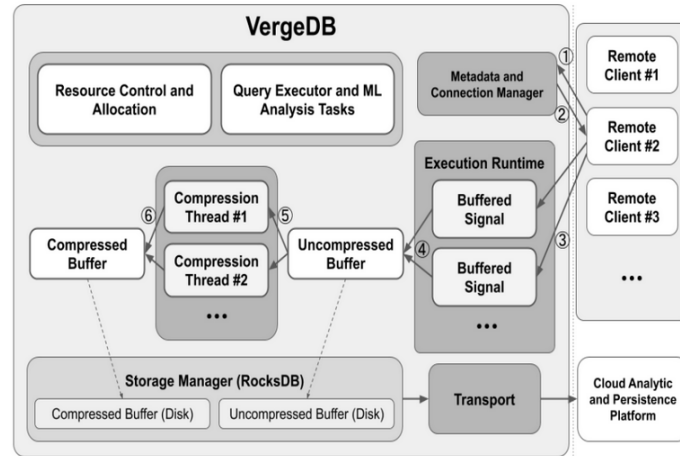
This section describes related time series database systems that also use some form of model-based compression.

### Towards Online Multi-Model Approximation of Time Series

The ‘*Towards Online Multi-Model Approximation of Time Series*’ paper [10] is among the first to propose a multi-value-model-based compression approach for time series data. The paper’s main contribution is to propose an algorithm used to select the value model type that results in the best compression ratio for a specific segment among a set of value model types. Furthermore, the paper proves that their algorithm will always produce a fewer or equal amount of segments compared to any single model-based compression approach, which should lead to a better compression ratio.

### VergeDB

VergeDB [11] can be used as a lightweight storage engine or as an edge device database that compresses and offers analysis of compressed and uncompressed data. In other words, VergeDB ingests, compresses, and stores data on edge devices. The system has support for various compression methods and will attempt to select the best compression method based on available system resources and ingestion ratio. VergeDB focuses on better utilizing new edge device capabilities and the trend in workloads being shifted towards machine learning. **Figure 2.4** is an overview of the system components.



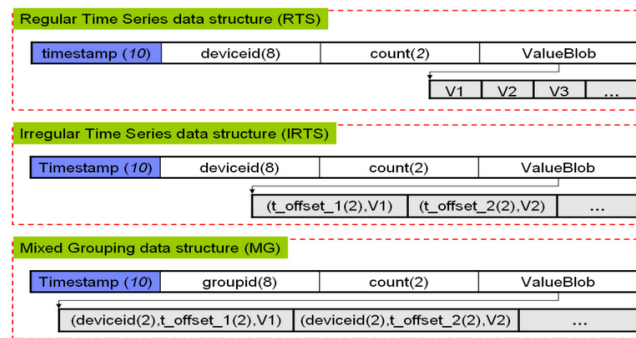
**Figure 2.4:** System component overview of the VergeDB system [11]

In step 5 the compression threads dynamically choose a specific compression algorithm depending on parameters such as storage capacity, network bandwidth, and ingestion rate. VergeDB does not attempt to achieve the optimal compression ratio always but adaptively selects a model type depending on current resource usage. Some of the compression methods VergeDB supports are Gzip, snappy, and Gorilla. VergeDB’s model-based compression compresses the time series as a combined entity i.e. their model types are a combined value model type and timestamp model type.

The VergeDB paper [11] also discovers that compression methods, that require full decompression of stored data before users can perform data analytics, are weaker than other compression methods because they give extended query time due to decompression overhead.

### Informix extension

The paper [2] suggests a distributed TSDBMS that uses three different storage formats for different types of time series. The compression used depends on the time series type. The system supports both lossless and lossy (within the user-defined error bound) compression. The three formats can be seen in **Figure 2.5**.



**Figure 2.5:** The different storage formats proposed by [2].



The data format used depends on the time interval that the source time series are sampled. The first two formats are similar in that they store a *timestamp*, *deviceid*, *count*, and *valueBlob* header. The *valueBlob* header is a pointer to the compressed time series data. The regular time series structure only stores values in the blob data, as for regular time series the timestamps can be reconstructed simply using the sampling interval. For the irregular time series the data structure stores timestamp-value pairs in the blob data. Both the regular and irregular time series blob data structure only contains data from a single time series source and each structure can hold up to  $b$  data points. The  $b$ -value is user-configurable.

The mixed grouping data structure is different by grouping different time series sources in a single blob. In this blob, triples are stored: (deviceid, timestamp, values). The system maintains which group a device is in. The mixed grouping data structure is used for time series with a low sampling interval. The mixed grouping data structure is compressed using a quantization algorithm.

Informix uses multi-timestamp-model-based compression as it uses two different methods to store the regular and irregular timestamps. It also supports value-model-based compression as both the regular and irregular data structures are compressed using the linear compression algorithm [12].

## Heracles

Heracles [9] focuses on improving the existing performance monitoring time series systems. To do this they made the following two main contributions:

- **New storage model:** Heracles uses a new storage model that tries to remove the issue of storing the same timestamps multiple times. This is done by grouping together time series with the same timestamps and then utilizing a shared timestamp column.
- **Two-level epoch memory manager:** They designed an improved memory manager that keeps data available for querying in-memory data for longer periods by not immediately discarding data after flushing but instead keeping it in a two-level cache before discarding it.

We will focus on describing the new storage model as it is the most relevant part in relation to IrregularDB's domain. The paper's main idea is to store each timestamp only once and create a mapping from these timestamps to the values associated with the timestamps. They created two different mappings and storage compression methods one for in-memory and one for on-disk storage. We only focus on the compression methods. The compression methods work as follows:

- **In-memory compression:** for value compression they use a value model type similar to Gorilla's value model type [8]. For timestamp compression, they develop a timestamp model type, where they store a base timestamp  $t_b$  and then store for each of the remaining timestamps  $t_i$  store a delta value calculated as  $\Delta = t_i - t_b$ .
- **On-disk compression:** they use the same value model type as in-memory, but improve it by e.g. choosing more efficient base values. For their on-disk timestamp model type, they choose to calculate an approximation of the sampling interval  $SI$ . Then for each timestamp  $t_i$  they store the difference  $d = t_i - (t_1 + SI \cdot (i - 1))$ , which is how far away the actual time is from a value recreated using the calculated  $SI$ . This approach should perform better

than the in-memory timestamp model type as this difference value should be smaller than the deltas from the base timestamp.

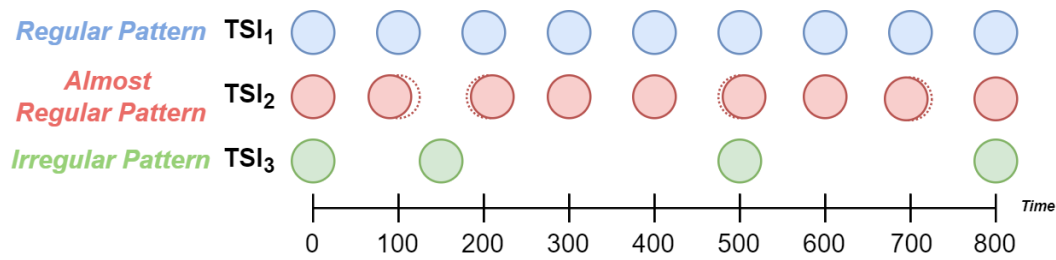
Heracles, therefore, utilizes multi-timestamp-model-based compression as it uses two different timestamp model types for in-memory and on-disk compression.

## 2.5 IrregularDB's main idea

The main idea utilized in IrregularDB is to use multi-timestamp-model-based compression as well as multi-value-model-based compression. This is achieved by utilizing multiple timestamp model types and multiple value model types to compress data points. It was also chosen to compress timestamps and values separately in IrregularDB as it can result in better compression. We call this, to our knowledge, novel approach Multi-Timestamp Multi-Value Model Compression (MTVMC).

### 2.5.1 Multi-timestamp-model-based compression

The reason for utilizing multiple different timestamp model types is that sensors can produce time series intervals that follow various *timestamp patterns*. Certain timestamp model types are better suited to handle certain timestamp patterns. **Figure 2.6** illustrates some of the patterns that can occur for the time series' timestamps.



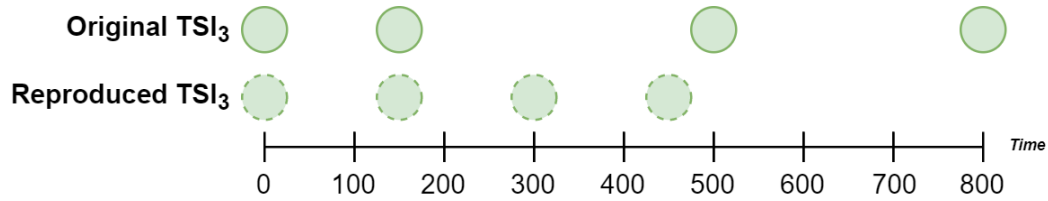
**Figure 2.6:** Example of various timestamp patterns

IrregularDB differentiates between the following categories of timestamp patterns:

- **Regular timestamp pattern:** If the timestamps of the data points in the time series interval arrive with a fixed sampling interval then the time series interval follows the *regular timestamp pattern*.
- **Almost regular timestamp pattern:** In some cases regularly sampled time series do not produce exactly regular data points due to noise or imprecise sensors. We identify these time series intervals as almost regular.
- **Irregular timestamp pattern:** Some time series contain data that might be sampled randomly or based on other factors e.g. the value of a stock changing. Time series intervals that do not follow the *regular timestamp pattern* nor the *almost regular timestamp pattern* are classified as following the *irregular timestamp pattern*.

As an example of how different timestamp model types are better suited to handle certain timestamp patterns consider a simple timestamp model type called  $M_{T_{simple}}$  that produces timestamp models by calculating a sampling interval for a time series interval by taking the difference between the first two timestamps i.e.  $SI = t_2 - t_1$ . The timestamp models produced by  $M_{T_{simple}}$  would then be:  $m_t := f(i) = t_1 + SI \cdot (i - 1)$ , where  $i$  is the index of the data point in the time series interval.

$M_{T_{simple}}$  would work well for  $TSI_1$  from **Figure 2.6** but would be ill-suited for  $TSI_3$  as it would reconstruct data points with timestamps that are very different from the original timestamps as shown in **Figure 2.7**.



**Figure 2.7:** Example of the simple timestamp model type  $M_{T_{simple}}$  being ill-suited

Instead, some other timestamp model types such as the Gorilla timestamp model  $M_{T_{gorilla}}$  explained in **Appendix A** would be better suited to construct a timestamp model for time series intervals, such as  $TSI_3$ , that follow an *irregular timestamp pattern*. This example, therefore, illustrates why a system that supports regular and irregular time series could benefit by utilizing multiple different timestamp model types thereby supporting multi-timestamp-model-based compression.

The patterns are primarily used as an abstraction by the developers of IrregularDB to determine which timestamp model types should be supported in IrregularDB to enable better timestamp compression. The users of IrregularDB do not have to specify which timestamp patterns their time series follow.

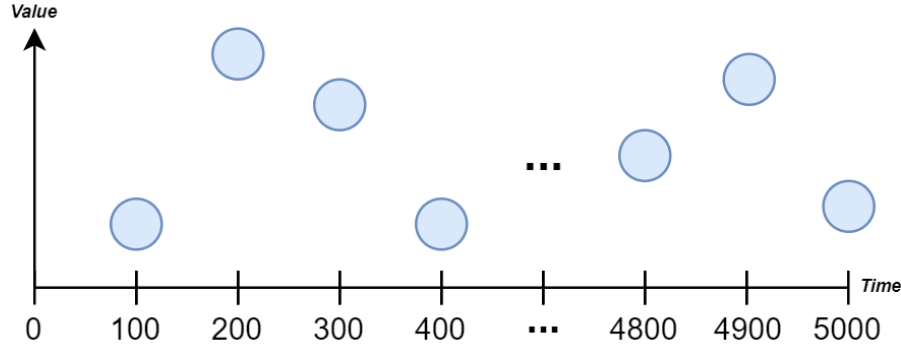
### 2.5.2 Multi-value-model-based compression

The idea is that IrregularDB will use multi-value-model-based compression by utilizing multiple value model types. The reason for using multi-value-model-based compression is that multi-value-model-based compression has been shown to always produce equal or better compression compared to single model value compression in papers such as [1] and [10].

### 2.5.3 Separating compression of timestamps and values

Existing systems often look at data points as a single entity and compress both the values and timestamps together instead of separating them. This leads to them using a predefined combination of the timestamp model type and value model type e.g.  $M_{gorilla} = (M_{T_{gorilla}} M_{V_{gorilla}})$ , which is a combination of Gorilla's timestamp model type (**Appendix A**) and value model type (**Appendix B.3**). There are cases where the predefined combinations of model types do not

give the optimal compression, which could instead be achieved by changing either the timestamp model type or value model type. Consider the example shown in **Figure 2.8**.



**Figure 2.8:** Example of time series interval for which combined Gorilla model type is not ideal

In the figure, the data points arrive with a fixed sampling interval so they follow a *regular timestamp pattern*. Using the Gorilla timestamp model type i.e.  $M_{T_{gorilla}}$  would not give the optimal compression. The timestamps could instead be further compressed using a timestamp model type that uses a sampling interval value to reconstruct the timestamps. For example a timestamp model type similar to  $M_{T_{simple}}$ . For the values, the best compression model would still be  $M_{V_{gorilla}}$ . The reason for this is that the values cannot be represented using a linear function or a constant function within a reasonable error bound.

The idea is to have no predefined combinations in IrregularDB but instead define timestamp model types and value model types separately. IrregularDB should therefore compress the timestamps and values separately, then select the combination of model types that achieves the best compression.

#### 2.5.4 Compared to related work

Our novel Multi-Timestamp Multi-Value Model Compression (MTVMC) approach is a new contribution because, to our knowledge, no work has been published that supports:

- Multi-timestamp-model-based compression
- Multi-value-model-based compression
- Does compression of timestamps and values separately.

For example, ModelarDB [1] supports multi-value-model-based compression as it utilizes multiple value model types. However, ModelarDB only supports regular time series, therefore, only supports a single timestamp model type. Another example is the Informix extension, discussed in [2], which supports multi-timestamp-model-based compression as they utilize multiple different timestamp model types but Informix only support a single value model type.

## 3 | Algorithm Design

This chapter will describe the design of the model types used in *IrregularDB*. The **TIMESTAMP MODEL TYPES** will be discussed in **Section 3.1** and the **VALUE MODEL TYPES** in **Section 3.2**. Then, before discussing the segment generation the necessity of *Fallback* model types is discussed in **Section 3.3**. Then **Section 3.4** describes how the model types are utilized for creating **SEGMENTS** representing **DATA POINTS**. **Section 3.5** describes the solution to a problem with the threshold affecting ordering of timestamps.

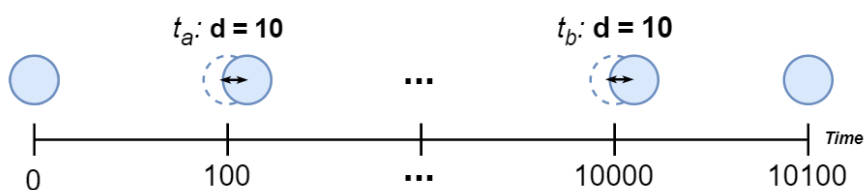
The following sections will mention user-configurable metrics. These user-configurable metrics are included in *IrregularDB* to allow its users to customize the system to fit their needs. An overview of the metrics can be found in **Appendix C**.

### 3.1 Timestamp model types

The timestamp model types implemented in *IrregularDB* to support *multi-timestamp-model-based compression* are *Regular*, *DeltaDelta*, and *SIdiff*. Before describing these timestamp model types the concept of threshold used for timestamps is discussed in **Section 3.1.1**. Then the three timestamp model types will be described in **Sections 3.1.2 to 3.1.4**.

#### 3.1.1 Error for timestamps

The users of *IrregularDB* can allow deviations in the reconstructed timestamps. Allowing deviation can result in higher compression. The naive approach used to allow deviation could be to use percentage error. However, due to the nature of the time dimension it is not ideal. The problem with this approach is that the same difference between the original timestamp  $t_i$  and its approximation  $t_i^*$  would give different percentage errors depending on the value of  $t_i$ . To illustrate this consider the timestamps shown in **Figure 3.1**.



**Figure 3.1:** Example of timestamps where the same difference between actual and approximated timestamps gives different error percentages

In **Figure 3.1** the original timestamps are shown as full circles and approximations are shown with dotted lines. As seen in the figure for both  $t_a$  and  $t_b$  the difference between them and their approximations is 10 but we would get the following percentage errors:

$$\bullet \ t_a: e(t_a, t_a^*) = \frac{|t_a - t_a^*|}{t_a} = \frac{|100 - 110|}{100} \approx 9.09\%$$

$$\bullet \ t_b: e(t_b, t_b^*) = \frac{|t_b - t_b^*|}{t_b} = \frac{|10010 - 10000|}{10010} \approx 0.1\%$$

Percentage error is therefore not satisfactory because the same difference would give smaller errors as time passes. The problem is also magnified by the fact that IrregularDB uses Unix epoch time as mentioned in **Section 2.3**. This means that timestamps will reach quite high base values because for example 1/1/2022-T00:00:00 has the timestamp 1640991600000. These high base values result in differences having to be quite big to give meaningful percentage errors or the users having to use small error bounds.

### Threshold

Instead, a threshold value  $T_t$  is used. The motivation for the threshold is to ensure that differences are treated equally no matter what the current value of the timestamp is. This means that for a timestamp  $t_i$  and its approximation  $t_i^*$  their difference should be less than the threshold  $T_t$  as seen in the definition for timestamp model types: **Definition 2.3.7**.

The downside of this approach is that threshold values are strongly affected by how often data points arrive for any given TIME SERIES. For example, a time series sampled every 10ms would be more affected by a threshold of  $T_t = 10$  than a time series sampled every 1000ms. IrregularDB, therefore, supports individual thresholds for each time series as seen in **Appendix C**.

### 3.1.2 Regular timestamp model type

The Regular timestamp model type is based on the timestamp model type used in *ModelarDB* [1]. The idea behind this timestamp model type is that for a TIME SERIES INTERVAL  $TSI = \langle t_s, t_{s+1}, \dots, t_e \rangle$  simply save a start time  $t_s$ , end time  $t_e$ , and a SAMPLING INTERVAL  $SI$  and use these values to reconstruct approximations of the timestamps by using **Equation (3.1)**.

$$t_i^* = t_s + SI \cdot (i - 1) \text{ for } i \in \langle s, \dots, e \rangle \quad (3.1)$$

#### Determining the sampling interval naively

In IrregularDB the sampling interval is not known before constructing segments. It is therefore necessary to be able to determine a sampling interval for a regular time series interval. Ideally, the sampling interval found should allow as many data points as possible in the segment when the threshold is considered.

The first approach considered for determining the sampling interval is similar to  $M_{T-simple}$  discussed in **Section 2.5.1**. The idea is to simply calculate the difference between the first two timestamps and use this as the sampling interval. This can, however, easily result in sub-optimal compression as the first two timestamps can easily have deviations compared to the actual SI. Consider the following list of timestamps, which come from a time series with a sampling interval of 100, as a small example of the problems with this approach:

$$\langle 0, 90, 200, 305, 400 \rangle$$

The second and fourth timestamps did not arrive at their exact expected time due to either noise or imprecise sensors. If the naive approach is used the sampling interval is calculated to

be  $SI = 90$  and the recreated timestamps would then be:

$$\langle 0, 90, 180, 270, 360 \rangle$$

The recreated timestamps will continuously drift further from the original timestamps. Only the first two timestamps can be represented with, say, a threshold  $T_t = 15$ , since the difference between the third approximated timestamp and the actual timestamp is  $|t_3 - t_3^*| = |200 - 180| = 20$  which is larger than  $T_t = 15$ . This approach is, therefore, prone to represent shorter time series intervals than what is possible.

### Improvements to determining a sampling interval

The approach shown in **Listing 3.1** is a suggested improvement to combat the problem of short segments from the naive approach.

```

1 Input: Stream of timestamps  $S_T = \langle t_1, t_2, \dots \rangle$ , threshold  $T_t$ .
2 Output: The found SI as an integer.
3
4  $SI \leftarrow t_2 - t_1$ 
5  $TimestampList \leftarrow \langle t_1, t_2 \rangle$ 
6  $i \leftarrow 2$ 
7 while  $S_T.hasNext()$  do:
8    $i \leftarrow i + 1$ 
9    $t_i \leftarrow S_T.next()$ 
10   $TimestampList.add(t_i)$ 
11   $t_i^* \leftarrow t_1 + SI \cdot (i - 1)$ 
12  if not  $withinThreshold(t_i, t_i^*, T_t)$  then:
13    // Current SI does not fit. So calculate and test new Candidate SI
14     $duration \leftarrow t_i - t_1$ 
15     $candidateSI \leftarrow \text{round}(duration / (TimestampList.size() - 1))$ 
16     $j \leftarrow 1$ 
17    while  $j < TimestampList.size()$  do:
18       $t_j^* \leftarrow t_1 + SI \cdot (j - 1)$ 
19      if not  $withinThreshold(TimestampList.get(j), t_j^*, T_t)$  then:
20        // Candidate SI could not fit so return previous SI
21        return  $SI$ 
22       $j \leftarrow j + 1$ 
23     $SI \leftarrow candidateSI$ 
24
25 return  $SI$ 

```

**Listing 3.1:** Improved Regular approach

The idea in the improved approach is as before to initially calculate the SI from the first two timestamps as seen on line 4. The difference is that a new candidate SI is calculated when the current timestamp cannot be fitted to the current sampling interval as seen on lines 12-15. The candidate SI is then tested against all the ingested timestamps on lines 17-19. If one of the ingested timestamps is not within the threshold  $T_t$  then the old SI is returned as seen on line 21. Otherwise, the current sampling interval is updated as seen on line 23 and more timestamps can be ingested.

To illustrate how this improvement allows the Regular timestamp model type to fit more timestamps consider the timestamps from the example in the naive approach. With the naive approach, only

the first two timestamps could be represented using the initial sampling interval  $SI = 90$ . With the improved approach, a new candidate sampling interval is calculated:  $SI = \frac{200-0}{3-1} = 100$ . Using the new candidate sampling interval we would get the following differences between the real timestamps and their approximations:

$$\langle |0 - 0| = \mathbf{0}, |90 - 100| = \mathbf{10}, |200 - 200| = \mathbf{0}, |305 - 300| = \mathbf{5}, |400 - 400| = \mathbf{0} \rangle$$

These differences are all below the threshold  $T_t = 15$  meaning that the improved approach for recalculating the sampling interval can fit all the timestamps. The new method can be used to determine a better sampling interval  $SI = 100$  because that sampling interval can represent more timestamps within the provided threshold.

This improvement of the Regular timestamp model type allows it to fit more timestamps, however, it adds some computational overhead. The effect of this computational overhead is limited by the fact that IrregularDB uses a max segment length for segments as discussed in **Section 3.4.1**.

### 3.1.3 SIdiff timestamp model type

The main idea utilized in SIdiff is inspired by Heracles's [9] on-disk time stamp compression. The idea is to compute a sampling interval  $SI$  for a sequence of timestamps  $T = \langle t_1, t_2, \dots, t_n \rangle$  by using **Equation (3.2)**.

$$SI(T) = \frac{t_n - t_1}{n - 1} \quad (3.2)$$

The SIdiff timestamp model type stores a sequence of differences ( $d$ ) between actual timestamps and expected timestamps. The function for calculating the difference  $d$  for the  $i$ 'th timestamp between the actual timestamp  $t_i$  and its expected timestamp is shown in **Equation (3.3)**.

$$d(i) = t_i - (t_1 + SI \cdot (i - 1)) \quad (3.3)$$

The expected timestamps as seen in **Equation (3.3)** are calculated for each timestamp by using the sampling interval and the start time of the sequence of timestamps, this is similar to **Equation (3.1)**

#### Bucket encoding

The data needs to be stored in a database that uses a fixed number of bytes per value so no "real" compression is achieved as long as the  $d$  values would need to be stored using integers. Therefore, an integer variable-length encoding is used to actualize the compression gained from using the SIdiff timestamp model type.

The variable-length encoding scheme used to store each difference value  $d$  is shown in **Listing 3.2**. This variable-length encoding is referred to as bucket encoding because the values are placed in different buckets based on their size. The bucket encoding from **Listing 3.2** ensures that smaller values are stored with fewer bits.



**Input:** Difference value  $d$  of a timestamp  $t_i$

**Output:** Bit pattern used to represent  $d$

**Return** one of the following bit patterns based on  $d$ :

**Bucket 1:** Control bits '00' if the difference value is zero i.e.  $d = 0$ .

**Bucket 2:** Control bits '01' if  $d \in [-511, 511]$ , followed by a signed bit that is '0' if the number is negative and '1' otherwise, and then the absolute value of  $d$  using 9 bits.

**Bucket 3:** Control bit '10' if  $d \in [-65535, 65535]$  followed by a signed bit and then absolute value of  $d$  (16 bits).

**Bucket 4:** Control bit '11' if  $d \in [-2147483647, 2147483647]$  followed by a signed bit and then the absolute value of  $d$  (31 bits).

**Listing 3.2:** Pseudo code for the bucket encoding

The bucket encoding used in IrregularDB is similar to the encoding used in the Gorilla timestamp model type discussed in **Appendix A**. The primary difference between IrregularDB's bucket encoding and Gorilla's encoding is that a constant amount of bits are used as control bits in IrregularDB. Only two control bits are used in IrregularDB, which results in a total of four different cases/buckets compared to Gorilla's five buckets. Using a constant amount of control bits uses less control bits in cases where the higher buckets are used more frequently. This is seen as an advantage since the Regular timestamp model type will likely be used in cases where bucket 1 is predominately used.

IrregularDB uses a different amount of buckets and a different amount of bits in each bucket because IrregularDB works with time in milliseconds, whereas Gorilla uses seconds. IrregularDB, therefore, needs more bits to store the same timestamps compared to Gorilla. IrregularDB, therefore, uses different bucket sizes than Gorilla. The size of each bucket is currently defined based on what is thought to be desirable. However, further testing is needed to determine the bucket sizes that give the best compression, which as discussed in **Section 7.2** is left as future work. The bucket sizes catch the following timestamp values:

- **Bucket 2:** the second bucket is used to store values up to around half a second as it can store absolute integer values up to  $511ms$  or about 0.5 seconds.
- **Bucket 3:** can contain values for just over a minute as this bucket can store absolute values up to  $65535ms$  or about 65 seconds.
- **Bucket 4:** is the catch-all bucket that handles the remaining values.

### Approximation

IrregularDB introduces approximation in the SIdiff model type by utilizing that smaller difference values use fewer bits due to the variable-length encoding. If a difference value  $d$  can be put in a smaller bucket, thus using fewer bits to store the value, and still be within the threshold value  $T_t$ , then approximation is applied on  $d$ .

How the approximation is done can be seen in **Listing 3.3**. A max values set MAX-VALUES is used. The set contains the maximum value that can be contained in each bucket from **Listing 3.2** (except the largest one as no values can be shrunk to the largest bucket). For the above-mentioned bucket sizes these are the following values:  $\text{MAX-VALUES} = \{0, 511, 65535\}$ .

```

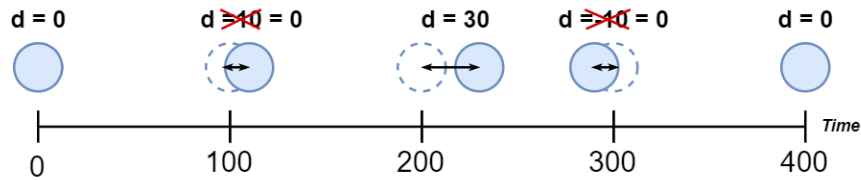
1 Input: Absolute value of the difference  $d$ , threshold  $T_t$ , and a set of max values MAX-VALUES
2 Output: Approximation of the absolute value of  $d$  within threshold  $T_t$ 
3
4 foreach  $max \in \text{MAX-VALUES}$  do:
5     if  $max \leq d$  and  $d \leq max + T_t$  then
6         return  $max$ 
7
8 return  $d$ 

```

**Listing 3.3:** Pseudocode for approximation of  $d$

In **Listing 3.3** the  $d$  value is compared with all max values to see if  $d$  is within the threshold range for any of them. If  $d$  is within the threshold range of one of the max values, then the current max value is returned as seen on line 6. If  $d$  could not be approximated using any of the max values within our threshold then the original  $d$  value is returned as seen on line 8.

The effect of the approximation can have is illustrated in **Figure 3.2**.



**Figure 3.2:** SIdiff approximation example with  $SI = 100$  and  $T_t = 15$

In the example shown in **Figure 3.2**, a threshold of  $T_t = 15$  is used. For the second timestamp  $t_2 = 110$  this means  $d = 0$  is stored instead of  $d = 10$  since for  $max = 0$  we have that  $0 \leq d$  and  $d \leq 0 + 15$  (from **Listing 3.3**). This approximation means that IrregularDB would store  $t_2$ 's difference using **Bucket 1** from **Listing 3.2**. This reduces the number of bits used to 2 bits instead of 12 bits if **Bucket 2** was used for  $t_2$ .

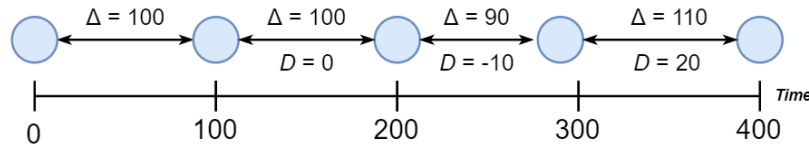
### 3.1.4 DeltaDelta timestamp model type

The DeltaDelta timestamp model type is heavily inspired by the Gorilla timestamp model type. A detailed description of the Gorilla timestamp model type can be found in **Appendix A**. This description of the Gorilla timestamp model type is placed in the appendix as it was produced in connection with previous work.

The idea behind the DeltaDelta timestamp model type is to store delta-of-delta values for the timestamps. Delta-of-delta timestamp values are likely smaller than raw timestamp values. A delta-of-delta value  $D$  is calculated for all timestamps except the first two in a sequence of timestamps  $\langle t_1, t_2, \dots, t_n \rangle$ . The delta-of-delta value  $D$  is calculated as shown in **Equation (3.4)**.

$$D = (t_i - t_{i-1}) - (t_{i-1} - t_{i-2}) \text{ for } 2 < i \leq n \quad (3.4)$$

**Figure 3.3** exemplifies delta-of-delta values.



**Figure 3.3:** Example of delta-of-delta values

As shown in **Figure 3.3** it is not possible to calculate a delta-of-delta value  $D$  for the first two timestamps  $t_1$  and  $t_2$  so special handling is done for them.  $t_1$  is stored as a base timestamp and for  $t_2$  a delta value is stored  $\Delta = t_2 - t_1$ . For the remaining timestamps, we store their delta-of-delta values  $D$ . The delta-of-delta values can be used to reconstruct the original timestamps by using  $t_1$ , the initial  $\Delta$  value, and previous delta-of-delta values.

The delta-of-delta values are compressed using a variable-length encoding similar to the bucket encoding used for SIdiff shown in **Listing 3.2**. The same approximation approach used for the SIdiff timestamp model type (**Listing 3.3**) is also used for the DeltaDelta timestamp model type to gain further compression.

## 3.2 Value model types

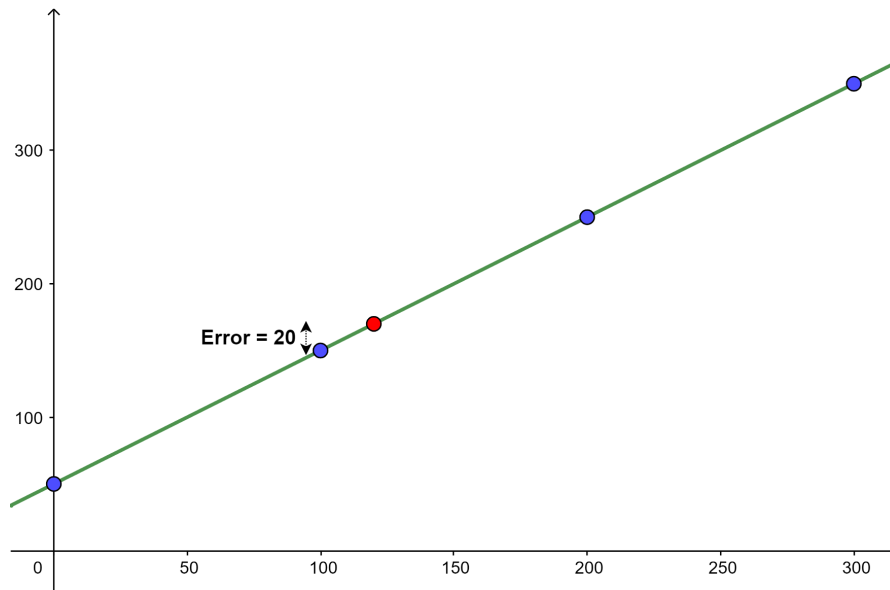
IrregularDB supports *multi-value-model-based compression* by using value model types to compress the values of data points. The value model types supported in IrregularDB are *PMC-mean*, *Swing*, and *Gorilla*. PMC-mean can construct VALUE MODELS that represent data using a constant mathematical function. The Swing value model type builds value models that represent data using linear mathematical functions. Gorilla is a lossless value model type that uses the XOR operator with the previous value to store a variable-length encoding of the values. A detailed description of the value model types used in IrregularDB can be found in **Appendix B**. This description is put in the appendix as it was produced in a previous project.

Floats are used instead of doubles to save additional space when compressing the values, as this reduction in storage usage was considered to be worth the precision loss. Using floats results in PMC-mean storing a float and Swing storing two floats for each value models. Thus, the value models use 4 and 8 bytes respectively instead of the 8 and 16 bytes needed if doubles were used. For Gorilla the initial value is a float and the number of bits used for leading zeroes (LZ) is reduced from 5  $\rightarrow$  4 bits and for the length (L) it is reduced from 6  $\rightarrow$  5 bits.

PMC-mean and Swing both guarantee that the values reconstructed using their value models are within a given error bound  $\varepsilon_v$  thereby upholding the condition defined in **Definition 2.3.8**. However, Swing does not provide this guarantee if the timestamps are approximated as will be explained in **Section 3.2.1**. Gorilla will always reconstruct the actual values without any error and will therefore always be within  $\varepsilon_v$ .

### 3.2.1 Threshold's effect on value model types

Having a threshold for timestamps introduces a problem when it comes to value model types such as Swing as its recreated values depend on the timestamp. For example consider the time series interval  $TSI = \langle (0ms, 50), (100ms, 150), (200ms, 250), (300ms, 350) \rangle$ , an error bound  $\varepsilon_v = 0\%$ , and a threshold  $T_t = 20$ . The Swing value model type could then be used to create the linear mathematical function  $m_v := f(t) = t + 50$  which represents  $TSI$ 's values. The problem arises by the second data point  $d_2 = (100ms, 150)$ . This data point could be approximated to be  $t_2^* = 120ms$  as this is within the threshold  $T_t$ . This timestamp approximation is what would be stored in IrregularDB and be used to reconstruct the values for  $TSI$ . Applying the linear function to  $t_2^* = 120ms$  would give  $v_2^* = 170$ , which is not within the error bound  $\varepsilon_v = 0\%$ . This is also illustrated in **Figure 3.4**. Where the red data point represents the approximation of  $d_2$  created using  $t_2^*$ .



**Figure 3.4:** Example of threshold affecting values

Using a threshold value above 0, therefore, means that the Swing value model type cannot guarantee that its reconstructed values will be within  $\varepsilon_v$ . In order to combat this IrregularDB by default disables Swing if a threshold above zero is provided.

However, it does not necessarily make sense to disable Swing in this example, as the data points follow a linear pattern as seen in **Figure 3.4**. The sensor would likely have measured the value 170 if the sensor would have measured at time 120. IrregularDB, therefore, allows the user to enable Swing when the threshold is not 0 with the user-configurable parameter 'model.value.error-bound.strict'. This parameter is described in more detail in **Appendix C**.

### 3.3 Fallback model types

Some of the model types supported in IrregularDB need a certain amount of data points to be able to create a model. For example, the Regular timestamp model type needs at least two data points to be able to calculate the sampling interval used in its `TIMESTAMP MODEL`. There can be cases where there are not enough data points left to create a model since the user can choose to disable model types used for ingesting data.

In order to handle this a Fallback value model type and a Fallback timestamp model type are added to IrregularDB. These two model types will be utilized if none of the used model types can handle the remaining data points. The Fallback model types work by creating models containing the raw timestamp and value of a data point.

### 3.4 Segment generation

This section will discuss the approach used in IrregularDB to create the timestamp model and value model that are stored in a segment in order to represent a time series interval. **Section 3.4.1** discusses the user-configurable length bound and the max segment length. **Section 3.4.2** then describes how the models are created while **Section 3.4.3** describes how models are selected. Then **Section 3.4.4** gives a small example to illustrate how segment generation works.

#### 3.4.1 Length bound

IrregularDB introduces a user-configurable length bound for its model types which is necessary because some of the model types can represent an arbitrary number of data points. For example, the Gorilla value model type can consume an infinite amount of values and never exceed the error bound as it reconstructs precise values. The supported model types that need to adhere to the length bound are: DeltaDelta, SIdiff, and Gorilla.

The value chosen for the length bound affects both the compression ratio, query speed, and ingestion speed of the system. The query speed is affected by the length bound because entire segments need to be decompressed, even when only querying a single data point. If the length bound is large then it takes longer to answer queries as larger segments need to be decompressed. The advantage of having a large length bound is that fewer segments need to be stored, which generates less overhead. The length bound is therefore a trade-off parameter between compression ratio and query performance, and it can be configured by the user based on what they want. Length bound can also affect the ingestion time of the system because having a larger length bound can lead to faster ingestion as fewer segments have to be generated and sent to the database for storage.

The test described in **Appendix D.1** was done in order to determine a default value for the length bound. The result of this test is that a default length bound of  $L = 400$  is chosen as increasing the length bound beyond this value provided very little increase in ingestion speed and compression ratio.

## Max segment length

Some model types are not limited by a length bound (Regular, PMC-mean, and Swing). These model types are not bound by length bound since they are bound by either error bound or threshold instead. Not being bound by a length bound means that their models can grow very large resulting in long query times. Furthermore, model types that are not limited by the length bound can create indefinitely large models if the data follow the right pattern. This can lead to out-of-memory exceptions as observed during the evaluation of the TSBS dataset. A max segment length is therefore used which is a user-configurable parameter named ‘segment.max\_length’ (**Appendix C**). The max segment length applies to all model types.

### 3.4.2 Creating the models

This section describes the approach used in IrregularDB to create models from its supported model types. **Listing 3.4** shows the approach used to create value models from the value model types. A similar approach is used to create timestamp models.

```

1  Input: A length bound  $L$ , a maximum segment length  $SEG_{MAX-L}$ , an error bound  $\varepsilon_v$ , and a
    stream of data point values  $S_V = \langle v_1, v_2, \dots \rangle$ .
2  Output: List of values models finishedValueModels
3
4  activeValueModelTypes  $\leftarrow$  {PMCMean, Swing, Gorilla}
5  finishedValueModels  $\leftarrow$  {}
6  amtIngestedValues  $\leftarrow$  0
7
8  while activeValueModels.notEmpty() do:
9      nextValue  $\leftarrow$  S_V.next()
10     // Try and append new value to still active value model types
11     foreach  $M_V \in$  activeValueModelTypes do:
12         appendSuccessful  $\leftarrow$   $M_V.appendValue(nextValue, \varepsilon_v)$ 
13         if not appendSuccessful then:
14             activeValueModelTypes  $\leftarrow$  activeValueModelTypes  $- M_V$ 
15             finishedValueModels  $\leftarrow$  finishedValueModels  $\cup M_V.getValueModel()$ 
16
17     amtIngestedValues  $\leftarrow$  amtIngestedValues + 1
18
19     if amtIngestedValues =  $L$  then:
20         // Length bound, L, reached finish GORILLA
21         activeValueModelTypes  $\leftarrow$  activeValueModelTypes  $-$  Gorilla
22         finishedValueModels  $\leftarrow$  finishedValueModels  $\cup$  Gorilla.getValueModel()
23
24     if amtIngestedValues =  $SEG_{MAX-L}$  then:
25         // Maximum segment length reached finish remaining value model types
26         foreach  $M_V \in$  activeValueModelTypes do:
27             activeValueModelTypes  $\leftarrow$  activeValueModelTypes  $- M_V$ 
28             finishedValueModels  $\leftarrow$  finishedValueModels  $\cup M_V.getValueModel()$ 
29
30 return FinishedValueModels

```

**Listing 3.4:** Value model creation approach

The idea used in **Listing 3.4** is to first pull a new value from the stream as seen on line 9 (waiting until a value arrives if the stream is empty). This value is appended to each of the active value model types as seen on line 12. Then, on lines 13-15 the value model types that could not fit the current value within the error bound  $\varepsilon_v$  are removed from the list of active value model types and their value model is moved to the list of finished value models. Then, on lines 19-22 the value model type that adheres to the length bound, i.e. Gorilla, is finished if the length bound has been reached. An additional check is performed to test if the max length for segments has been reached on line 24. If the max segment length has been reached all the remaining active value model types are finished on lines 26-28. The loop continues until all of the value model types are finished as seen on line 8. Finally, the list of finished value models is returned on line 30.

### 3.4.3 Model selection

The result of the approach discussed in **Section 3.4.2** is two lists: *FinishedTimeStampModels* and *FinishedValueModels*. Since IrregularDB only uses one timestamp model and one value model for a segment then IrregularDB needs to select one model from each list. IrregularDB selects models based on their achieved bytes per data point.

The models stored in the finished lists have different lengths as the models fit data points differently. The value model and timestamp model for a segment must have the same number of data points. The problem is then how to select a value model and a timestamp model for a segment and ensure they have the same length. Consider the following two ways to handle this problem:

- **Brute force:** For each timestamp model in *FinishedTimeStampModels* combine it with each value model in *FinishedValueModels* to get a set  $S$  of all possible model combinations. Then, for each combination-pair  $P$  from  $S$  do the following:
  - Reduce the longest model in  $P$  to be the same length as the shorter model.
  - Then calculate  $P$ 's number of bytes per data point as the storage usage of both models in  $P$  divided by the number of data points the models represent.

Then select the pair that has the lowest amount of bytes per data point to be used in the segment.

- **Greedy:** Calculate the amount of bytes per data point of the timestamp models and value models separately. Then select the timestamp model and value models with the lowest amount of bytes per data point to be used in the segment. Then reduce the longer model to have the same amount of data points as the shorter model.

IrregularDB supports both methods and allows the user to choose which of the model selection methods to use. This is done by utilizing the user-configurable parameter `'model.picker'` described in **Appendix C**.

A segment can then be created using the timestamp model and value model that were selected by one of the selection methods above. The segment also contains a start and end time that are derived from the selected models. The original data points cannot be used to determine the start time and end time because the data points are approximated using a threshold. Therefore, the timestamp model are decompressed to determine the correct timestamps for the start time and end time that should be stored in the segment.

### 3.4.4 Example of segment generation

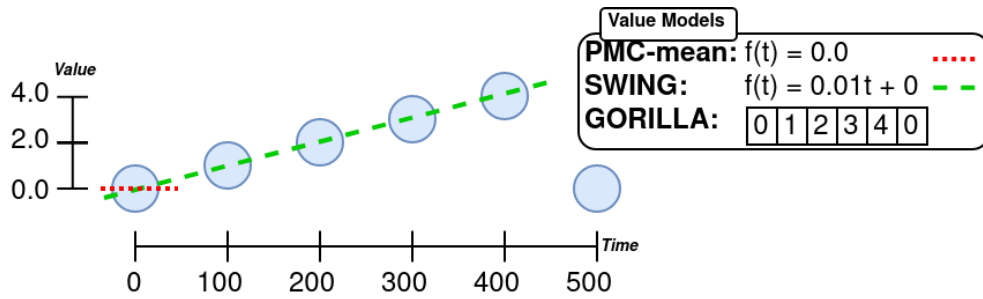
This section gives an example of how segment generation works. An error bound  $\varepsilon_v = 0\%$ , a threshold  $T_T = 0$ , a length bound  $L = 6$ , and the time series interval:

$$TSI = \langle (0ms, 0.0), (100ms, 1.0), (200ms, 2.0), (300ms, 3.0), (400ms, 4.0), (500ms, 0.0) \rangle$$

is used for this example. The example will primarily be focused on the creation and selection of value models to keep the example concise.

#### Creating the models example

The first step to generate a segment is to create timestamp models and value models using the approach discussed in **Section 3.4.2**. **Figure 3.5** illustrates the three value models created using the supported value model types in IrregularDB for the example time series interval.



**Figure 3.5:** Example of value models created in IrregularDB with error bound  $\varepsilon_v = 0\%$  and length bound  $L = 6$  for a TSI

Because the error bound is  $\varepsilon_v = 0\%$  the PMC-mean can only fit the first data point since the second data point's value is 1.0 and a constant function cannot represent both 0.0 and 1.0 with  $\varepsilon_v = 0\%$ . PMC-mean is therefore removed from the *ActiveValueModelTypes* list and its constant function  $m_{vPMCmean} := f(t) = 0.0$  is added to the *FinishedValueModels* list.

Swing can still represent the value at  $v_2$  as it can create a linear function value model going through 0.0 and 1.0. Swing can also append the next three values, however, the last value cannot be fitted to the linear function. Swing is, therefore, removed from the *ActiveValueModelTypes* list at value  $v_6$  and its current linear function  $m_{vSwing} := f(t) = 0.01t + 0$  is added to the *FinishedValueModels* list.

Gorilla can fit all the values as it will continue to fit data points up to the defined length bound of 6. Furthermore, the values are exact since the Gorilla is lossless. Notice that the model shown in **Figure 3.5** for the Gorilla is simplified. In practice, a sequence of bytes representing the original values is stored. The Gorilla model finishes after the sixth data point when the length bound of  $L = 6$  is reached and the model is then added to the *FinishedValueModels* list.

#### Model selection example

The greedy model selection approach is used to keep the model selection simple as it means the selection of value models is done separately from the selection of timestamp models. The



following amounts of bytes per data point are calculated to select a value model for the created value models in the example:

- **PMC-mean:**  $4.0 \frac{\text{bytes}}{\text{data point}}$ 
  - **Storage usage:** PMC-mean's constant function is stored using a float giving a total of 4 bytes.
  - **Amount of data points:** can only represent 1 data point.
- **Swing:**  $1.6 \frac{\text{bytes}}{\text{data point}}$ 
  - **Storage usage:** Swing's linear function is stored using two floats giving a total of 8 bytes.
  - **Amount of data points:** Can represent 5 of the 6 data points.
- **Gorilla:**  $2.5 \frac{\text{bytes}}{\text{data point}}$ 
  - **Storage usage:** The base value and the following XORs are stored in 15 bytes.
  - **Amount of data points:** All 6 data points.

The selected value model would be the Swing value model as it uses the least amount of bytes per data point. The calculations here are simplified as they do not consider the storage usage of the additional/overhead information added when storing a segment. An example of overhead could be the start and end times of a segment.

Similar model creation and selection would be done for the timestamp model types. However, in this example, it is clear to see that the timestamps follow a regular pattern, which means that the Regular timestamp model type is best suited for this example. The greedy approach would therefore select the Regular timestamp model.

The Regular timestamp model would have to be reduced in length as the final step of the greedy model selection since the Swing value model is only able to represent five of the data points in this example. A segment is then created using the selected models.

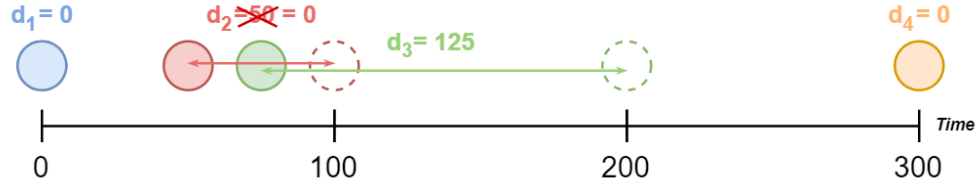
### 3.5 Keeping the order of the data points

Intuitively it makes sense to keep the data points in a time series ordered after time, meaning that each data point's timestamp is larger than the previous data point's timestamp. That is the reason for the condition  $t_i < t_{i+1}$  is included in the definition for time series found in **Definition 2.3.2**. However, during the integration tests discussed in **Section 5.2**, it was discovered that IrregularDB did not uphold this condition.

Approximations of data points' timestamps can lead to some data points being out of order when a threshold above zero is used. This problem can occur both within a timestamp model of a single segment or across multiple segments. The following sections will discuss these problems and the solutions chosen to handle them.

### 3.5.1 Within model approximation problem

The approximation used for the bucket encoding used in the SIdiff and DeltaDelta timestamp model types lead to a problem of timestamps overtaking each other within a single timestamp model. To exemplify this consider the timestamp sequence  $T = \langle 0, 50, 75, 300 \rangle$ , which is ingested using the SIdiff timestamp model type and a threshold  $T_t = 50$ . This leads to the approximations shown in **Figure 3.6** as SIdiff creates a timestamp model with  $SI = 100$ .



**Figure 3.6:** Example illustrating why additional conditions are needed when approximating with a threshold larger than 0

In this example, the difference value  $d_2$  is within the threshold  $T_t = 50$ . The difference value  $d_2$  is, therefore, reduced to  $d_2 = 0$  approximating the second data point's timestamp to 100. However,  $d_3$  is not approximated because of  $d_3 > T_t$ . The actual difference value is stored for  $d_3$ . This results in the second data point having timestamp 100 and the third data point timestamp 75 when reconstructing the data points from the SIdiff. The full list of reconstructed data points is:  $\langle 0, 100, 75, 300 \rangle$ . Each of the reconstructed timestamps is within the threshold  $T_t = 50$  but out of order. A similar example could be created for the DeltaDelta timestamp model type.

#### Solution

The solution is to add an additional condition for when approximations are allowed for SIdiff and DeltaDelta timestamp model types. The condition is the following: “*Approximations of the difference/delta-of-delta values are only allowed as long as the approximated timestamp created using them is between the previous and next timestamp in the timestamp model*”. It is possible to check this condition for each timestamp except the first and last timestamp in a model because SIdiff and DeltaDelta first create their timestamp models once the length bound discussed in **Section 3.4.1** has been reached.

The Regular timestamp model type is not affected by the within model approximation problem. Recall the reconstruction of approximated timestamps formula from **Section 3.1.2**:

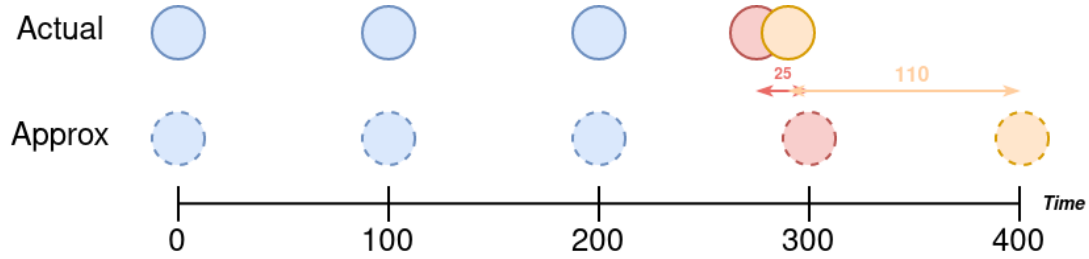
$$t_i^* = t_s + SI \cdot (i - 1) \text{ for } i \in [s, \dots, e]$$

Ordering of timestamps for the Regular timestamp model is guaranteed because only positive SI values are used and  $i$  is increasing in size for each successive reconstructed data point.

### 3.5.2 Overlapping segments problem

Timestamp model types do not have knowledge about data points outside their current timestamp model. This leads to a problem where the last data point in a segment can have an approximation of its timestamp that comes after the first data point in the next segment.

**Figure 3.7** illustrates the problem. To keep the example simple only the Regular timestamp model type is used to ingest the timestamp sequence  $T = \langle 0, 100, 200, 275, 290 \rangle$  with a threshold  $T_t = 25$ .



**Figure 3.7:** Example where the end time of a segment comes after the start time of the next segment with threshold  $T_t = 25$

The Regular timestamp model type can fit the first four data points with the initial sampling interval  $SI = 100$ . This is because for the fourth timestamp  $t_4$  the reconstructed timestamp would be  $t_4^* = 300$  which is within the threshold.

The fifth timestamp  $t_5 = 290$  cannot be fitted to the current model as its approximation  $t_5^* = 400$  is outside the threshold. The Regular timestamp model type, therefore, tries to find a new sampling interval that can fit all the timestamps:  $SI_{candidate} = \frac{290}{5-1} = 72.5$ . This candidate  $SI$  can not fit all the data points. A segment would therefore be created to represent the four first timestamps using a timestamp model with the initial sampling interval of  $SI = 100$ . The last timestamp created using this model would be  $t_4^* = 300$ .

The problem is now that the next segment would start with the timestamp  $t_5 = 290$ , which is before the previous timestamp, thereby breaking the condition that each timestamp must be larger than the previous.

#### Solution

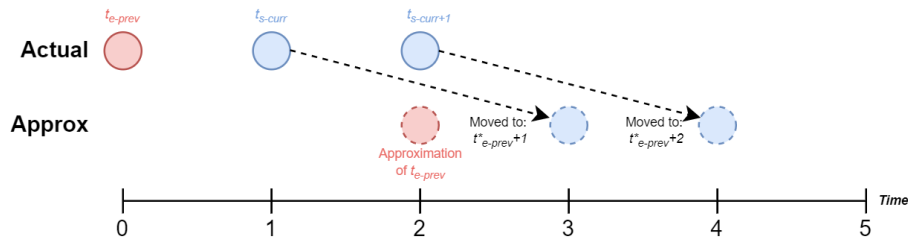
The chosen solution for the overlapping segments problem is to move the start time,  $t_{s-curr}$ , of the following segment to be after the approximation of the end time,  $t_{e-prev}^*$ , of the previous segment. This is done by updating the timestamp of the first data point in the current segment to be  $t_{s-curr} \leftarrow t_{e-prev}^* + 1$ . In the above example, this would mean that  $t_5$  would be moved to  $t_5^* = t_4^* + 1 = 301$ .

The reassignment of  $t_{s-curr}$  should be within any given threshold  $T_t$  since none of the timestamp model types used in IrregularDB tries to approximate the start time of a segment.

### Problem with the chosen solution

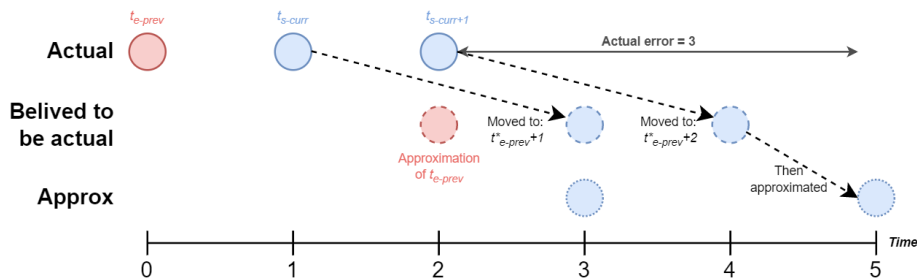
The chosen solution used to handle overlapping segments can in certain cases lead to IrregularDB breaking the threshold if a threshold higher than the minimum difference between two timestamps is used. The reason for this is that this solution can then require that, in addition to updating  $t_{s-curr}$ , its successor  $t_{s-curr+1}$  should be moved to  $t_{e-prev}^* + 1$ . This occurs if the moving of  $t_{s-curr}$  makes it surpass its successor  $t_{s-curr+1}$ .

**Figure 3.8** illustrates this moving of multiple timestamps. In this example a threshold of  $T_t = 2ms$  is used. The approximation  $t_{e-prev}^*$  is within the threshold and the initial moving of both  $t_{s-curr}$  and  $t_{s-curr+1}$  is still within  $T_t$ .



**Figure 3.8:** Example of moving multiple timestamps

This moving of multiple timestamps can lead to IrregularDB breaking the threshold  $T_t$  in certain edge cases. The problem is then that the timestamp model types have no knowledge about this moving of  $t_{s-curr}$  and  $t_{s-curr+1}$ . Since none of the supported timestamp model types approximate the first timestamp in a segment there it is not a problem that  $t_{s-curr}$  is moved. However, since the timestamp model types believe the actual value for  $t_{s-curr+1}$  to be 4, then the timestamp model types believe that they can approximate  $t_{s-curr+1}$  to  $t_{s-curr+1}^* = 5$ . This would break the threshold value of  $T_t$  as  $t_{s-curr+1}^*$  is 3 away from the original value as illustrated in **Figure 3.9**.



**Figure 3.9:** Example of how moving of timestamps can lead to breaking the threshold

Implementing a solution to this problem was left as future work, as discussed in more detail in **Section 7.6**. It was left as future work since the problem cannot occur in the setups used to test/evaluate IrregularDB. The reason for this is that in order for the problem to occur the irregular data has to be ingested with a threshold higher than the minimum difference between two timestamps, which is never the case during the integration test and evaluation of IrregularDB.

## 4 | Design

The design chapter is structured into five main parts. **Section 4.1** will contain an overview of the design of the full system. **Section 4.2** contains a more in-depth description of the system components that are responsible for data receiving. **Section 4.3** will focus on the parts of the system that are directly related to *model-based compression*. An overview of the core components of the systems is then presented in **Section 4.4**. Then the design of the database and querying of the system will be described in **Section 4.5**.

### 4.1 Architecture overview

This section will provide a simplified broad design overview. Later sections will introduce the concrete detailed diagrams of the system.

Before presenting the design of *IrregularDB* an overall design choice is discussed. This design choice is that *IrregularDB* is designed as a single node system. There are several reasons for this, the first of which is that the scope of this project is limited. It would take significant time to set up and get acquainted with multi-node programming frameworks, which would leave less development time for the focus of this project which is developing a system supporting our newly defined Multi-Timestamp Multi-Value Model Compression (MTVMC) approach. Another reason for developing a single node system is that several other time series databases only provide the single node version of the system for free. Therefore, the systems that *IrregularDB* will be compared to in our evaluation in **Chapter 6** will also be single-node systems.

#### 4.1.1 Initial Idea

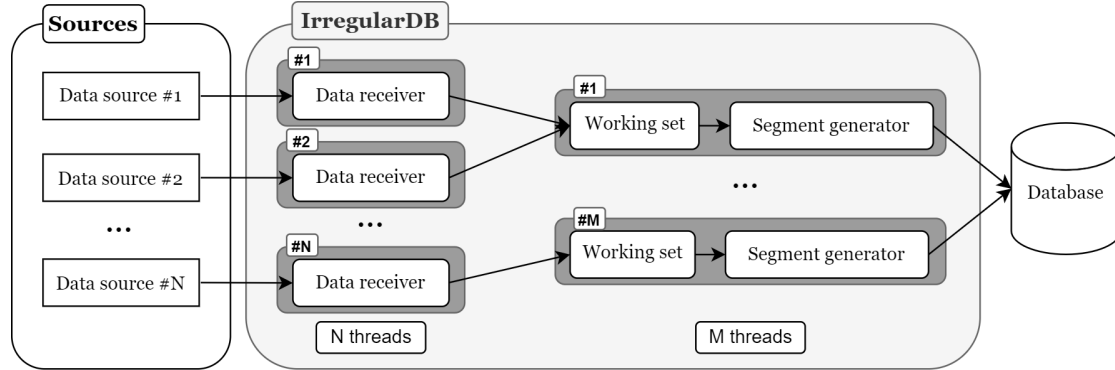
An initial idea for the structure of the system was to simply perform the model-based compression of the data as soon as it was received and on the same thread. However, compression can be quite a heavy process. Therefore, if this approach was used it would mean that while the thread is busy performing the model-based compression it will be unavailable for receiving data. Therefore, to keep the data receiving responsive for the clients that are sending data, it was decided that the receiving of the data should be separated from the model-based compression.

#### 4.1.2 Improved idea

To achieve a separation between data receiving and model-based compression the concept of working sets is introduced in *IrregularDB*. The idea behind the working sets is that each working set has a thread-safe buffer. Having this buffer allows the receiver to always receive new data points, as it only has to deliver the data to the buffer. Each working set will then, in its own thread, dequeue elements from the buffer and perform model-based compression.

**Figure 4.1** shows a general overview of the system based on the idea of utilizing working sets. The lines in this figure denote data flow. As indicated in the figure there exist several instances

for some of the components that are running in separate threads. Namely, the data received from each data source is handled in its own thread. Each working set also has its own thread. As seen in **Appendix C** the amount of working sets in the system is user-configurable.



**Figure 4.1:** Component overview diagram

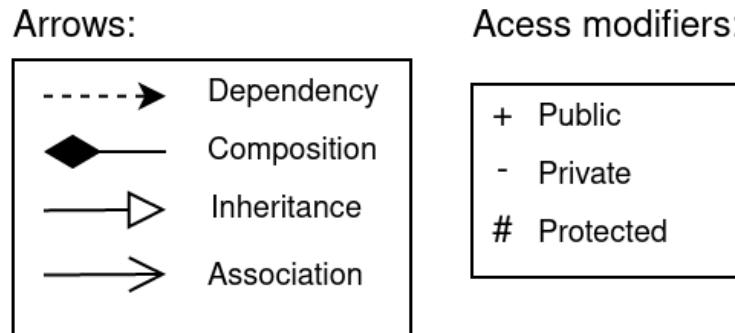
The separation between data receiving and model-based compression leads to a natural split of the system into two main parts. A quick description of each part is given here:

- **Data receiving:** First, some data source outside IrregularDB send their data to IrregularDB. Each new data source is handled by IrregularDB spawning a new data receiver in its own thread. The data receivers are responsible for transforming the data from the data sources into a format that the rest of IrregularDB understands. Each data receiver is then assigned to a working set using a partitioning component. The data receivers can then push their data directly to the buffer of their assigned working set.
- **Model-based compression:** Each working set will then process the received DATA POINTS by extracting them from its buffer and then transferring them to a segment generator in a FIFO manner. The segment generator component contains logic for performing the model-based compression, as well as the segment generation discussed in **Section 3.4**. Lastly, the generated SEGMENTS are sent from the segment generator to the database for storage.

The two halves of the system will be further elaborated on in **Sections 4.2 and 4.3**. In these sections whenever we reference a class then it will be highlighted as demonstrated here: `WorkingSet`.

### 4.1.3 Class diagram standards

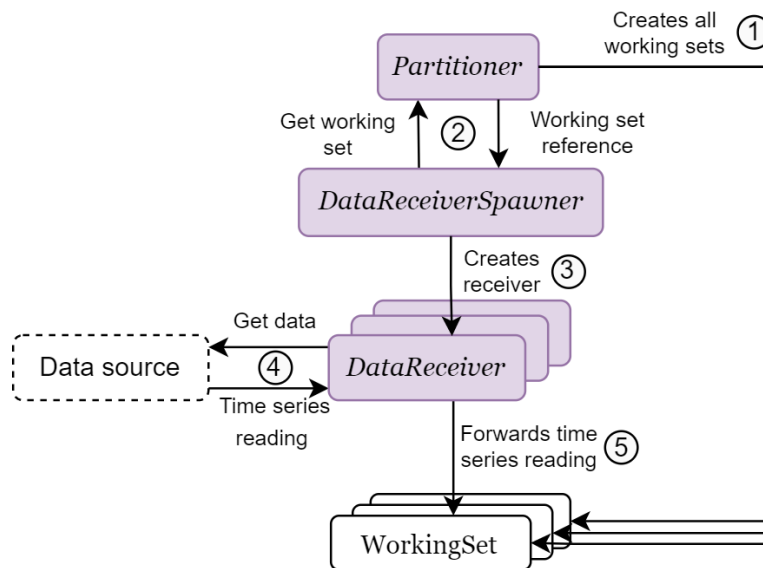
For the class diagrams shown in **Sections 4.2 and 4.3**, the UML class diagram standards will be used. For quick reference, **Figure 4.2** is provided to show the meaning of the arrows and access modifiers used. Note that these definitions only apply to figures described as class diagrams.



**Figure 4.2:** Meaning of arrows and access modifiers from UML class diagram standard

## 4.2 Data receiving

The first part of the system is responsible for receiving the data from outside the system, which will be described in this section. A flow diagram has been created as seen in **Figure 4.3** to further specify the flow of the first half of the system. The purple color indicates abstract classes whose implementation classes are described in ensuing sections.



**Figure 4.3:** Flow diagram of data receiving and partitioning part of the system

First, the `Partitioner` creates all `WorkingSets` in step 1. Then, in step 2, the `DataReceiverSpawner` gets a reference to a `WorkingSet`. Then in step 3 a `DataReceiver` is created by the `DataReceiverSpawner`. This `DataReceiver` is assigned to the `WorkingSet` retrieved in step 2. Each `DataReceiver` receives time series readings in step 4 from a single data source and forwards the time series readings to their assigned `WorkingSet` in step 5.

The `DataReceiverSpawner` is needed since socket connections are initiated by a server socket.

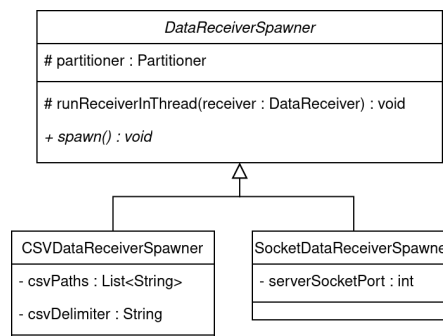
The `DataReceiverSpawner`'s purpose is to spawn a new `DataReceiver` every time a new client connects to the server socket.

Multi-threading is used in IrregularDB to increase performance. This is among other things achieved by having multiple `WorkingSets` that each run on their own thread in parallel. To achieve this the data needs to be partitioned between the `WorkingSets` which is done using the `Partitioner`.

This in broad terms covers the data receiving and partitioning part of the system. The following sections give an in-depth explanation of each class.

### 4.2.1 Data receiver spawner

As mentioned earlier the `DataReceiverSpawner` is responsible for spawning `DataReceivers`. Because of this, there must exist a specific `DataReceiverSpawner` for each type of `DataReceiver`. In **Figure 4.4** a slightly simplified class diagram can be seen. The abstract superclass `DataReceiverSpawner` holds the common parts of each specific implementation and there are implementations for both CSV and Socket data sources. One thing to highlight is how the superclass has the protected method `runReceiverInThread()`, which takes a newly created `DataReceiver` and runs it in a new thread. This approach avoids having to implement this functionality twice.

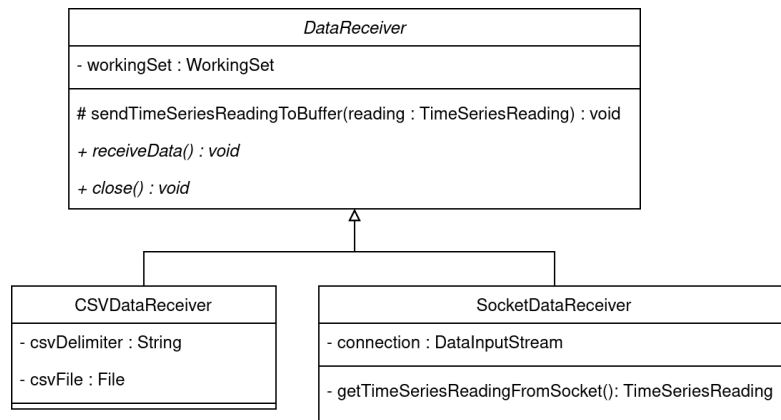


**Figure 4.4:** Simplified class diagram of `DataReceiverSpawner`

### 4.2.2 Data receiver

The `DataReceiver`'s purpose is to handle the data from a `DataSource` and then transfer it to its assigned `WorkingSet`. In **Figure 4.5** both the super and sub classes can be seen with some of the important properties and methods. Something to highlight here is the abstract method `receiveData()`. This is the method that is executed in a separate thread by the `DataReceiverSpawner`.





**Figure 4.5:** Simplified class diagram of *DataReceiver*

As reading data from a CSV file is relatively trivial this will not be described. However, the socket communication has nontrivial parts. When communicating using a socket connection the sender and *DataReceiver* must agree on how data are sent. This is because the *DataReceiver* must know how many bytes to read to get a *TIME SERIES READING* as well as how to interpret these bytes. To achieve this some protocol must be used. Since time series readings are transferred over the socket connection then the following needs to be transferred:

$\langle \textit{TimeSeriesTag} : \textit{String}, \textit{Timestamp} : \textit{Long}, \textit{Value} : \textit{Float} \rangle$

For writing the timestamp and the value there are no major challenges as these have a fixed byte size. The timestamp is an 8-byte integer (*Long*) since epoch time is used for timestamps as mentioned in **Section 2.3**. The value for the time series reading is simply a float. The time series tag is problematic as it is a string, meaning that the number of bytes used depends on both the length of the string as well as the encoding used for the string.

A string encoding that both sender and receiver agree on must be selected. It was chosen to use UTF-8 in IrregularDB as it is a commonly used encoding. Next, for handling the variable length of the string, an integer is sent on the socket connection before the string. The integer specifies the number of bytes used for the encoded string.

For the transport protocol just described the sending of the string will take up the majority of the data. One solution to combat this is to only send the tag with the first time series reading, however, this would limit a connection to only a single time series. This would mean that if some monitoring unit has say 5 different sensors it would have to make 5 connections to IrregularDB, which is excessive. Therefore, the transport protocol is further updated to first send a byte where the value 1 indicates that a string will be included in the message. Reversely, the value 0 indicates that the string from the last message should be used. This has the potential to save a lot of bandwidth while maintaining the ability to send data for several time series through one connection.

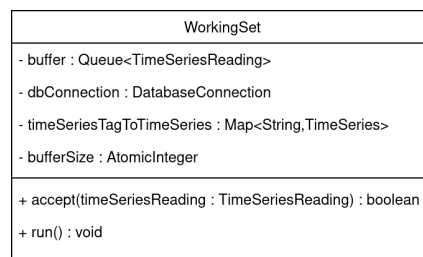
### 4.2.3 Partitioner

The responsibility of the `Partitioner` is to assign each of the `DataReceivers` to a `WorkingSet`. An optimal `Partitioner` would perform load balancing such that each `WorkingSet` gets an almost equal amount of data points per time unit. In practice, it is hard to implement a `Partitioner` that load balances perfectly since it is uncertain how many data points each `DataReceiver` will produce.

Instead, a simpler solution was chosen in `IrregularDB` by utilizing a round-robin approach where `DataReceivers` are distributed evenly across `WorkingSets`. This means that each receiver is assigned incrementally to one of the allocated `WorkingSets`. The load balancing performed by the `Partitioner` is, therefore, in the number of `DataReceivers` per working set instead of the optimal approach where `DataReceivers` are distributed such that each working set receives the same number of data points per time unit.

### 4.2.4 Working set

The `WorkingSet` is the workhorse of the system as it is responsible for receiving data from multiple `DataReceivers` and then performing the model-based compression on the received data. In **Figure 4.6** the `WorkingSet`'s two public methods can be seen.



**Figure 4.6:** Simplified class diagram of the `WorkingSet`

The *accept()* method is a method that takes a time series reading and places it in the thread-safe buffer. The *run* method starts an infinite loop in which it will attempt to dequeue a time series reading from the buffer. If the dequeue operation succeeds it will process the time series reading by feeding the time series reading to the `TimeSeries` it belongs to. This `TimeSeries` can be identified using the private field *timeSeriesTagToTimeSeries* which, as its name states, is a map from a time series tag to a `TimeSeries` instance. If no instance exists a new `TimeSeries` instance is created for the given tag. More details about the `TimeSeries` class, which is an abstraction used in the model-based compression, are found in **Section 4.3.1**.

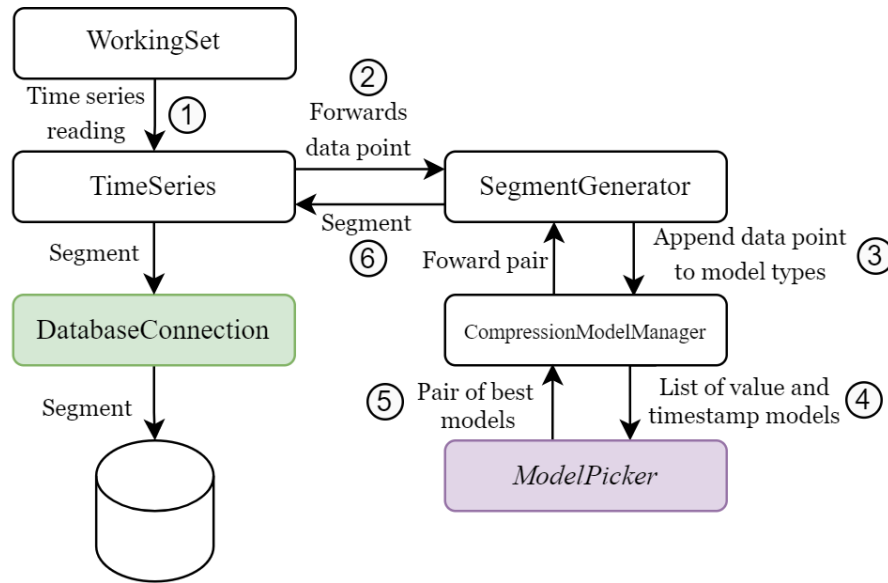
### Throttling data receiving

Consider the case where the data receiving part of the system is faster than the model-based compression then the `DataReceivers` would just keep on adding data points to the `WorkingSets`' *buffers* potentially causing a memory overflow. In order to combat this, the *bufferSize* instance variable is used to ensure that the `DataReceivers` are throttled when the *buffer* reaches a certain amount of data points.

The user can configure this amount of data points with the ‘`workingset.max_buffer_size_before_throttle`’ configuration parameter. When the defined amount of data points is reached the `accept()` method will return false (but still insert the data point in the buffer) signaling to the `DataReceiver` to sleep for a duration before receiving data again. This duration is user-configurable with the ‘`receiver.throttle_sleep_time`’ parameter.

### 4.3 Model-based Compression

This section will cover the right half of the diagram from **Figure 4.1** in the architecture overview section, which is the compression part of IrregularDB. The `DatabaseConnection` will also be briefly described, however, the database design itself is left for **Section 4.5**. A flow diagram for the compression part of the system can be seen in **Figure 4.7**.

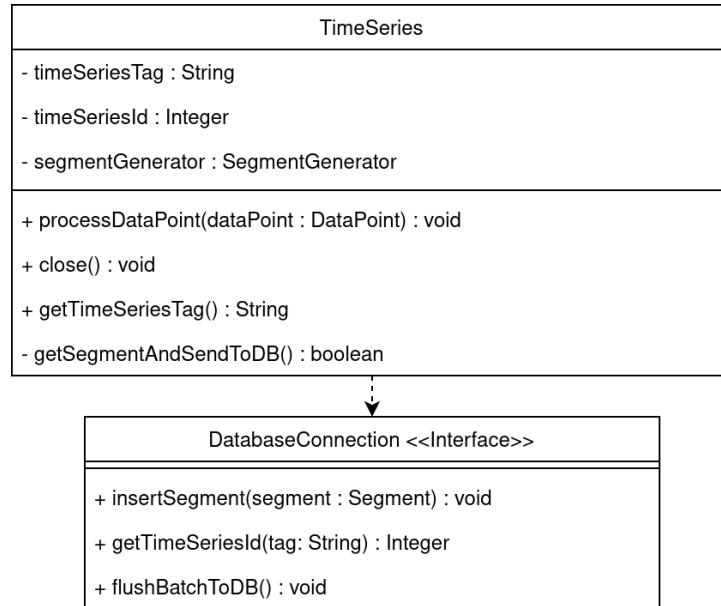


**Figure 4.7:** Flow diagram of compression part of the system

The `WorkingSet` is responsible for sending time series readings from its local buffer to the correct `TimeSeries` instance in step 1. There are as many instances of the `TimeSeries` class as the number of time series the system processes data for. The `TimeSeries` class has the responsibility of forwarding the data points from the time series readings to the `SegmentGenerator` in step 2. The `SegmentGenerator` is a core component of the model-based compression part of IrregularDB and it is responsible for generating the segments that compress the measured data points. The `SegmentGenerator` utilizes a `CompressionModelManager` and a `ModelPicker` that help in the construction of segments in step 3 and 4. In step 5 the best pair of timestamp model and value model is sent back to the `SegmentGenerator`. The `SegmentGenerator` then in step 6 creates a segment from the models and additional information (e.g. start time and end time) and sends the segment back to the `TimeSeries` instance that will write the segment to the database. The following sections describe the individual classes in more detail.

### 4.3.1 Time Series

The `TimeSeries` class is responsible for starting segment generation and communicating with the `DatabaseConnection` interface. The class diagram for the `TimeSeries` can be seen in **Figure 4.8**.

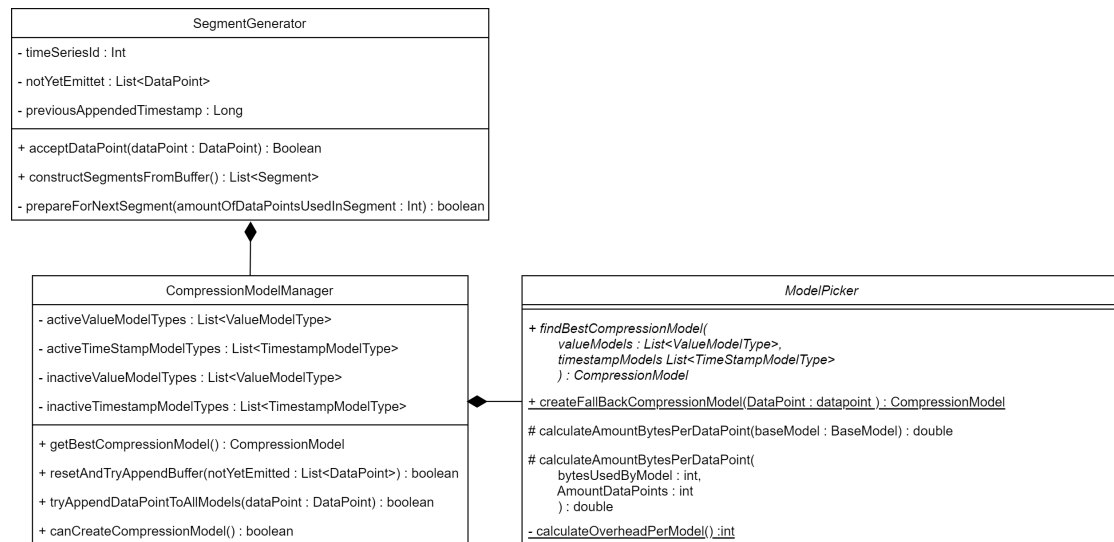


**Figure 4.8:** Simplified class diagram of `TimeSeries` & `DatabaseConnection`

The `TimeSeries`'s *processDataPoint()* method will forward a data point to a `SegmentGenerator` instance. If the `SegmentGenerator`'s segment is finished the `SegmentGenerator` will signal this to the `TimeSeries`. The `TimeSeries` will then ask the `SegmentGenerator` for the finished segment and pass that segment to the `DatabaseConnection` through the *getSegmentAndSendToDb()* method.

### 4.3.2 Segment Generator

The `SegmentGenerator` class is responsible for creating segments. **Figure 4.9** is a class diagram of the `SegmentGenerator` class and its dependencies. The `SegmentGenerator` class uses two helper classes: `CompressionModelManager` and `ModelPicker`. The `ModelPicker` functionality is put into its own class to make it easier to support different model selection approaches in `IrregularDB` (currently a greedy and brute-force approach is available).



**Figure 4.9:** Simplified class diagram of **SegmentGenerator** and **CompressionModelManager** which make use of a **ModelPicker**.

The **SegmentGenerator** has an `acceptDataPoint()` method. This method first applies the solution to the overlapping segments problem, described in **Section 3.5.2**. The main idea behind this solution is to move the timestamp of the current data point if it is before the end time of the previous segment. The end time of the previous segment is stored in the variable `previousAppendedTimestamp`.

The data point is then added to `notYetEmitted` buffer. Then the `tryAppendDataPointToAllModels()` method is invoked on the **CompressionModelManager**. The `tryAppendDataPointToAllModels()` is used to construct timestamp- and value models in a manner similar to the approach described in **Section 3.4.2**. The `tryAppendDataPointToAllModels()` method returns false when either all the VALUE MODEL TYPES or all the TIMESTAMP MODEL TYPES could no longer append the given data point, which indicates that they are ready to construct their models.

Once the `tryAppendDataPointToAllModels()` method returns false then the **SegmentGenerator**'s `constructSegmentsFromBuffer()` method is invoked. This methods is split into three parts:

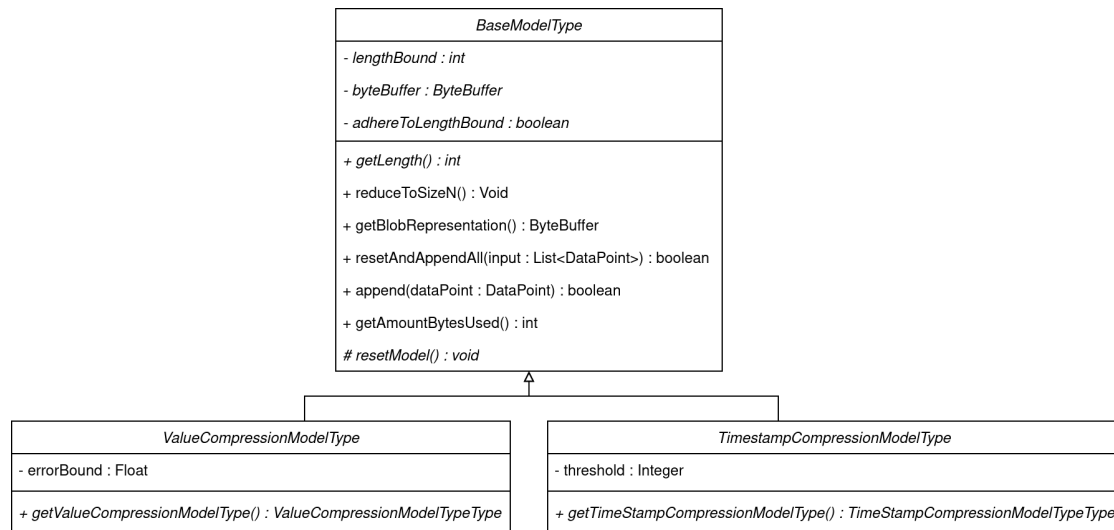
1. **Model selection:** This method first tries to select a timestamp and value model using one of the model selection approaches described in **Section 3.4.3**. This is done by calling the `getBestCompressionModel()` method on the **CompressionModelManager** class, which in turn calls the `findBestCompressionModel()` method on the **ModelPicker** class. This selection method uses bytes per data point as selection criteria. When calculating bytes per data point the method considers both the bytes used for the models themselves as well as additional segment information. Additional segment information is required when storing segments in the database since a segment for example contains a start time and an end time. The summary information, discussed in **Section 4.5.2**, if enabled is also part of this additional segment information. The selected timestamp- and value model are returned as a compression model object, which is an object that wraps the timestamp- and value model.

2. **Generating the segment:** The `SegmentGenerator` then uses the selected models to generate a segment object that can be stored in the database and update the *previousAppendedTimestamp* based on the end time of the generated segment.
3. **Preparing for the next segment:** Finally, the `SegmentGenerator` removes the  $N$  earliest data points from the *notYetEmitted* buffer, where  $N$  is the length of the selected models. The `SegmentGenerator` then calls the `resetAndTryAppendBuffer()` method, which does the following two things:
  - (a) Resets the *activeValueModelTypes* and *activeTimeStampModelTypes* lists in the `CompressionModelManager` to contain all the enabled model types.
  - (b) Then appends the remaining data points in the *notYetEmitted* buffer to the now restarted *activeValueModelTypes* and *activeTimeStampModelTypes* lists.

The `constructSegmentsFromBuffer()` returns a list of segments because sometimes none of the reset models can fit all the data points in the *notYetEmitted* buffer in **step 3b**. An example of when this occurs is when the value of the newest data point is very far away from the previous data point's value and the *Gorilla* value model type is not enabled. If this occurs then we go back to **step 2** and generate a segment for data points that could be fitted to the model types in **step 3b**.

### 4.3.3 Compression models

As part of performing the model-based compression, IrregularDB should have classes for the timestamp model types and value model types described in **Sections 3.1 and 3.2**. For this, the abstract classes `ValueModelType` & `TimeStampModelType` are created, which act as superclasses for the specific model types. As the functionality for these two abstract classes overlap they both inherit from an abstract base class called `BaseModelType`. The diagram for the abstract classes can be seen in **Figure 4.10**.



**Figure 4.10:** Simplified class diagram of `BaseModelType`, `ValueModelType` & `TimeStampModelType`

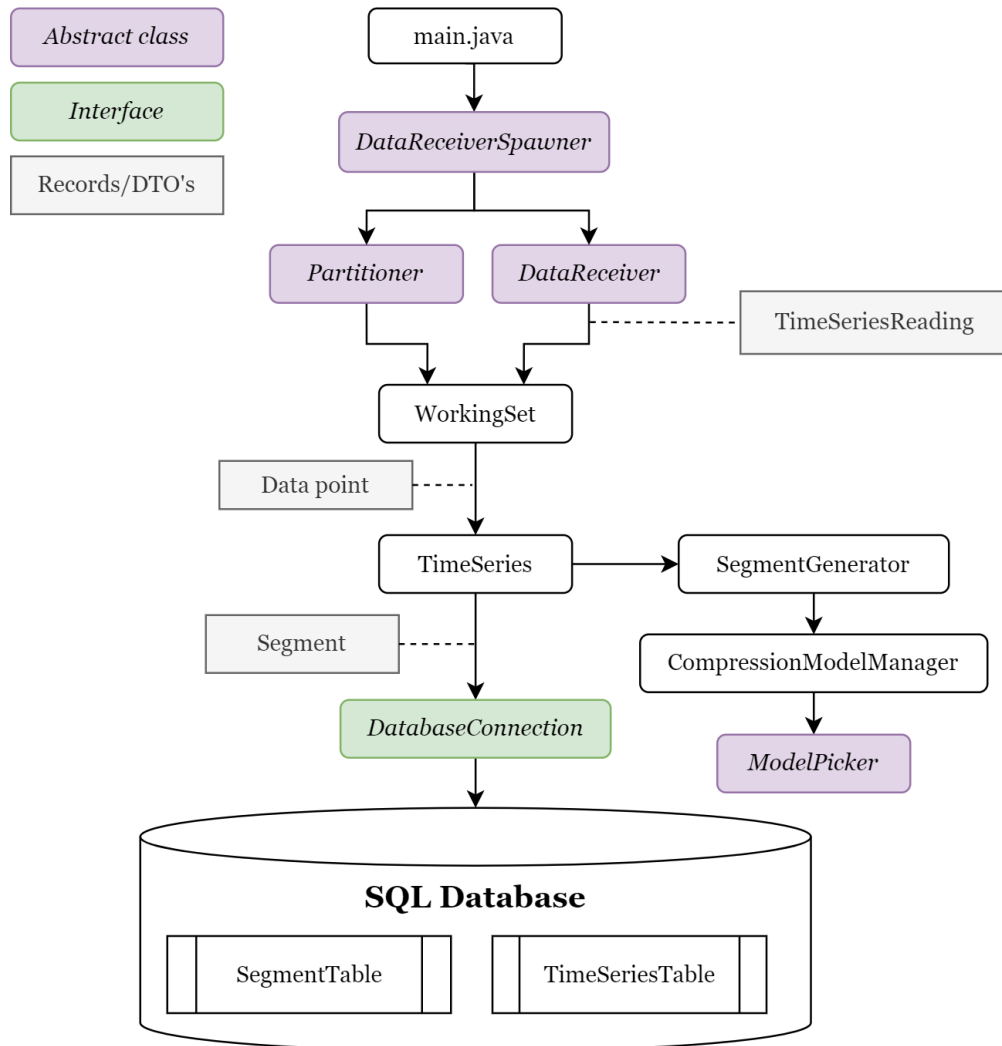
The `BaseModel` contains common method implementations that both the `ValueModelType` and `TimeStampModelType` use. The *`getBlobRepresentation()`* method returns a blob representation of the current `TIMESTAMP MODEL` for timestamp model types and `VALUE MODELS` for value model types.

The *`reduceToSizeN()`* method is used during the model selection in the `ModelPicker`. Consider the case where a value model type represents fewer data points than a timestamp model type. In this case, the timestamp model type will be reduced to the same size as the value model type because the models created by them must represent the same amount of data points in one segment.

The *`resetAndAppendAll()`* method is used in the `CompressionModelManager` to append the list of data points that are left in the `SegmentGenerator`'s *`notYetEmitted`* buffer after a segment has been created.

## 4.4 Component overview

Finally, connecting all the components described in the previous sections the full detailed overview of the system can be seen in **Figure 4.11**. The arrows in this figure denote associations.



**Figure 4.11:** Full component diagram.

The `DataReceiverSpawner`, `Partitioner`, and `DataReceiver` are responsible for handling input data and transferring it to a `WorkingSet`.

The `WorkingSet`, `TimeSeries`, `SegmentGenerator`, `CompressionModelManager`, and `ModelPicker` components are responsible for ingesting and compressing the input data.

Lastly, the `DatabaseConnection` and SQL database components are responsible for storage.

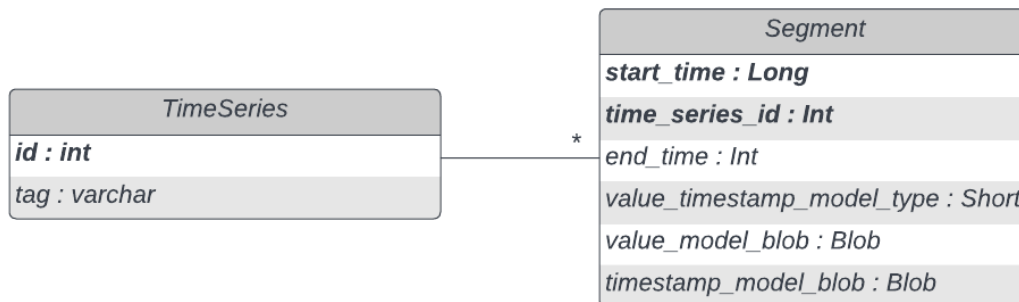


## 4.5 Database design

A requirement for the SQL database is that it must support the creation of UDFs (User-Defined Functions) in an imperative language. The reason for this is that it allows the database to be extended with segment decompressing capabilities as discussed in more detail in **Section 4.5.2**. Two databases were considered for the system: Oracle Database [13] and PostgreSQL [14]. Both databases support UDFs written in an imperative language and are relational databases. PostgreSQL was chosen because Oracle's terms of use do not allow for publishing test data run on their system whereas PostgreSQL is an open-source DBMS.

### 4.5.1 Schema design

When creating a database schema, consideration regarding if tables grow at a fast or slow pace is needed. This is particularly important to consider when developing a time series database system. For IrregularDB, there are two major types of information that need to be stored: metadata for TIME SERIES and time series readings for the different time series. For this purpose two tables are created, namely a time series table to store metadata and a segment table to store segments representing the time series readings. The time series table grows slowly as only one row is created per time series whereas the segment table grows rapidly as it stores the time series data. The tables are shown in the ER diagram in **Figure 4.12**.



**Figure 4.12:** ER diagram for the database of the system

The time series table only contains a tag and an auto-generated (incremental) id. A time series is identified by the tag from an outside perspective, however, a varchar is a very large key. Optimization is therefore done by adding an auto-generated integer id that serves as the primary key of the table instead. This is relevant because the segment table has a reference to the time series table. The time series table is kept very simple for now as it does not currently store any metadata regarding the time series. However, the table could easily be expanded with meaningful dimensions. For the scope of the project, this has not been implemented.

The segment table holds the information that represents the data that has been ingested by the system. Since this table grows rapidly several considerations have been given regarding keeping this table as small as possible:

- **Smaller foreign key:** The first measure, as mentioned above, is to use an auto-generated integer id for the time series as the primary key instead of a varchar tag. This ensures that the segment table only has to store a single integer to have a reference to the time series table.
- **Combined model type column:** The smallest numeric column type PostgreSQL supports is a smallint (2 bytes). This leads to an optimization being done on the `value_timestamp_model_type` column. Instead of storing two individual shorts for the model types it was chosen to combine two 1-byte values into a single short. The combination of the two values is done by storing the first value in the most significant byte and the second value in the least significant byte thereby saving 2 bytes for every segment.
- **Column tetris:** PostgreSQL expects data type columns to fit in eight bytes. If a four-byte column is followed by an eight-byte column followed by a four-byte column, PostgreSQL will insert four bytes of padding on the first column since it aligns all columns except the last column into eight bytes using a total of 20 bytes. If instead the eight-byte column is placed first then followed by the two four-byte columns only 16 bytes are used as no padding is needed saving a total of four bytes per row. The saved space can be verified by testing the `'pg_column_size()'` function on the previously mentioned table column layouts. This kind of column tetris has been done for the segment table in IrregularDB by ensuring the segment table's columns are aligned into 8 byte chunks so that no unnecessary padding is added.
- **End time stored as difference:** The last optimization that has been made in terms of compression ratio is to store the `end_time` as the difference from the `start_time`. This results in IrregularDB being able to store the `end_time` as an integer instead of a long reducing `end_time`'s storage usage from 8 bytes to 4 bytes.

Other than the already discussed fields the segment table simply holds the `start_time` of the segment, as well as the value model and timestamp model stored in Blob objects with a binary encoding.

## 4.5.2 Data querying

An important part of a DBMS is to be able to query the data stored in the system efficiently. In [15], a paper concerned with managing sensor (time series) data, they identify four types of fundamental query patterns for time series data:

- *Time point query:* returns the data point with the specified timestamp from a time series.
- *Value point query:* returns the data points from a time series whose value is equal to the query value. There may be multiple data points for which the time series' values satisfy the query value.
- *Time range query:* returns the data points of a time series that exist in the time range.
- *Value range query:* returns the data points of a time series whose values are within the value range.

Since IrregularDB stores segments it needs support for decompressing the segments to data points to support the above query patterns. The following section discusses possible solutions to handling the decompression of segments.

## Decompressing segments

The following two solutions for decompressing the segments were considered:

- **Query interface:** Some time series management systems, such as *ModelarDB*, provide a query interface. This requires queries to be sent to ModelarDB. From here, ModelarDB will extract the required data from the database and pass the result back to the caller. This requires that query handling is built into the application but alleviates the necessity of expanding the database with user-defined functions.
- **UDFs:** Another solution is instead to extend the DBMS with user-defined functions. This means that the responsibility to retrieve the original data points is placed solely on the DBMS.

The approach of using user-defined functions is chosen for IrregularDB. This solution was chosen as it allows IrregularDB to have full SQL syntax support without needing to implement custom query syntax parsing. Due to time constraints, only a proof-of-concept segment decompress UDF is implemented in Java and installed into the PostgreSQL database with the PL/Java extension [16]. It was chosen to go with a Java implementation as it allowed us to reuse code, however, the PL/Java expansion is relatively slow compared to native PostgreSQL functions implemented natively in C.

The user-defined function is defined to take a row from the segment table as input. In other words, the user-defined function takes an argument of the composite type segment, which represents a single row of the segment table. The function then executes Java code that can decompress segments into data points. A set of the composite type sqlDataPoint given as (timeSeriesId:integer, timestamp:bigint, value:real) is returned. The user-defined function can be seen as a flat map function that maps segments to data points.

The user-defined function is called ‘*decompressSegment*’. An example of its usage can be seen in **Listing 4.1**. The example shows how the UDF is used together with the time series table to select all data points for a single time series by providing a tag in place of *@myTimeSeriesTag*, which is a placeholder for an actual tag that is present in the database. The uncommon notation: (decompressSegment(s)).\* seen on line 1 turns the result, which is a set of composite type objects, into a table response. This notation is specific to PostgreSQL.

```

1 SELECT (decompressSegment(s)).* FROM segment s
2 JOIN timeseries t on s.time_series_id = t.id
3 WHERE t.tag = @myTimeSeriesTag;
```

**Listing 4.1:** Example query using UDFs

## Summary tables

Storing segments can risk resulting in poor query performance because full decompression of all segments in a time series might be required before being able to produce results depending on the query. To illustrate this consider the example where a user asks for all data points that have values larger than 2.0 for a specific time series. In this case, all the segments representing the time series would have to be decompressed before being able to produce results which are very costly for the query time.

From the four query patterns discussed in **Section 4.5.2** the *time point query* and *time range*

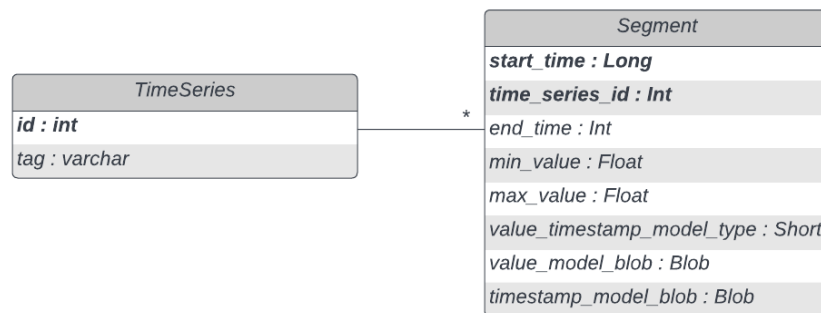
*queries* are well supported in IrregularDB by having the start and end times on the segment table. Having the start and end times on the segment table simplifies filtering segments that do not need to be decompressed. This filtering can be done by checking the specified timestamp against the start time and end time of a segment.

The current design used in IrregularDB results in long query times to answer the *value point* and *value range queries*, as IrregularDB does not support accessing the values of the data points without full decompression as illustrated in the value range query example above, where the range  $[2.0, \text{Double.MAX\_VALUE}]$  was queried. To speed up value queries the proposed idea is to add summary information in the database, which for example could be the average, max, or min value for a segment.

When answering a value point or range query the summary information can then be used to prune away segments that should not be decompressed to answer the query. The user-configurable parameter ‘`model.segment.compute.summary`’ is added to allow the user to decide if the increase in query performance is worth the extra storage space and a possible increase in ingestion time. Two options to implement summary information in the database were considered:

1. **Summary information on segment table:** The summary information can be directly added as new columns on the segment table.
  - **Advantage:** No joins necessary and no wasted storage on foreign keys.
  - **Disadvantage:** If summary information is disabled by the user then null values have to be stored in the empty columns using storage space.
2. **Summary table:** Adding a table that maintains summary information with a reference to segments.
  - **Advantage:** No need to store null values when summary information is disabled.
  - **Disadvantage:** Joins need to be done to access summary info and foreign key references are necessary.

The first approach is chosen because in PostgreSQL if a table contains eight or fewer columns then it is free to store null values [17] thereby eliminating the disadvantage of the first approach. The segment table from **Figure 4.12** is therefore updated with `min_value` and `max_value` columns used for summary information as seen in **Figure 4.13**. Note that the order of the fields in the segment table is important given the column tetris discussed in **Section 4.5**.



**Figure 4.13:** Updated version of the database schema from **Section 4.5**

The added summary information could then be used to prune away irrelevant segments before decompressing them. An example of this could be done as shown in **Listing 4.2** in which the user should provide a min and max value of their range in place of *@minRangeValue* and *@maxRangeValue*. In this query an overlapping range check is added on line 3 that ensures only the relevant segments are selected for decompression.

```

1  SELECT timeseriesId, timestamp, value FROM (
2      SELECT (decompressSegment(seg)).* FROM segment seg
3      WHERE seg.min_value <= @maxRangeValue AND @minRangeValue <= seg.max_value
4  ) WHERE @minRangeValue < value AND value < @maxRangeValue;

```

**Listing 4.2:** Value range query example using summary information

### Support for common query patterns

Since IrregularDB allows a threshold and error bound the actual stored values can differ from the original values. This affects the queries. Consider the case where a user requests the data point with timestamp 1303132931. If a threshold higher than zero was used when ingesting then there is a chance that the exact timestamp does not exist in the database. The same can happen for data point values when an error bound is used.

To be able to locate data given knowledge of the data before ingestion, four UDFs are implemented. These UDFs are implemented in SQL and support the four query patterns, mentioned at the start of **Section 4.5.2**. The four UDFs functions can be found on GitHub<sup>1</sup> and their definitions can be seen below:

- `timePointQuery(tid integer, timestamp bigint, threshold integer)`
- `valuePointQuery(tid integer, theValue real, errorBound real, useSummary boolean)`
- `timeRangeQuery(tid integer, lowerBound bigint, upperBound bigint, threshold integer)`
- `valueRangeQuery(tid integer, min real, max real, errorBound real, useSummary boolean)`

As seen here, the time-based queries take a threshold as input. This threshold is then for the *time range* query used to extend the provided lower and upper bounds to accommodate the approximations done by IrregularDB. This is done by updating the bounds to be: `lowerBound := lowerBound - threshold` and `upperBound := upperBound + threshold`

The *time point* query is implemented as a small time range query where both the upper and lower bound are equal to the provided timestamp. The threshold is therefore used in the same manner for the *time point* queries as for the *time range* queries.

The *value point* query is also implemented as a *value range* query, where similar moving of the min and max value is done using the provided error bound. It is worth noting that the value query UDFs also takes a boolean value called `useSummary` as input, which specifies if summary information should be used to help speed up the query. This parameter should only be set to true if the data has been ingested with summary information enabled, otherwise, the function does not work.

<sup>1</sup><https://github.com/IrregularDB/IrregularDB/blob/master/irregulardb-core/src/main/resources/allSqlUDF.sql>

# 5 | Implementation

This chapter will describe the implementation of *IrregularDB*. **Section 5.1** will present an overview of relevant details regarding the implementation. Then **Section 5.2** will describe an integration test created to ensure that IrregularDB works as intended.

## 5.1 Overview

IrregularDB was implemented in Java 18 as this is the programming language with which the team has the most experience. The source code is publicly available through GitHub [18]. In total, the project consists of approximately 7800 lines of Java code. A total of 157 unit tests have been created to ensure the functionality of individual components works as expected.

IrregularDB can be built with maven. A configuration file is needed to execute the resulting JAR. The IrregularDB JAR expects the configuration file to be named `config.properties` and to be in the same directory as the JAR.

The choice of IrregularDB using floating-point values instead of doubles, discussed in **Section 3.2**, led to a problem when implementing the *Swing* VALUE MODEL TYPE. To exemplify the problem, consider the cases where Swing ingests the TIME SERIES INTERVAL  $TSI = \langle (0ms, 1.00), (1ms, 1.05), (2ms, 1.10) \rangle$ . In this case, one would expect Swing to be able to represent the  $TSI$  within an error bound of  $\varepsilon_v = 0\%$ . However, due to floating-point imprecision Swing tried to create the linear function  $m_v := f(t) = 0.049999952t + 1$ , which meant that Swing could not represent  $TSI$  within  $\varepsilon_v$ . The problem is solved by transforming the error bound to  $\varepsilon_v = 0.001\%$  when an error bound  $\varepsilon_v = 0\%$  is provided. This solution allows Swing to represent  $TSI$  from the example above, however, also means that IrregularDB does not support truly lossless value compression as it allows errors up to 0.001%.

## 5.2 Integration test

The goal of the integration test is to ensure that DATA POINTS can be correctly ingested and reconstructed when all system components are working together. The integration test is performed by running the system and ingesting the dataset described in **Section 5.2.1**. The dataset is ingested with various combinations of error bounds and thresholds as described in **Section 5.2.2**. The results of the integration test are then discussed in **Section 5.2.3**.

### 5.2.1 REDD dataset

The REDD dataset [4] is used for the integration test. The REDD dataset is a public dataset containing time series data describing energy consumption for various channels/sources in six different houses. In total, the REDD dataset contains 56,341,629 data points. If each data point uses 12 bytes (timestamp:long, value:float) then REED uses a total of approximately 676MB if stored as raw data points.

Running a script that checks if a successive data point has a timestamp that is larger than the previous reveals that data points in some cases are out of order in the dataset. The dataset was therefore preprocessed by sorting the data points based on timestamps with a script.

### 5.2.2 Integration test cases

All integration tests were performed with the following fixed values for user-configurable parameters:

- **Model Types:** All `TIMESTAMP MODEL TYPES` and value model types
- **Model Picker:** Brute Force (based on results in **Appendix D.2**)
- **Length bound:** 400
- **Strict Error Bound:** True (should only affect test cases, where  $\text{threshold} > 0$  by disabling Swing in these test cases as discussed earlier in **Section 3.2.1**)

Integration tests were performed with the following three variations in user-configurable parameters for the data in house 1:

- **Error bound=10%, Threshold = 0**
- **Error bound=0%, Threshold = 1000**
- **Error bound=10%, Threshold = 1000**

An integration test with the following settings was performed for all houses:

- **Error bound=0%, Threshold=0**

After the REDD data was ingested the resulting `SEGMENTS` were extracted from the database and decompressed. The decompressed data points were written to a file. A script was then used to verify that the data points from the original file and the recreated data points were within the error bound and threshold from the corresponding test case settings. This script also verifies that the timestamps of the reconstructed data points are still ordered correctly to ensure IrregularDB upholds the condition for time series found in **Definition 2.3.2**.

### 5.2.3 Result of the integration test

The integration tests revealed errors in the initial implementation of the system. The oversights found were:

- Missing handling of threshold affecting the reconstructed values (solution in **Section 3.2.1**)
- The need to handle out of order data points due to threshold (solution in **Section 3.5**)
- The necessity of throttling the amount of data received to avoid memory overflow (solution in **Section 4.2.4**)

After having fixed the oversights all the integration test cases passed as all the data points could be reconstructed correctly within their respective error bounds and thresholds.

## 6 | Evaluation

*IrregularDB* will be tested and compared to *ModelarDB* and *InfluxDB* [19]. *ModelarDB* is an obvious choice as *IrregularDB* is inspired by *ModelarDB*. As *ModelarDB* only supports ingestion of regular data, *ModelarDB* will only be compared to the other systems on regular TIME SERIES data. *InfluxDB* is an opensource time series database that is widely popular [3]. *InfluxDB* is offered as three products: *InfluxDB* enterprise, *InfluxDB* cloud, or *InfluxDB* OSS. For the evaluation, *InfluxDB* OSS 2.2 is used as it is the newest opensource version available.

The systems are tested on two datasets: The REDD dataset and a TSBS [20] generated dataset, these datasets will be described in **Section 6.1**. Then in **Section 6.2** the hardware used and settings of the three systems will be described. For each of the systems the following evaluation metrics will be measured: storage usage in **Section 6.3**, ingestion speed in **Section 6.4**, and query speed in **Section 6.5**.

### 6.1 Datasets

For evaluating the systems two different datasets are used: The REDD dataset that was described in **Section 5.2.1** and a second dataset generated using the data generator found in TSBS (Time Series Benchmark Suite) [20]. This section will describe the two datasets as well as describe any preprocessing that has been performed on the data before using it for the evaluation.

#### 6.1.1 REDD

To be able to test all the systems two versions of the REDD dataset are created:

- **Irregular version:** the original timestamps are used. This is used to test *IrregularDB* vs. *InfluxDB*. This dataset is also referred to as REDD-IR.
- **Regular version:** each timestamp is set to the previous timestamp + 1 second. This is used to test *IrregularDB* vs. *ModelarDB* vs. *InfluxDB*. This dataset is also referred to as REDD-R.

The need for the regular version of the dataset is because *ModelarDB* does not support irregular data.

#### Extending the REDD dataset

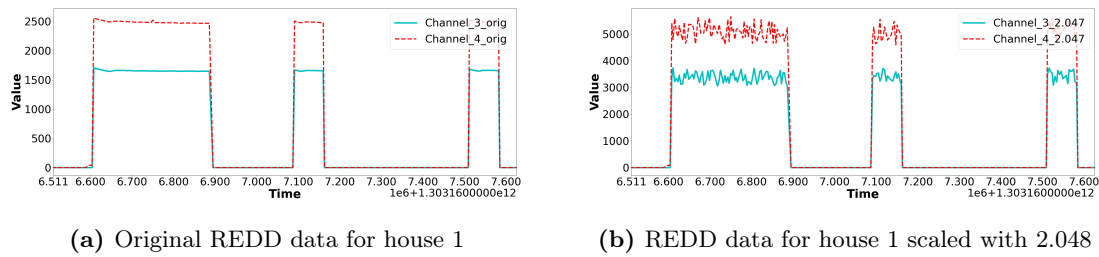
As the original REDD dataset is relatively limited in size, both the regular and irregular versions are extended. This means that copies of the houses from the REDD dataset were created. However, to not simply have the same data, the values in the time series were multiplied by a random factor between 0.1 and 10, which ensures that the generated data is different from the original data. The factors used were rounded to three decimal places and all factors used were unique meaning no factor was used twice. For example, we apply the following three factors to house 1:



- House\_1\_1.000x (the original values)
- House\_1\_2.000x (house 1 scaled so that each value  $v_i = 2 \cdot v_{i-orig}$ )
- House\_1\_0.642x (house 1 scaled so that each value  $v_i = 0.642 \cdot v_{i-orig}$ )

Some additional variance was then introduced by multiplying each value in each time series with a random number between 0.9 and 1.1 after multiplying them with their scaling factor. This means that for each time series its rows in the CSV-file would have their value multiplied by a random variance-value, say 0.961 for the first row's value, 1.031 for the second row's value, etc.

**Figure 6.1** illustrates the effect of this scaling and variance. **Figure 6.1a** shows a small sample of the original data from house 1's channel 3 and channel 4. Then in **Figure 6.1b**, the data has been transformed by using a scaling factor of 2.048 and multiplying it with the random numbers between 0.9 and 1.1.



**Figure 6.1:** Example showing the effect of scaling and variance

The original REDD dataset, as discussed in **Section 5.2.1**, uses approximately 676 MB if stored as raw data points (long + float). However, since the data is saved in CSV format encoded in UTF-8 then the actual storage used on disk will be larger than the raw data. We, therefore, decided to create raw data equal to around a third of the size of the disk to ensure all the data could be contained on a single disk with size 480GB. This meant that we ended up multiplying both the regular and irregular REDD datasets with 225 different factors as it gives a total of 12.677 billion DATA POINTS resulting in  $225 \cdot 676\text{MB} = 152\text{GB}$  of raw data for both the regular and irregular dataset. The upscaled REDD datasets each uses a total of 334GB when stored as CSV data on disk. This meant that each data set could be stored separately on one of the disks while still leaving space on the disk as a precaution.

### 6.1.2 TSBS

The TSBS dataset was generated by using the data generator from the Time Series Benchmark Suite (TSBS) [20] project, which is an opensource project that provides a variety of tools for testing time series DBMSs. The data generator was configured to use the ‘dev ops’ use case, and generate data for 3 hosts, spanning 770 hours with a data point every 100 ms. The data format of the generator was set to ‘influx’. This results in a total of 8.399 billion data points, which is equal to  $8.399 \cdot 10^9 \text{data point} \cdot 12 \frac{\text{byte}}{\text{data point}} \approx 100\text{GB}$  of raw data. Raw data means 4 bytes per value and 8 bytes per timestamp. For the TSBS dataset, a lower amount of raw data is used than for REDD because the TSBS data contains some additional metadata in the form of dimension data. The total amount of storage space used by the TSBS data, when stored as

CSV data on disk is 325GB. Similar to the REDD datasets, empty space was left on the disk as a precaution.

The TSBS data generator outputs generated data into a single file using the line protocol format [21]. Each row includes a row type, a set of dimensions, a set of readings, and a timestamp.

**Listing 6.1** shows an example of two rows in the generated data. The dimensions, readings, and timestamp are separated by a space and the individual metrics are comma-separated.

```

1  cpu,hostname=host_0,region=eu-west-1,datacenter=eu-west-1c,rack=87,
   os=Ubuntu16.04LTS,arch=x64,team=NYC,service=18,service_version=1,
   service_environment=production usage_user=58i,usage_system=2i,
   usage_idle=24i,usage_nice=61i,usage_iowait=22i,usage_irq=63i,
   usage_softirq=6i,usage_steal=44i,usage_guest=80i,usage_guest_nice=38i
   1451606400000000000
2  cpu,hostname=host_1,/* ... */ usage_user=47i,usage_system=93i,
   usage_idle=16i,usage_nice=23i,usage_iowait=29i, usage_irq=48i,
   usage_softirq=5i,usage_steal=63i,usage_guest=17i,usage_guest_nice=52i
   1451606400000000000

```

**Listing 6.1:** Example of two rows in generated data from TSBS data generator tool

The generated data measures different readings for hardware components on different hosts with some additional dimensions. The dimensions can be simplified to a hardware component (CPU in the example) and a hostname as these will still uniquely identify a reading as shown on line 2, where the irrelevant dimensions have been commented out.

IrregularDB and ModelarDB are not able to ingest this format. Therefore, a python script was created that transforms the generated data into a format IrregularDB and ModelarDB can ingest. The new structure is to split the file into multiple files by creating a file for each hardware component, host, and measurement key. This would for example lead to the following files being created:

- host0-cpu-usage\_user.csv
- host0-cpu-usage\_system.csv
- host0-cpu-usage\_idle.csv
- ...

The created files contain a comma-separated timestamp and the measured value at this time for measurement stored in the given file.

## 6.2 Evaluation setup

All of the evaluation is performed on the same server, which is provided by Aalborg University. The server specs are the following: 1x AMD 7302p 16 cores @ 3GHz, 256 GB RAM (8x32), 8 x 480 GB SATA SSDs. All of the systems are run as single-node systems on the mentioned server running Ubuntu 20.04.2. The following relevant programs were installed on the server: Java 18.0.1 and Postgres 12.9. All JARs executed for tests were run with additional heap memory with the flag: *-Xmx200g*, meaning 200 GB of heap memory is allowed to be usable for the executable JAR.

### 6.2.1 IrregularDB setup

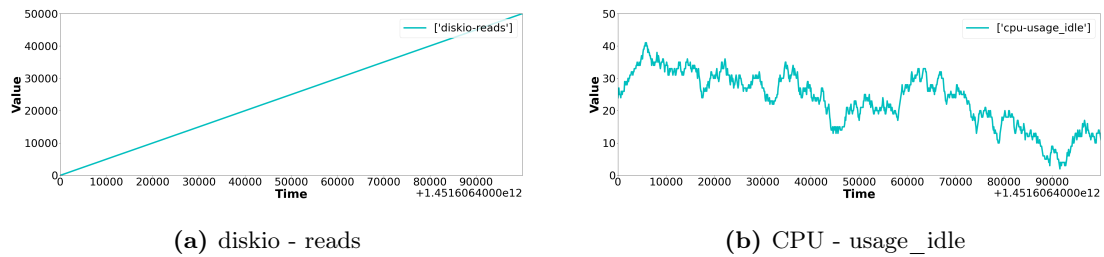
The fixed parameters set in IrregularDB are the following for all tests:

- **Value model types:** *PMC-mean*, *Swing*, and *Gorilla*
- **Model picker:** Brute force (based on **Appendix D.2**)
- **Strict error bound:** false
- **Threshold:** 500
- **Length bound:** 400 (based on **Appendix D.1**)
- **Maximum Segment Length:** 40,000
- **Throttle buffer size:** 10,000,000
- **Throttle sleep time:** 5,000
- **Batch size:** 10,000
- **Working sets:** 90

In the following the parameters whose values change based on the test case will be described. When testing on regular data only the *Regular* TIMESTAMP MODEL TYPE is active as it should be able to fit all the timestamps. However, when testing on irregular data all timestamp model types are active.

For the REDD dataset doing ingestion, an error bound of  $\varepsilon_v = 10\%$ . This error bound is reasonable because the REDD data seems to follow an ON/OFF pattern as illustrated earlier in **Figure 6.1**. This pattern means that even with a relatively high error bound the VALUE MODEL TYPES should still not be able to create linear or constant models that cross over these ON/OFF states as the difference between the ON/OFF values are very large.

In TSBS however, the data does not follow this ON/OFF pattern. Instead, the data for example follows a linear pattern as shown in **Figure 6.2a** or has small changes in value as seen in **Figure 6.2b**.



**Figure 6.2:** TSBS data visualization of part of HOST 0's data

The patterns shown in **Figure 6.2** mean that the changes in values are not as drastic as in REDD, where a device changes from ON to OFF. The error bound of  $\varepsilon_v = 10\%$  might therefore be too high as it could allow very long segments and might approximate away information that is of interest to the users. Two different configurations are therefore created for TSBS one with error bound  $\varepsilon_v = 1\%$  to test a lower error bound and one with  $\varepsilon_v = 10\%$  to allow for a more

precise comparison with the REDD results. This also allows us to test the effect of different error bounds.

Finally, all the datasets are ingested once with summary enabled and once without summary to test the effect of including summary information. This in total results in the following 8 runs for IrregularDB:

- Regular version of REDD w. summary
- Regular version of REDD no summary
- Irregular version of REDD w. summary
- Irregular version of REDD no summary
- TSBS with  $\varepsilon_v = 1\%$  and w. summary
- TSBS with  $\varepsilon_v = 1\%$  and no summary
- TSBS with  $\varepsilon_v = 10\%$  and w. summary
- TSBS with  $\varepsilon_v = 10\%$  and no summary

### 6.2.2 ModelarDB setup

ModelarDB is tested with the following parameters:

- **Error bound:** same as for IrregularDB
- **Length bound:** same as for IrregularDB
- **Ingestors:** same as amount of working sets for IrregularDB
- **Value models:** All available
- **Database:** PostgreSQL database

Since ModelarDB only supports regular time series it is only tested with the regular part of the REDD dataset. For the TSBS dataset both an error bound of  $\varepsilon_v = 10\%$  and  $\varepsilon_v = 1\%$  is used to test ModelarDB to be able to make a fair comparison to IrregularDB.

### 6.2.3 InfluxDB setup

InfluxDB is run with default parameters. Two different methods were used to ingest the data in InfluxDB:

- For the REDD datasets, a custom Java client using the influxdb-java library [22] was used to transform the CSV files of the REDD dataset into a format that InfluxDB could understand.
- For the TSBS dataset, the InfluxDB2 Client 2.3 [23] was used since the data generated for the TSBS dataset was already in a format InfluxDB could understand.

The TSBS dataset was also ingested using our Java client to test if these different ingestion methods had an effect. This test showed that ingesting the TSBS data using our Java client was

34.14% slower than when using the InfluxDB2 Client. This suggests that the Java client is likely to bottleneck the ingestion speed of InfluxDB for the REDD datasets.

Due to time constraints, it was chosen not to try and remove this java client bottleneck from the REDD ingestion speed test. As it is highly unlikely that InfluxDB would achieve better ingestion speed results than the other systems even if this bottleneck was removed. Because, as shown later in **Section 6.4**, IrregularDB ingests the REDD datasets 16.69 to 19.98 times faster than InfluxDB.

## 6.2.4 Presentation of results

In the following sections, bar charts show the results of the evaluation of the different systems. In these bar charts the following abbreviations will be used:

- **Datasets:**
  - The regular version of the REDD data set (REDD-R)
  - The irregular version of the REDD data set (REDD-IR)
  - TSBS data when ingested with a 1% error bound (TSBS-1)
  - TSBS data when ingested with a 10% error bound (TSBS-10)
- **Systems:**
  - IrregularDB without summary information (IRDB-no-sum)
  - IrregularDB with summary information (IRDB-w-sum)
  - InfluxDB (Influx)
  - ModelarDB (MDB)

Note that for TSBS-1 and TSBS-10 duplicate results will be used for InfluxDB as it has no concept of error bounds.

## 6.3 Storage usage

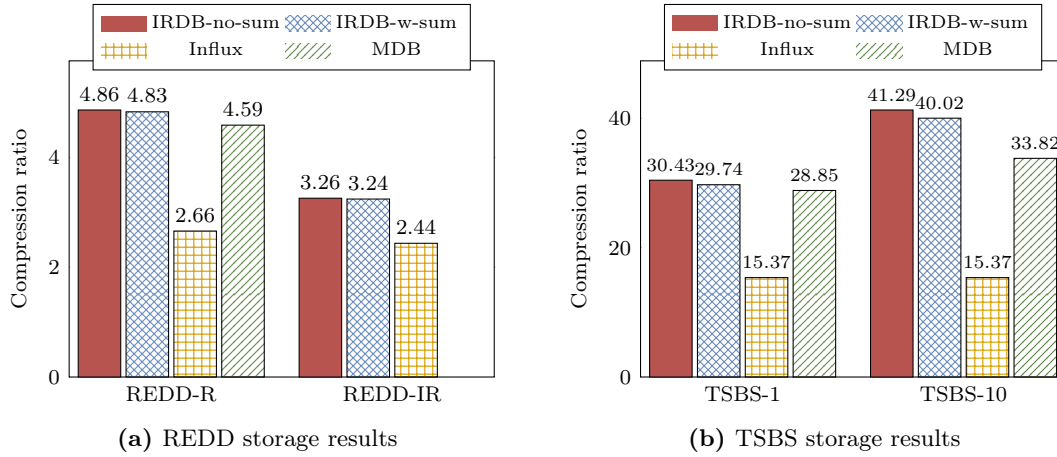
To compare the storage usage of the systems we need to first retrieve the storage usage values. For IrregularDB and ModelarDB the storage usage can be read from the PostgreSQL database. PostgreSQL supports the command `'pg_database_size()'`, which is used to measure the size of a database.

The size of the InfluxDB database is measured by seeing how much space the InfluxDB data folder uses on the disk.

### 6.3.1 Results

To be able to compare the different systems their compression ratios are calculated. Their compression ratios are calculated by dividing the size of the raw data (i.e. 152GB for REDD,

and 100GB for TSBS) by the measured storage usage. Compression ratios make it easier to compare results across the different data sets. **Figure 6.3** shows the calculated compression ratios for each of the systems - higher values mean better compression.



**Figure 6.3:** Storage usage results

The observations described in the following paragraphs can be derived from the storage usage results.

**Slightly worse compression with summary information.** IRDB-w-sum has a slightly worse compression ratio than IRDB-no-sum in all four test cases which is expected as two additional floats have to be stored for each segment. In the following paragraphs, the other systems will only be compared to IRDB-w-sum since the differences between IRDB-w-sum and IRDB-no-sum are insignificant and IRDB-w-sum performs the worst of the two.

**Similar results for IrregularDB and ModelarDB.** This is as expected, as both systems use similar *model-based compression* methods. IRDB-w-sum performs slightly better than MDB in all the test cases, where it is possible to test both systems. There can be various reasons for the small increase in compression ratio. The following reasons have been identified:

- **Optimized Gorilla value model type:** The Gorilla value model type, as explained in **Section 3.2**, was updated to be better suited for float values in IrregularDB. Similar updates are not done in ModelarDB.
- **Segment table optimizations:** IrregularDB as explained in **Section 4.5** stores differences for the end time of a SEGMENT. This allows IrregularDB to use integers for end time instead of the longs used in ModelarDB saving 4 bytes per segment. In addition, IrregularDB has also done column tetrism, which is not done in ModelarDB saving an additional 4 bytes per segment.

**IrregularDB gains a significant compression advantage when compressing regular data.** This can be derived from the fact that IRDB-w-sum for REDD-R compresses 81.7% better than Influx. Whereas for REDD-IR, IRDB-w-sum has a 33.1% better compression ratio than Influx. This can be contributed to fact that IrregularDB supports the Regular timestamp model type, which is very storage efficient.

**A higher error bound leads to better compression.** This can be seen in the measured compression ratios. As an example, IRDB-w-sum has a compression ratio of 29.735 and 40.024 for TSBS-1 and TSBS-10 respectively. These compression ratios show an improvement of 34.6% when the error bound is increased from 1% to 10%. Similar results can be observed for IRDB-no-sum and MDB. The reason behind this effect is that it allows the systems to create longer Swing and PMC-mean VALUE MODELS.

**All systems have better compression ratios on TSBS data.** Even though both TSBS-10 and REDD-R are regular and ingested with a 10% error bound then all three systems achieve significantly better compression ratios for the TSBS dataset than for REDD-R. For example, IRDB-w-sum has a 728.5% higher compression ratio for TSBS-10 than for REDD-R. The higher compression ratio is due to the value patterns present in the TSBS dataset being better suited for the utilized value model types as a large portion of the time series in TSBS follow a linear or constant pattern. In addition, no variance has been introduced in the TSBS dataset.

### 6.3.2 Summary

Overall IrregularDB achieves the best compression ratios based on the following observations:

- IRDB-no-sum achieves:
  - 1.337 to 2.687 times better compression compared to InfluxDB.
  - 1.055 to 1.221 times better compression compared to ModelarDB.
- IRDB-w-sum achieves:
  - 1.331 to 2.604 times better compression compared to InfluxDB
  - 1.031 to 1.184 times better compression compared to ModelarDB

## 6.4 Ingestion speed

The current system time in milliseconds is measured before starting to ingest the data and after finishing ingesting the data to measure ingestion speed for the different systems. The difference between these two values is then the ingestion time. The ingestion time of each system is measured three times and the averages are used when comparing the systems to reduce variance.

### 6.4.1 Results

**Figure 6.4** shows the ingestion speed results achieved for the three systems. The ingestion times have been converted to millions of data points ingested per second - higher values are therefore better. Using this unit allows for a more convenient comparison across the two datasets.

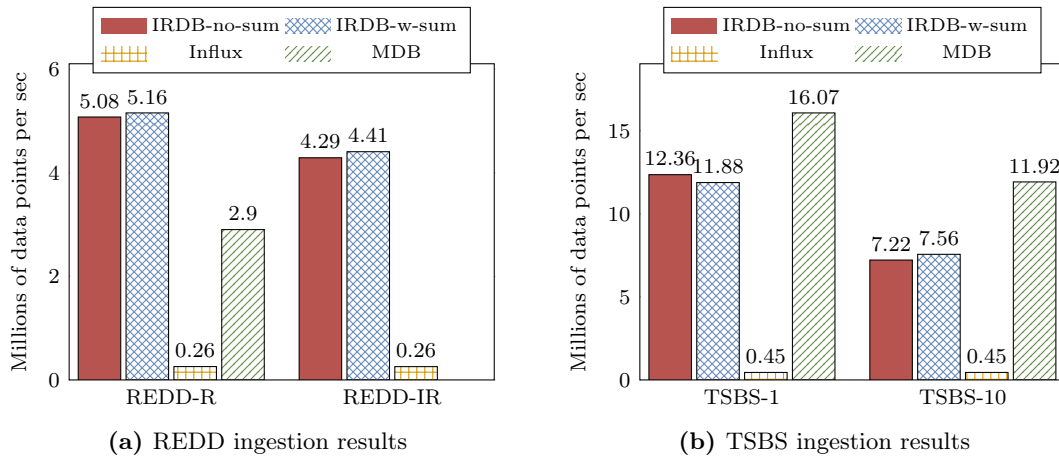


Figure 6.4: Ingestion speed results

The observations described in the following paragraphs can be derived from the ingestion speed results.

**InfluxDB is significantly slower than the other systems.** InfluxDB has an ingestion speed that is significantly lower than the other systems in all test cases. The primary reason for this is that the OSS version of InfluxDB used in this evaluation according to InfluxDB’s documentation is limited to around 750,000 field writes per second [24]. InfluxDB is therefore not considered in the following paragraphs when discussing ingestion speed.

**A lower error bound gives a higher ingestion speed for TSBS.** It can be observed that both ModelarDB and IrregularDB ingest TSBS-1 faster than TSBS-10. The difference between these two setups is that for TSBS-1 both ModelarDB and IrregularDB usually fall back on the Gorilla value model type, which always reaches the length bound. However, for TSBS-10 then Swing and PMC-mean can fit more data points, which can lead to them being selected over Gorilla even though they are still shorter than the length bound. Selecting shorter value models leads to shorter segments, which lowers the ingestion speed. Why shorter segments can lower the ingestion speed will be explained in **Section 6.4.2**.

To confirm that shorter segments were the cause of lower ingestion speed the percentage of segments that are short segments was calculated for TSBS-1 TSBS-10 for all three systems as shown in **Table 6.1**. We define a short segment as a segment, which has a length smaller than half of the length bound i.e. less than  $\frac{400}{2} = 200$ .

	TSBS-1	TSBS-10
IRDB-no-sum	0.21 %	76.00 %
IRDB-w-sum	0.08 %	72.00 %
MDB	0.19 %	76.27 %

Table 6.1: Percentage of segments that are short segments (i.e. shorter than 200 data points)

As can be seen from the table a significantly higher percentage of short segments is created for TSBS-10 than for TSBS-1. This means that TSBS-10 is more greatly affected by the problem described in **Section 6.4.2**, thereby leading to TSBS-1 being ingested faster. However,



IrregularDB still achieves higher compression ratio for TSBS-10 than for TSBS-1.

**Summary information affects ingestion speed.** Summary information effects ingestion speed as can be seen from the results it can both help speed up or slow down the ingestion speed. There are two primary reasons for this:

- **Recomputing values:** if summary information is enabled then created value models need to be decompressed to be able to identify the minimum and maximum value of a segment. This decompression takes time, which increases the ingestion time.
- **Summary leads to fewer short segments:** Summary information adds additional overhead to each segment. This overhead affects shorter models more than long models, as they represent fewer data points meaning their bytes per data point value are greater affected by additional overhead. The bytes per data point values, as stated earlier, are used to select models. This means that long models are more likely to be selected, which is a benefit for ingestion speed. This is a benefit because the system is less likely to run into the problem of short segments lowering ingestion speed discussed in **Section 6.4.2**.

**Table 6.2** is an extension of **Table 6.1** that also shows the percentage of short segments for REDD-R and REDD-IR. However, the data for MDB has been dropped as it has no concept of summary information.

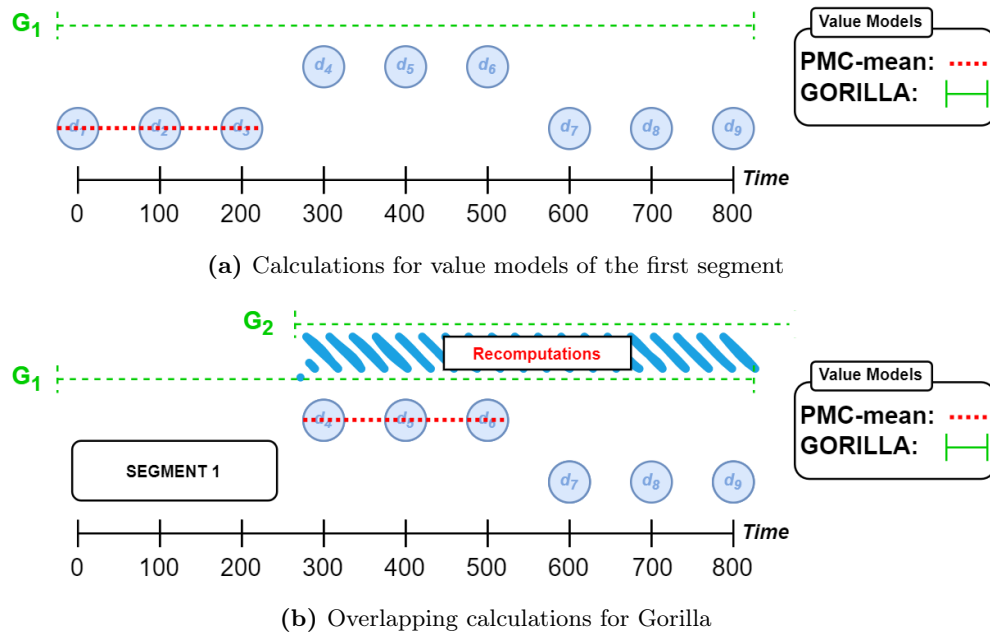
	REDD-R	REDD-IR	TSBS-1	TSBS-10
IRDB-no-sum	21.36 %	7.97 %	0.21 %	76.00 %
IRDB-w-sum	14.13 %	4.92 %	0.08 %	72.00 %

**Table 6.2:** Extension of **Table 6.1** with percentage short segments for REDD-R and REDD-IR

As seen in **Table 6.2** for IRDB-no-sum and IRDB-w-sum there is a difference ranging from 3.05 to 7.23 percent points in the number of short segments for REDD-R, REDD-IR, and TSBS-10. Having this many fewer short segments outweighs the additional cost of decompressing the values in these three cases. This leads to IRDB-w-sum ingesting the data faster as seen in **Figure 6.4**. However, for TSBS-1 the difference in the number of short segments is only 0.13 percent point, which means that here the benefit of having fewer short segments is smaller than for the other cases, which is also the reason why IRDB-w-sum is slower than IRDB-no-sum for TSBS-1.

## 6.4.2 Short segments slow ingestion

The problem with creating short segments is that it can lead to recomputations. This occurs because even though a short model, e.g. with a length of  $\frac{1}{3}$  of the length bound, is selected for the segment then computations are still done to create models with a length equal to the length bound for the model types that are length-bounded e.g the Gorilla value model type. **Figure 6.5** has been created to exemplify this. To keep the example short only the PMC-mean and Gorilla value model types are used with a length bound of  $L = 9$ .



**Figure 6.5:** Example of recomputations done when selecting models

In **Figure 6.5a** we can see how after the third data point the PMC-mean value model types can no longer fit the subsequent data points. However, computations continue for Gorilla until the length bound  $L = 9$  is reached. After reaching this length bound then one of the two value models has to be selected. If it is assumed that PMC-mean's value model has the best compression ratio then it is selected and the three data points represented by it are removed from the buffer.

This leads to the case shown in **Figure 6.5b**, where we can now start creating a value model for the following segment. The problem is that the Gorilla value model type ingests the data points  $d_4$  to  $d_9$  multiple times. These recomputations slow down the ingestion speed of the system. Having fewer short segments reduce this effect.

### 6.4.3 Summary

IrregularDB achieves the best ingestion speed on both the regular and irregular versions of the REDD dataset. However, on the TSBS dataset ModelarDB is the fastest system, which is different than for the REDD dataset. In total IrregularDB achieves the following:

- IRDB-no-sum achieves:
  - 16.69 to 27.39 times faster ingestion compared to InfluxDB.
  - 0.61 to 1.749 times the ingestion speed of ModelarDB.
- IRDB-w-sum achieves:
  - 16.15 to 26.34 times faster ingestion compared to InfluxDB.
  - 0.63 to 1.776 times the ingestion speed of ModelarDB.

The identified reason behind why ModelarDB outperforms IrregularDB on the TSBS dataset is that the time series in the TSBS dataset follows linear patterns. These linear patterns help ModelarDB beat IrregularDB because ModelarDB has no max segment length meaning that one to two segments are enough to represent many of the time series in TSBS. Whereas, IrregularDB enforces a max segment length, which leads to IrregularDB generating and sending more segments to the database than ModelarDB.

## 6.5 Query Speed

Seven unique queries are tested to evaluate the query performance of the different systems. The query patterns used for testing are described in **Section 6.5.1**. Every query is executed five times per system, where the highest and lowest results for each query are discarded to eliminate outliers. Then, the average of the remaining three results is presented in **Section 6.5.2**.

### 6.5.1 Query Patterns

The query patterns shown in **Table 6.3** are used to compare the query speed of the different systems. The first four query patterns are inspired by the query patterns used in TSBS [20], which is a popular benchmark suite for time series databases [9]. Query patterns no. 5 and 6 are created to test the *time point* and *value point* query patterns from the four fundamental patterns discussed in **Section 4.5.2**. There is no need to make an explicit query for the *time range* query pattern as it is tested in query pattern no. 1-2. The *value range* queries are tested using query pattern no. 4 as this is a value range from the high value to `Double.MAX_VALUE`. The last query pattern is for fetching an entire time series which could be useful for data visualization tools.

No.	Query Name	Description	Reason for testing
1	1-12	Simple aggregate (AVG) for <b>1</b> time series every 5 mins over a time span of <b>12</b> hours	Tests query speed for a single time series. This is also a time range query.
2	5-12	Simple aggregate (MAX) for <b>5</b> time series every 5 mins over a time span of <b>12</b> hours	Tests time range querying and aggregates across multiple time series
3	Last-point	Get the last data point of each time series	Tests lookup speed for newest readings from many sources (useful in system monitoring)
4	High-value	Get all data points with a value above a threshold for a single time series	Tests the systems' ability to filter on data point values
5	Value-point	All data points with a specified value for a time series	Tests performance for the <i>value point</i> query pattern
6	Timestamp-point	All data points with a specified timestamp for a time series	Tests performance for the <i>time point</i> query pattern
7	All	All data points for one time series	Tests query speed for an entire time series

Table 6.3: Query patterns

### 6.5.2 Results

Figure 6.6 is the legend data for the charts shown in Figure 6.7 and Figure 6.8. Figure 6.7 shows all query results on the REDD dataset across all systems. Each chart represents results for a query from Table 6.3 and shows the query time in milliseconds for each system tested. Figure 6.8 is the corresponding chart for the TSBS dataset. In both Figure 6.7 and Figure 6.8, lower values are better. Information about which specific time series from the datasets were used can be found on IrregularDB's GitHub page <sup>1</sup>. Note that ModelarDB does not support Last-point queries and the irregular dataset.





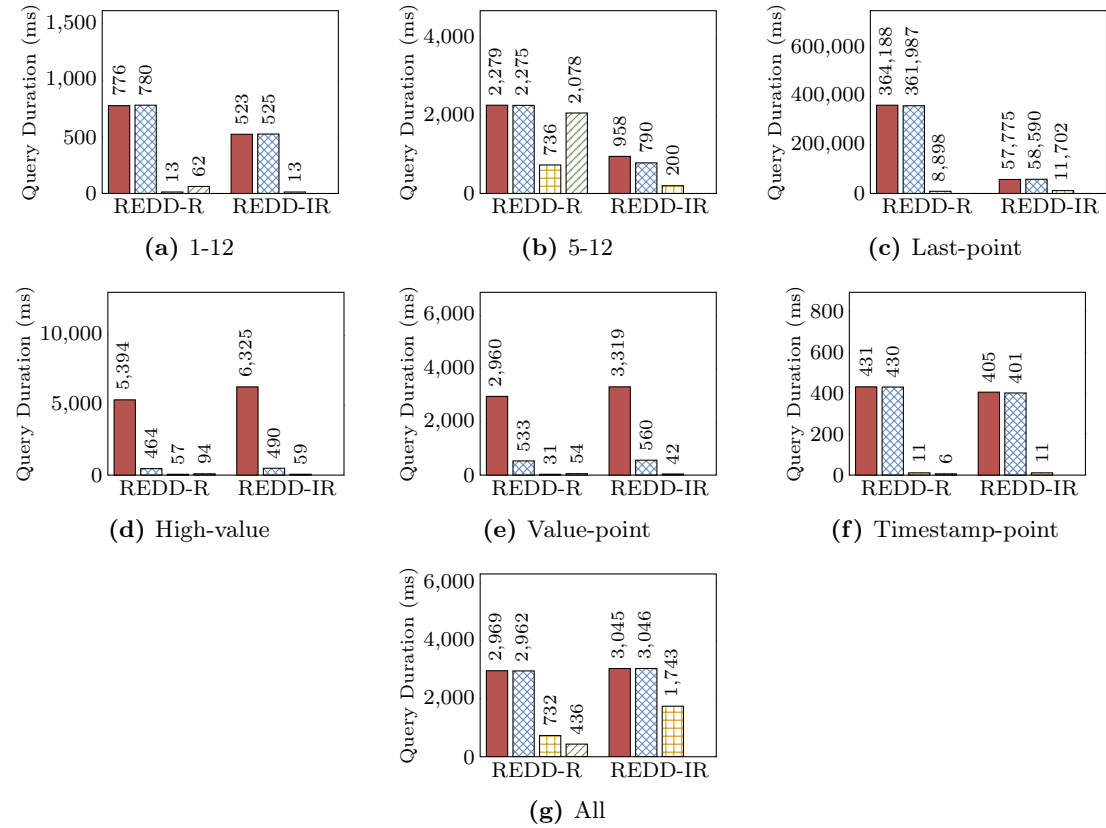
	IRDB-no-sum		IDB-w-sum
	Influx		MDB

Figure 6.6: Legend info for bar charts

<sup>1</sup><https://github.com/IrregularDB/Scripts>

### REDD query results



**Figure 6.7:** Query speed results for both the regular and irregular REDD dataset

**IrregularDB performs worse than the other systems overall across all queries for both REDD datasets.** This is to be expected as the focus has not been put on improving query speed in IrregularDB and is left as future work as discussed in **Section 7.4**. Whereas, both InfluxDB and ModelarDB have already implemented improved query support, which helps speed-up their queries.

**Summary information leads to speed-up in relevant queries.** IrregularDB only provides summary information in form of min and max values in an attempt to demonstrate the speedup that can be achieved. The speedup of the summary information can be seen for the high-value and value-point queries in **Figure 6.7d** and **Figure 6.7e**. Here, the following can be observed:

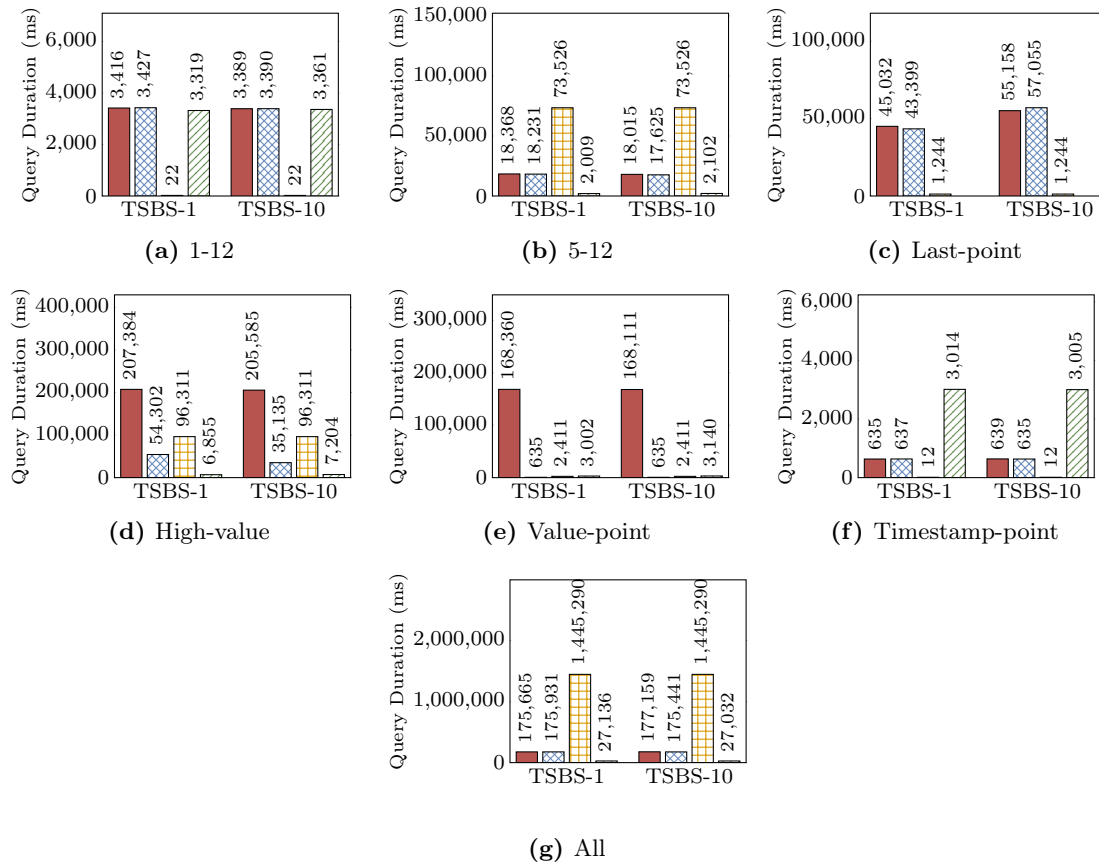
- For the high-value query for REDD-IR, the query response time was 6325 ms and 490 ms for IRDB-no-sum and IRDB-w-sum respectively. This means IrregularDB achieves a query speedup factor of 12.91 by utilizing summary information.
- For the value-point query for REDD-IR, the query response time was 3319 ms and 560 ms for IRDB-no-sum and IRDB-w-sum respectively. This means IrregularDB achieves a query speedup factor of 5.93 by utilizing summary information.

Similar results can be observed for the regular version of the data showing that summary in-

formation can significantly improve query speed without sacrificing much storage space and even increasing ingestion time in certain cases. The summary information also does not seem to slow down any of the other queries.

**IrregularDB is slower to decompress segments than ModelarDB.** In the all data points query pattern, all the segments in the time series need to be decompressed. This makes segment decompression speed the primary factor that affects the query speed for IrregularDB and ModelarDB for this query pattern. As shown in **Figure 6.7g** IRDB-w-sum is 6.8 times slower than ModelarDB for this query pattern. This suggests that IrregularDB is significantly slower at decompressing segments than ModelarDB.

### TSBS query results



**Figure 6.8:** Query speed results for the TSBS dataset

The TSBS dataset shows different results for query speeds compared to the REDD datasets in some cases. The primary differences will be discussed in the following paragraphs.

**Max segment length affects certain queries.** In the REDD dataset ModelarDB outperformed IrregularDB dominantly in the 1-12 query and timestamp point query. However, as seen in **Figure 6.8a**, ModelarDB performs similarly to IrregularDB for the 1-12 query on the TSBS dataset. Moreover, IrregularDB performs better than ModelarDB on the timestamp point query

for the TSBS data set as seen in **Figure 6.8f**. This is due to the time series queried for the 1-12 and timestamp point query following a linear pattern meaning that ModelarDB can represent them using only a single segment. This slows ModelarDB down significantly as it then has to decompress the entire time series to answer the queries. IrregularDB does not run into this problem as it has implemented a max segment length allowing it to only decompress the relevant data.

Notice that ModelarDB achieves a significantly lower query duration than IrregularDB on the 5-12 query, as seen in **Figure 6.8b**, even though both systems achieved similar results for the 1-12 query pattern. The reason for this is that the time series used in the 5-12 query does not follow a linear pattern. Therefore, the max segment length is irrelevant in the 5-12 query test case.

**InfluxDB results for the 5-12 query pattern are heavily impacted by source data format.** The way the TSBS data was ingested with InfluxDB made it necessary to introduce an extra map operation in the 5-12 query that converts each integer value to a float. Without the mapping operation, the query would fail. To our knowledge, this is the way to query the Influx database correctly. The results are representative of the real response time needed to answer an aggregate query across multiple time series when the source data is ingested as integers in InfluxDB. However, it is worth mentioning that better query speeds are expected if the source data was ingested as floats.

**Summary information can lead to IRDB-w-sum outperforming the other systems.** As was the case for the REDD datasets, the added summary information lead to a significant speed up for high-value and value-point queries as shown in **Figure 6.8d** and **Figure 6.8d**. For the high-value query pattern, IRDB-w-sum outperforms InfluxDB but not ModelarDB. The summary information is especially helpful for the value-point query, where IRDB-w-sum performs better than the other systems.

### 6.5.3 Summary

The results show that IrregularDB achieves worse query speeds for most queries compared to the other systems. The primary reason behind this is IrregularDB's decompress segment function being slow. IrregularDB is currently using a proof-of-concept segment decompress UDF implemented in Java as described in **Section 4.5.2**. This leads to IrregularDB being around 6 times slower at decompressing and returning an entire time series than ModelarDB, as seen in **Figure 6.7g** and **Figure 6.8g**.

The results also show that summary information (max and min for a segment) offers a significant increase in query speed for the relevant query patterns (high-value and value-point). It is therefore worth it to enable summary information compared to the relatively small trade-off in storage space.

# 7 | Future work

This chapter will cover the main future work that is left for *IrregularDB*.

## 7.1 Prettier querying

IrregularDB is queried through PostgreSQL. This leads to the syntax for the decompress function implemented being unintuitive, as users have to wrap the call in parenthesis and then say `.*` on it as illustrated in **Listing 7.1**.

```
1 SELECT (decompressSegment(s)).*
2 FROM segment s
3 WHERE s.time_series_id = 1
```

**Listing 7.1:** Example query using decompress segment

Another problem with having this decompress function is that the users need a deep understanding of the internals of IrregularDB to efficiently and correctly query the data. The reason for this is that the users need to understand how to efficiently filter out irrelevant SEGMENTS so that they don't waste time decompressing unnecessary segments.

Instead, IrregularDB could provide a data point view to make querying easier and prettier for the users. This would also allow the users to disregard the segment table completely and query the data as if it was stored plainly as normal data points.

A good middle ground between the data point view and decompress segment function is to implement SQL UDFs for common query patterns, as these can help the user query the system. Support for some of these has already been implemented as discussed in **Section 4.5.2**. However, supporting additional UDFs such as `lastpoint()`, etc. could help further improve the usability of IrregularDB.

## 7.2 Bucket sizes

The bucket sizes for the bucket encoding in **Section 3.1.3** were estimates for what were believed to be desirable bucket sizes. Further tests and analyses could be performed to optimize the bucket sizes. Selecting optimized bucket sizes could lead to better compression. The Gorilla [8] paper performed timestamp distribution analysis of their data domain to select fitting bucket sizes. A similar analysis could be done for data to be ingested in IrregularDB.

The bucket sizes are also subjects to become user-configurable parameters since the bucket sizes are likely to depend on the data domain.



## 7.3 Parameter tuning

Some of the parameters used for evaluating IrregularDB were decided based on some initial testing. For example, testing was done for the length bound and model picker as described in **Appendix D**. However, most of the configurable parameters were set to values thought to be good for the current data. A similar analysis could be done for the threshold, error bound, batch size etc. However, this parameter tuning was left as future work since there exist endless combinations of parameters and the ideal parameters depend on the ingested data.

## 7.4 Faster query times

Designing, implementing, and testing a full time series DBMS is not a trivial task and takes time. Early on the primary focus was, therefore, on supporting both regular and irregular time series data and achieving relatively good ingestion speeds and compression ratios. Query speed was therefore not prioritized as highly, which is also why IrregularDB achieved worse query speed for most of the queries tested in **Section 6.5**.

The central decompress UDF function implemented in Java seems to be a bottleneck in IrregularDB. This can be observed from the fact that IrregularDB is around 6 times slower to decompress an entire time series than *ModelarDB* as discussed in **Section 6.5.3**. A way to alleviate the bottleneck is by implementing this central decompress function in C. A faster execution time is expected from a C implementation due to the PL/Java integration with PostgreSQL having a significant overhead resulting in all queries being slower than necessary.

One of the few improvements made to try and speed up query time was to implement simple min and max summary information. As discussed in **Section 6.5.3** this kind of summary information can speed up certain queries. To further speed up IrregularDB an idea could be to further extend the system with additional helpful summary information to offer better query times in certain cases. An example of this could be to include information about the newest DATA POINT on the time series table in order to improve the ‘Last-point’ query performance.

## 7.5 Not having to decompress models during ingestion

In the IrregularDB system’s implementation, the models are decompressed during ingestion after models have been found for a segment. The full decompression is required to find the actual end time of the segment as this can be different from the original TIME SERIES INTERVAL after applying the threshold. The full decompression is also used for finding summary information for segments when summary information is activated.

A better implementation could be to let the model classes keep track of the summary information and compressed end times. Then, the information can be extracted from the model classes instead of performing a costly decompress operation on segments.

## 7.6 Breaking threshold

Recall that the chosen solution used to handle overlapping segments discussed in **Section 3.5.2** can in certain cases lead to IrregularDB breaking the threshold if a threshold higher than the minimum difference between two timestamps is used.

A possible solution to this problem is to make the `TIMESTAMP MODEL TYPES` aware that the timestamps have been moved. This could for example be done by providing the timestamp model types with both the original timestamp and the moved timestamp during ingestion. This would allow the timestamp model types to check that their approximations are within the allowed threshold compared to the original timestamp.

## 8 | Conclusion

The vast amount of sensor data produced for different purposes poses challenges in terms of storage and analysis. Different Time Series Management Systems (TSMSs) have already been developed, with varying success depending on the employed techniques, to offer acceptable compression and analytical capabilities. One such system that has been a source of inspiration for this project is the time series DBMS *ModelarDB*. *ModelarDB* constrains itself to regular time series data and then uses *multi-value-model-based compression* to achieve a high level of compression. This constraint of *ModelarDB* only supporting regular time series data leads to the following problem definition:

*How should a time series management system be made such that it supports effective compression and efficient ingestion using multi-model-based compression of values for both regular and irregular time series?*

To answer this problem definition a new time series DBMS, *IrregularDB* was designed, implemented, and tested. A novel model-based approach named Multi-Timestamp Multi-Value Model Compression (MTVMC) was proposed that was inspired by multi-value-model-based compression from *ModelarDB* and *multi-timestamp-model-based compression* from Informix [2]. The idea for *IrregularDB* was to use separate model types to compress the timestamps and values of the time series data for better compression. Using both `TIMESTAMP MODEL TYPES` and `VALUE MODEL TYPES` allow for an elegant solution to handle both regular, and irregular time series data.

Evaluation of the system was performed on three parameters: Storage usage, ingestion speed, and query times. The storage usage tests showed that *IrregularDB* achieves  $\sim 1.03$  to  $\sim 1.22$  times better compression ratios than *ModelarDB* on regular time series data and  $\sim 1.33$  to  $\sim 2.69$  times better compression ratios than *InfluxDB*.

For ingestion speed, the numbers showed that when comparing *IrregularDB* to *ModelarDB* then *IrregularDB* achieved  $\sim 0.61$  to  $\sim 1.78$  times *ModelarDB*'s ingestion speed depending on the value patterns present in the data. When comparing *IrregularDB* to *InfluxDB* then *IrregularDB* was between  $\sim 16.15$  to  $\sim 27.39$  times faster.

When it comes to query performance *IrregularDB* has the slowest query time for most of the queries. The bad performance is due to *IrregularDB* using a proof-of-concept decompression method implemented in PL/Java, which makes *IrregularDB*'s decompression significantly slower than *ModelarDB*'s decompression. A proof-of-concept decompression method was used since querying was not the primary focus of *IrregularDB*. Future work should therefore focus on improving the query performance by improving the custom user-defined function for decompressing segments by moving the implementation from Java to C. However, contrary to *ModelarDB*, *IrregularDB* offers full SQL query support and can handle irregular time series.

This project, therefore, fulfills the problem definition it sets out to solve as it succeeded in creating a TSMS that can manage both regular and irregular time series data while achieving effective compression and efficient ingestion by utilizing *multi-model-based compression* for the values.

# Glossary

**DeltaDelta** One of the timestamp model types implemented in IrregularDB. Focuses on handling all timestamp patterns. Similar to Gorillas timestamp model type. 14, 19, 20, 22, 27

**Fallback** For both timestamp and value model types a fallback model type is added IrregularDB. The purpose of this model is to handle singular data points. 14, 22

**Gorilla** One of the value model types implemented in IrregularDB. Focuses on handling arbitrary value patterns. 20, 22, 24, 25, 26, 39, 52, 55, 57, 58, 59

**InfluxDB** A time series DBMS developed by InfluxData written in Go. They have both an open source and enterprise version. Is a popular system and other systems are often compared to it. i, ii, 49, 53, 54, 56, 57, 59, 62, 64, 68

**IrregularDB** A time series DBMS developed by the authors. Can be used for both regular and irregular time series. It utilizes our novel approach Multi-Timestamp Multi-Value Model Compression (MTVMC) to do *model-based compression*. i, ii, iii, 1, 2, 4, 5, 7, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 29, 30, 31, 33, 34, 35, 36, 37, 39, 42, 43, 44, 45, 46, 47, 48, 49, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 76, 79, 80, 81, 83, 85

**ModelarDB** An open-source Time Series Management System that uses model-based compression to reduce the amount of storage needed to store sensor data in the form of time series. i, ii, iii, 1, 2, 7, 8, 13, 15, 44, 49, 51, 53, 54, 55, 56, 57, 59, 60, 61, 62, 63, 64, 66, 68

**model-based compression** A compression method used to compress time series data by representing their data using models instead of storing the raw data points. i, 2, 3, 4, 5, 7, 8, 9, 30, 31, 35, 36, 39, 55, 69, 70

**multi-timestamp-model-based compression** A sub set of *multi-model-based compression*, which focuses on utilizing multiple different timestamp model types to achieve better compression. i, 4, 10, 11, 12, 13, 14, 68

**multi-value-model-based compression** A sub set of multi-model-based compression, which focuses on utilizing multiple different value model types to achieve better compression. i, 4, 7, 8, 11, 12, 13, 20, 68

**multi-model-based compression** An extension of model-based compression, where instead of using only a single model type to compress time series data multiple different models are considered for a time series interval, and the model offering the best compression is used. i, 1, 4, 7, 68, 69

**PMC-mean** One of the value model types implemented in IrregularDB. Focuses on creating constant mathematical functions.. 20, 23, 25, 26, 52, 56, 57, 58, 59

**Regular** One of the timestamp model types implemented in IrregularDB. Focuses on regular and almost regular timestamp patterns. 14, 15, 16, 17, 18, 22, 23, 26, 27, 28, 52, 55

**SIdiff** One of the timestamp model types implemented in IrregularDB. Focuses on handling all timestamp patterns. 14, 17, 18, 20, 22, 27

**Swing** One of the value model types implemented in IrregularDB. Focuses on creating linear mathematical functions. 20, 21, 23, 25, 26, 47, 48, 52, 56, 57

**timestamp-model-based compression** A sub set of model-based compression, which is the part focused on compressing timestamp of a time series by using models. i, 3, 6

**value-model-based compression** A sub set of model-based compression, which is the part focused on compressing values of a time series by using models. i, 3, 6, 10

# Acronyms

**Influx** InfluxDB. 54, 55

**IRDB-w-sum** IrregularDB with summary information. 54, 55, 56, 57, 58, 59, 62, 63, 64

**IRDB-no-sum** IrregularDB without summary information. 54, 55, 56, 57, 58, 59, 62

**MDB** ModelarDB. 54, 55, 56, 57, 58

**MGC** Model-based Group Compression. 7

**MMGC** Multi-Model Group Compression. 7, 8

**MTVMC** Multi-Timestamp Multi-Value Model Compression. i, ii, 11, 13, 30, 68, 69

**REDD-R** The regular version of the REDD data set. 49, 54, 55, 56, 58

**REDD-IR** The irregular version of the REDD data set. 49, 54, 55, 58, 62

**TSBS-10** TSBS data when ingested with a 10% error bound. 54, 56, 57, 58

**TSBS-1** TSBS data when ingested with a 1% error bound. 54, 56, 57, 58

**TSMS** Time Series Management System. 1, 68

# Bibliography

- [1] Søren Kejser Jensen, Torben Bach Pedersen and Christian Thomsen. ‘Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB+’. In: (2021), pp. 1380–1391. DOI: 10.1109/ICDE51399.2021.00123.
- [2] Sheng Huang et al. ‘The next generation operational data historian for IoT based on informix’. In: (June 2014). DOI: 10.1145/2588555.2595638.
- [3] Alex Woodie. *It’s About Time for InfluxData*. 18th Feb. 2022. URL: <https://www.datanami.com/2022/02/18/its-about-time-for-influxdata/>.
- [4] J. Zico Kolter and Matthew J. Johnson. *REDD: The Reference Energy Disaggregation Data Set*. URL: <http://redd.csail.mit.edu/> (visited on 06/05/2022).
- [5] Florian Lautenschlager et al. ‘Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data’. In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 229–242. ISBN: 978-1-931971-36-2. URL: <https://www.usenix.org/conference/fast17/technical-sessions/presentation/lauteschlager>.
- [6] Søren Kejser Jensen, Torben Bach Pedersen and Christian Thomsen. *Time Series Management Systems : A Survey*. eng. 2017.
- [7] Tian Guo, Thanasis G Papaioannou and Karl Aberer. ‘Model-view sensor data management in the cloud’. eng. In: *2013 IEEE International Conference on Big Data*. IEEE, 2013, pp. 282–290. ISBN: 147991293X. URL: [https://www.researchgate.net/publication/261334847\\_Model-view\\_sensor\\_data\\_management\\_in\\_the\\_cloud](https://www.researchgate.net/publication/261334847_Model-view_sensor_data_management_in_the_cloud).
- [8] Tuomas Pelkonen et al. ‘Gorilla: A Fast, Scalable, in-Memory Time Series Database’. In: 8.12 (Aug. 2015), pp. 1816–1827. ISSN: 2150-8097. DOI: 10.14778/2824032.2824078. URL: <https://doi.org/10.14778/2824032.2824078>.
- [9] Zhiqi Wang, Jin Xue and Zili Shao. ‘Heracles: An Efficient Storage Model and Data Flushing for Performance Monitoring Timeseries’. In: *Proc. VLDB Endow.* 14.6 (2021), pp. 1080–1092. ISSN: 2150-8097. DOI: 10.14778/3447689.3447710. URL: <https://doi.org/10.14778/3447689.3447710>.
- [10] Thanasis Papaioannou, Mehdi Riahi and Karl Aberer. ‘Towards Online Multi-Model Approximation of Time Series’. In: 1 (June 2011). DOI: 10.1109/MDM.2011.57.
- [11] John Paparrizos et al. ‘VergeDB: A Database for IoT Analytics on Edge Devices’. In: *CIDR*. 2021. URL: <http://people.cs.uchicago.edu/~jopa/Papers/PaparrizosCIDR2021.pdf>.
- [12] J C Hale and H L Sellars. ‘Historical data recording for process computers’. In: *Chem. Eng. Prog.; (United States)* (Nov. 1981). URL: <https://www.osti.gov/biblio/5451707>.
- [13] Oracle. *Oracle Database Technologies*. URL: <https://www.oracle.com/database/technologies/> (visited on 19/05/2022).

- [14] The PostgreSQL Global Development Group.  
*PostgreSQL: The World's Most Advanced Open Source Relational Database*.  
URL: <https://www.postgresql.org/> (visited on 19/05/2022).
- [15] Tian Guo, Thanasis G Papaioannou and Karl Aberer.  
'Model-view sensor data management in the cloud'. eng.  
In: *2013 IEEE International Conference on Big Data*. IEEE, 2013, pp. 282–290.  
ISBN: 147991293X. DOI: 10.1109/BigData.2013.6691585. URL: <https://ieeexplore-ieee-org.zorac.aub.aau.dk/stamp/stamp.jsp?tp=&arnumber=6691585>.
- [16] PL / Java. *PL / Java*. 2017.  
URL: <https://tada.github.io/pljava/> (visited on 07/04/2022).
- [17] The PostgreSQL Global Development Group. *Database Page Layout*.  
URL: <https://www.postgresql.org/docs/12/storage-page-layout.html> (visited on 19/05/2022).
- [18] Esben Kaa Nedergaard, Kenneth Ljunggren Nørholm and Simon Teodor Manojlovic.  
*IrregularDB repository*.  
URL: <https://github.com/IrregularDB/IrregularDB> (visited on 19/05/2022).
- [19] influxdata. *InfluxDB OSS 2.2 Documentation*.  
URL: <https://docs.influxdata.com/influxdb/v2.2/> (visited on 19/05/2022).
- [20] Timescale. *Time Series Benchmark Suite (TSBS)*.  
URL: <https://github.com/timescale/tsbs#appendix-i-query-types-> (visited on 06/05/2022).
- [21] Influx Data. *Line Protocol*.  
URL: <https://docs.influxdata.com/influxdb/v2.2/reference/syntax/line-protocol/> (visited on 15/06/2022).
- [22] InfluxDB. *InfluxDB-Java repository*.  
URL: <https://github.com/influxdata/influxdb-java> (visited on 20/05/2022).
- [23] InfluxDB. *influx - InfluxDB command line interface*.  
URL: <https://docs.influxdata.com/influxdb/v2.2/reference/cli/influx/> (visited on 02/06/2022).
- [24] influxdata. *Hardware sizing guidelines*.  
URL: [https://docs.influxdata.com/influxdb/v1.8/guides/hardware\\_sizing/](https://docs.influxdata.com/influxdb/v1.8/guides/hardware_sizing/) (visited on 03/06/2022).
- [25] Esben Kaa Nedergaard et al. *Support for Adjustable Sampling Interval in ModelarDB*.  
14th Jan. 2022. URL: [https://github.com/ModelarDB-Dynamic/ModelarDB-Dynamic/blob/develop/ModelarDB-Dynamic\\_report.pdf](https://github.com/ModelarDB-Dynamic/ModelarDB-Dynamic/blob/develop/ModelarDB-Dynamic_report.pdf).
- [26] I. Lazaridis and S. Mehrotra.  
'Capturing sensor-generated time series with quality guarantees'.  
In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)* (2003), pp. 429–440. DOI: 10.1109/ICDE.2003.1260811.
- [27] Hazem Elmeleegy et al. 'Online Piece-Wise Linear Approximation of Numerical Streams with Precision Guarantees'. In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 145–156.  
ISSN: 2150-8097. DOI: 10.14778/1687627.1687645.  
URL: <https://doi.org/10.14778/1687627.1687645>.



# A | Gorilla Time Stamp Model Type

This appendix describes the `TIMESTAMP MODEL TYPE` used in the Gorilla paper [8] and Gorilla in general. Gorilla’s `VALUE MODEL TYPE` is presented in **Appendix B.3**. This appendix is taken from our previous report [25] and has have been slightly modified to better follow the definitions introduced in **Section 2.3**.

For compression of the timestamps, it was recognized that data often arrive at a relatively stable interval [8]. Therefore delta-of-delta time is used. Delta-of-delta time is as the name implies the difference in the difference of timestamps. E.g. if there are 60 seconds between each timestamp then the delta time is 60 for all of the timestamps. Then, because the delta time is the same between each data point the delta-of-delta time is 0 as there is no difference between the two delta timestamps. Delta-of-delta time is thus a good idea because timestamps often arrive with a fixed time interval.

In **Listing A.1** the algorithm for performing gorilla compression with a variable-length encoding on timestamps is described. In **Listing A.1** it can be seen that the timestamp is stored to a precision of within two hours of the first timestamp in the header of the chain, and the delta from this time to the actual time is stored as 14 bits for the first timestamp. From here, the range of the  $\Delta$ -of- $\Delta$  time defines which case from the algorithm is applied.

1. The block header stores the starting timestamp,  $t_{-1}$ , which is aligned to a two-hour window; the first timestamp,  $t_0$ , in the block is stored as a delta from  $t_{-1}$  in 14 bits.
2. For subsequent timestamps,  $t_n$ :
  - (a) Calculate the delta of delta:  

$$D = (t_n - t_{n-1}) - (t_{n-1} - t_{n-2})$$
  - (b) If  $D$  is zero, then store a single '0' bit
  - (c) If  $D$  is between  $[-63, 64]$ , store '10' followed by the value  $D$  (7 bits)
  - (d) If  $D$  is between  $[-255, 256]$ , store '110' followed by the value  $D$  (9 bits)
  - (e) If  $D$  is between  $[-2047, 2048]$ , store '1110' followed by the value  $D$  (12 bits)
  - (f) Otherwise, store '1111' followed by  $D$  using 32 bits

**Listing A.1:** An algorithm describing gorilla timestamp compression taken from [8]

**Table A.1** and **Table A.2** give an example of how the timestamps are encoded with the above algorithm. **Table A.1** is the block header. The specific case applied from the algorithm in **Listing A.1** can be seen in the "Case" column. In total (though excluding the header data) these  $4 \cdot 64$ -bit timestamps are stored in 40 bits (the encoding bits: 7 as well as the value bits: 33), which is a significant compression ratio.

Time	Binary
0	0000 0000 0000 0000

**Table A.1:** Block header for **Table A.2**

Timestamp	$\Delta$ -time	$\Delta$ -of- $\Delta$ time	Encoding bits	Value bits	Case
0	0	-	-	00 0000 0000 0000	1
60	60	$60 - 0 = 60$	10	011 1100	2C
120	60	$60 - 60 = 0$	0	-	2B
517	397	$397 - 60 = 337$	1110	0001 0101 0001	2E

**Table A.2:** Example values for timestamp compression

# B | Value model types

The following sections will describe the theory behind the VALUE MODEL TYPES used in *IrregularDB*. Their descriptions are taken from our previous report [25] and been slightly modified to better follow the definitions introduced in **Section 2.3**.

## B.1 PMC-mean Value Model Type

The PMC(Poor man's compression)-mean value model type [26] is a simple model type that constructs VALUE MODELS that are the average of a set of points [26]. Since PMC-mean represents the average then it is a value model type that outputs constant value models. New points are continuously sampled until the mean of all the current points is more than an error distance  $\varepsilon_v$  away from either the observed minimum or maximum point, and thus it is an online algorithm [26].

### Example

As a simple example consider the following TIME SERIES INTERVAL:

$$TSI = \langle (100ms, 3.33), (200ms, 3.31), (300ms, 3.41), (400ms, 3.35), (500ms, 3.28), (600ms, 5.30) \rangle$$

Say that the user defines an error bound  $\varepsilon_v = 5\%$ . When trying to construct a value model  $m_v$  for  $TSI$  we calculate the average value and calculate the error for the min and max point in  $TSI$ . In case both are within the error bound of  $\varepsilon_v$  the current DATA POINT can be appended to  $m_v$ .

To give an example of applying this value model type, the error for the **first five** data points is calculated to see if a value model can be constructed for these. To do this first the average, min, and max values are calculated:  $avg = \frac{3.33+3.31+3.41+3.35+3.28}{5} \simeq 3.34$ ,  $min = 3.28$ , and  $max = 3.41$ .

The errors for the min and max points are then:

$$error_{min} = \left| \frac{avg-min}{min} \right| \cdot 100\% = \left| \frac{3.34-3.28}{3.28} \right| \cdot 100\% \simeq 1.83\%$$

$$error_{max} = \left| \frac{avg-max}{max} \right| \cdot 100\% = \left| \frac{3.34-3.41}{3.41} \right| \cdot 100\% \simeq 2.05\%$$

Since the largest error is 2.05% for the max point we can construct a value model  $m_v$  for the first 5 data points within the error bound  $\epsilon = 5\%$ .

When including the sixth data point at time 600ms we get the following average, min, and max values:  $avg_{600} = \frac{3.33+3.31+3.41+3.35+3.28+5.30}{6} \simeq 3.66$ ,  $min_{600} = 3.28$ , and  $max_{600} = 5.30$ .

This gives the following error values:

$$error_{min-600} = \left| \frac{3.66-3.28}{3.28} \right| \cdot 100\% \simeq 11.59\%$$

$$error_{max-600} = \left| \frac{3.66-5.30}{5.30} \right| \cdot 100\% \simeq 30.94\%$$

Since the error is larger than  $\varepsilon_v = 5\%$  the data point at time  $600ms$  cannot be included in the value model constructed by PMC-mean for  $TSI$  and PMC-mean can therefore only construct a value model for the first five data points.

## B.2 Swing Filter Value Model Type

Swing Filter [27] is a filtering technique used to filter out data points that can be represented by a line segment within an error bound. Swing maintains a set of possible line segments for each filtering interval. The filtering interval is defined as being representable by a line segment within an error bound  $\varepsilon_v$ , such that whenever a new data point cannot be represented within the error bound, the line segment ends and a new interval is started. A filtering interval consists of two points (recordings) that together make a line. The last recording of an interval becomes the first recording of the next interval, meaning connected line segments are obtained.

Swing supports data in any number of dimensions, however, for the remainder of this section, we assume 1-dimensional data (i.e.  $d = 1$ ) so our data could for example look like:  $\langle (0ms, 0.0), (100ms, 1.0), (200ms, 2.0), (300ms, 0.0) \rangle$ . As data points are appended, Swing checks whether the value falls within the error bound of the upper and lower bounds. If it does the data point is filtered out and no recording is made. We say that the current bounds represent the data point within the error bound.

```

1 previous_recordings  $\leftarrow$  []
2  $(t_0, v_0) \leftarrow \text{getNext}()$ 
3  $(t_1, v_1) \leftarrow \text{getNext}()$ 
4  $R_0 \leftarrow (t_0, v_0)$  // Make a recording
5 previous_recordings.add( $R_0$ )
6 // Start a new filtering interval
7  $lower_1 \leftarrow$  a line passing through:  $R_0$  and  $(t_1, v_1 - \varepsilon_v)$ 
8  $upper_1 \leftarrow$  a line passing through:  $R_0$  and  $(t_1, v_1 + \varepsilon_v)$ 
9  $k \leftarrow 1$ 
10 while TRUE do
11    $(t_{next}, v_{next}) \leftarrow \text{getNext}()$ 
12   if  $(t_{next}, v_{next})$  is NULL or  $(v_{next} < lower_k(t_{next}) - \varepsilon_v)$  or  $(v_{next} > upper_k(t_{next}) + \varepsilon_v)$ 
13     // The point is null or outside allowed deviation
14     // Make a new recording
15      $R_k \leftarrow (t_k, v_k)$ , such that  $t_k \leftarrow t_{next-1}$ ,  $lower_k(t_k) < v_k < upper_k(t_k)$  and  $v_k$  minimize  $E_k$ .
16     previous_recordings.add( $R_k$ )
17     if  $(t_{next}, v_{next})$  is NULL
18       return previous_recordings
19     // Start a new filtering interval
20      $lower_{(k+1)} \leftarrow$  a line passing through:  $R_k$  and  $(t_{next}, v_{next} - \varepsilon_v)$ 
21      $upper_{(k+1)} \leftarrow$  a line passing through:  $R_k$  and  $(t_{next}, v_{next} + \varepsilon_v)$ 
22      $k \leftarrow k + 1$ 
23   else // The point was inside allowed deviation
24     if  $v_{next} > lower_k(t_{next}) + \varepsilon_v$ 
25       Swing  $lower_k$  up such that it passes through  $R_{k-1}$  and  $(t_{next}, v_{next} - \varepsilon_v)$ 
26     if  $v_{next} < upper_k(t_{next}) - \varepsilon_v$ 
27       Swing  $upper_k$  down such that it passes through  $R_{k-1}$  and  $(t_{next}, v_{next} + \varepsilon_v)$ 

```

**Listing B.1:** Swing Filter Algorithm, inspired by [27]

The algorithm for the Swing filter method can be seen in **Listing B.1**. `getNext()` reads the next data point and returns null if none exists. In lines 2-4 the first two data points are read and a recording is made of the initial data point.

On lines 7-9 the first filtering interval is started where  $upper_1$  is a line that has to pass through the initial data point and the second data point plus the error bound  $\varepsilon_v$ . The same occurs for  $lower_1$  but the  $\varepsilon_v$  is subtracted.

In line 12 it is checked if any more data points exists if they do then it checks if the next data point exceeds the upper bound (including error bound) or is below the lower bound (including error bound). If no data point exists or if the data point exceeds the bounds a recording is made (on line 15) of the previous timestamp with a value that generates a line segment that minimizes  $E_k$ , which is the mean square error for all data points observed in the  $k$ 'th interval. The equations used to determine the value for  $R_k$  that minimize  $E_k$  are omitted from this report, as they are not necessary to understand the idea behind the swing filter value model type. The equations can be found in [27]. Then if more data points exist a new filtering interval is started, as seen on lines 19-22.

If the data point instead was within the allowed deviation checked on line 12 then it is checked if the data point is more than  $\varepsilon_v$  within the error bounds. If this is the case then the bounds are "swung" to fit them within the distance of  $\varepsilon_v$ , as seen on lines 24-27. Finally, if the data point is within the error bound of both the upper and lower bound, the data point is said to be filtered out as it is already representable, which can be seen indirectly through the fact that no updates are done to the upper and lower bounds.

## Example

Consider the data points:  $\langle (0, 0), (1, 1), (2, 2), (3, 0) \rangle$  and an error bound of  $\varepsilon_v = 0.1$ <sup>1</sup>. The Swing Filter first makes a recording  $R_0$  at  $(0, 0)$ . It then creates an lower bound  $lower_0(t)$ , which should pass through  $R_0$  and  $(1, 1 - 0.1)$  and a upper bound  $upper_0(t)$ , which passes through  $R_0$  and  $(1, 1 + 0.1)$ . The lines used for the lower and upper bound are therefore:  $upper_0(t) = 1.1 \cdot t + 0$  and  $lower_0(t) = 0.9 \cdot t + 0$ . This is illustrated in **Figure B.1a**, where the black lines represent the upper and lower bounds, the green dotted lines represent the error bound of the upper bound, and the blue dotted lines represent the error bound of the lower bound.

The next data point  $(2, 2)$  is then read as seen in **Figure B.1b**. The data point is within the upper and lower bounds as none of the conditions checked on line 12 are true because  $v_{next} < lower_k(t_{next}) - \varepsilon_v \Rightarrow 2 < (0.9 \cdot 2 + 0) - 0.1 \Rightarrow 2 < 1.7$  for the lower bound and  $v_{next} > upper_k(t_{next}) + \varepsilon_v \Rightarrow 2 > (1.1 \cdot 2 + 0) + 0.1 \Rightarrow 2 > 2.3$  for the upper bound.

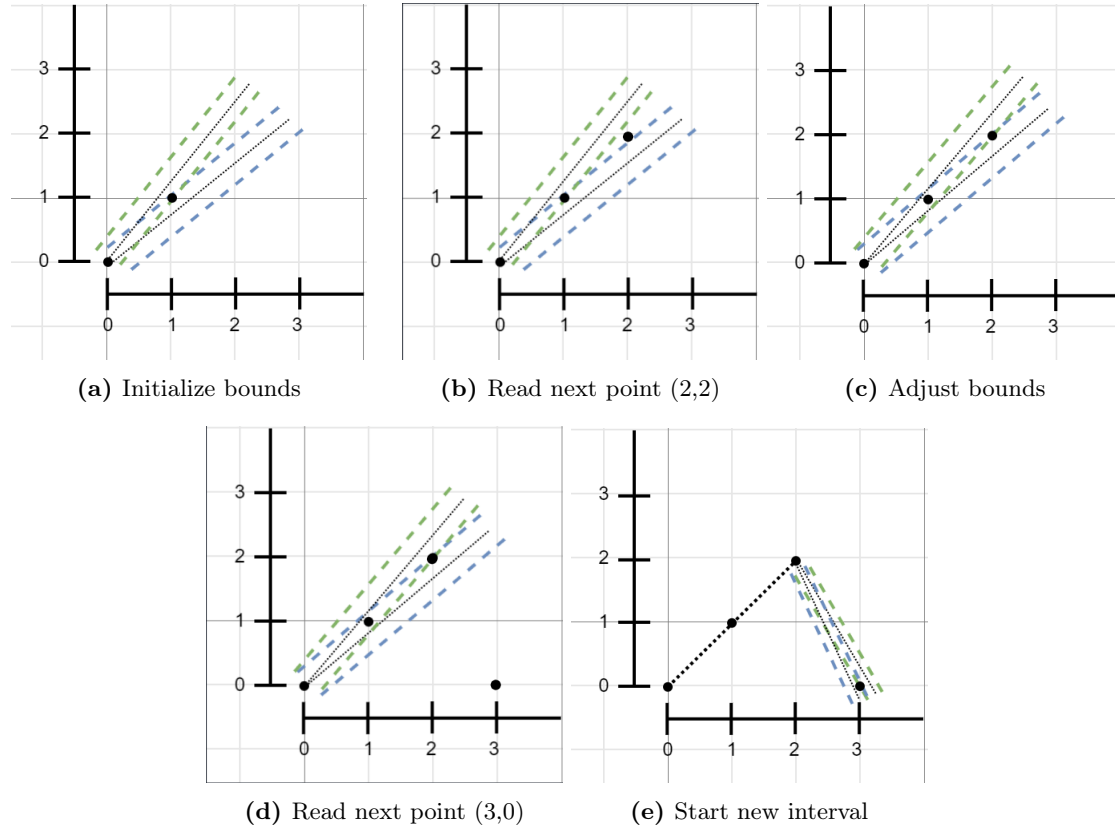
It is then checked if the upper and lower bounds require adjusting on line 24-27. Given that the data point is further than the defined error from the lower bound (line 24) because  $v_{next} > lower_k(t_{next}) + \varepsilon_v \Rightarrow 2 > (0.9 \cdot 2 + 0) + 0.1 \Rightarrow 2 > 1.9$  which is true so the lower bound is "swung" such that it passes through  $R_0$  and  $(2, 2 - 0.1)$  hence it is updated to  $lower_0(t) = 0.95 \cdot t + 0$  similar actions are taken for the upper bound resulting in  $upper_0(t) = 1.05 \cdot t + 0$ . This is illustrated in **Figure B.1c** where the adjusted bounds can be seen.

Then the data point  $(3, 0)$  is read as seen in **Figure B.1d**. This point is not within the lower bound since  $v_{next} < lower_k(t_{next}) - \varepsilon_v \Rightarrow 0 < (0.95 \cdot 3 + 0) - 0.1 \Rightarrow 0 < 2.75$  making at least one of the conditions on line 12 true. The current segment is therefore finalized and a new recording

<sup>1</sup>This value is a simplification of the real error bound because in IrregularDB the error bound would be e.g. 5%, which Swing filter would then use calculate an allowed derivation value such as 0.1

that minimizes the mean square error  $E_k$  is made.

The a new interval is started between our previous point and the new point (3,0), which is illustrated on **Figure B.1e**.



**Figure B.1:** Example of three steps of swing

In the case of IrregularDB, the data is also 1-dimensional, but rather than minimizing  $E_k$  to get a line segment, the average of the upper and lower bound is used. Since IrregularDB uses multi-model compression without overlap (i.e. disjoint segments) it will create a line segment only for one interval. This means that IrregularDB would therefore stop before the step shown in **Figure B.1e** and instead just emit the finalized segment from **Figure B.1d**. The compression ratio of the value model representing this line segment is then compared to the value models constructed by the other value model types.

## B.3 Gorilla Value Model Type

As part of IrregularDB, the value model type from Gorilla [8] is used. To get a better understanding of this value model type this section will describe the Gorilla algorithm conceptually and with an example.

Gorilla value model type was developed as part of the application Gorilla that Facebook developed to handle their monitoring needs. Gorilla value model type is a lossless value model type that works on time series data given as a 3 item tuple (*Key, Timestamp, Value*). The key is a string and serves to uniquely identify time series (this is the same as the *tag* described in our time series definition shown in **Definition 2.3.2**). The key is ignored for the remainder of this description as it is not relevant for understanding the compression algorithm. Both the timestamp (a long) and value (a double) are expected to use 64 bits.

The timestamp and value are split and compressed as two streams which we refer to as two chains. A chain, therefore, consists of either all timestamps (including the block header) or all values measured. The chains are essential in optimizing the compression. Both compression of timestamps and values utilize a 'variable length encoding' as a key feature in allowing a reduced amount of storage space to be used. The time stamp compression of Gorilla is explained in **Appendix A**.

For compressing the values, Gorilla has opted to store the XOR value of the previous value and the current value using a variable-length encoding scheme. The algorithm itself can be seen in **Listing B.2**.

1. The first value is stored with no compression
2. If XOR with the current value and the previous value is zero (same value), store single '0' bit
3. When XOR is non-zero, calculate the number of leading and trailing zeros in the XOR, store bit '1' followed by either a) or b):
  - (a) (Control bit '0') If the block of meaningful bits falls within the block of previous meaningful bits, i.e., there are at least as many leading zeros and as many trailing zeros as with the previous value, use that information for the block position and just store the meaningful XORed value.
  - (b) (Control bit '1') Store the length of the number of leading zeros (also referred to as LZ) in the next 5 bits, then store the length (also referred to as L) of the meaningful XORed value in the next 6 bits. Finally, store the meaningful bits of the XORed value.

**Listing B.2:** An algorithm describing gorilla value compression taken from [8]

An example can be seen in **Table B.1**. To make the example easier to follow, the values, as well as bit representation, are based on a regular integer representation (here using 8-bit unsigned), because floating-point bit representation will unnecessarily complicate the example. In practice, the values are, however, stored as double values. Note that this simplification will make the example appear worse, that is to say, that the compression will not be as high as in normal use.

In **Table B.1** 4 values are compressed using the different cases from the algorithm **Listing B.2**. For following happens for the different values:

- The first value is simply stored as-is.



- The second value is outside our current range (as there is not range) and is therefore stored using case 3B which is where first the control bits (CB) are stored followed by 5 bits used to store the amount of leading zeroes (LZ) of the XOR'ed values. Then 6 bits are used to describe the length (L) of the significant bits followed by said significant bits of the XOR'ed value i.e. (1111111).
- For the third value, the amount of meaningful digits is the same as for the 2nd value, therefore this is case 3A and we therefore simply only store the significant bits by reusing the previous range.
- For the last value, the XOR'ed value is 0 and thus this value is saved as a single 0 bit and is thus the case where the highest amount of compression is reached.

Value	Binary value	XOR	CB	LZ	L	Signif bits	Case
62	00111110	-	-	-	-	0011 1110	1
65	01000001	0111 1111	11	00001	000111	111 1111	3B
60	00111100	0111 1110	10	-	-	111 1110	3A
60	00111100	0000 0000	0	-	-	-	2

**Table B.1:** Example of gorilla value compression

## C | User-configurable parameters

*IrregularDB* offers several features and parameters that can be set by the user. A detailed overview of the configuration options is shown in **Table C.1**.

Key	Values	Description	Default
workingset	Unsigned integer	Amount of working sets used in the system	1
source.socket.port	Port no.	Server socket port where time series data producers can get a TCP connection to stream data to. Can be any unsigned integer representable with 16 bits	4672
source.csv	Path	File path to CSV files for ingestion from local files	None
source.csv.delimiter	Delimiter used in CSV	The delimiter used in the csv source data	,
model.timestamp.types	REGULAR, SIDIFF, DELTADelta	The timestamp model types to use during ingestion for compression	All timestamp model types
model.timestamp.threshold	Non-negative integer	The default threshold for timestamp model types	0
model.value.types	PMC_MEAN, SWING, GORILLA	The value model types to use during ingestion for compression	All value model types
model.value.error_bound	Non-negative float value	The default error bound for value model types	0.0
model.length_bound	Unsigned integer	The length bound for the maximum amount of data points that can be in a segment	400
model.value.error_bound.strict	Boolean	Specifies if reconstructed values are allowed to deviate further than error bound (see <b>Section 3.2.1</b> )	true
model.segment.compute.summary	Boolean	Specifies if summary information is calculated and inserted in database tables.	false
database.jdbc.connection_string	Connection string	Database connection string	None
model.picker	GREEDY, BRUTE_FORCE	Model picker configuration	BRUTE_FORCE
workingset.max_buffer_size_before_throttle	Non-negative integer	The size of a working set's buffer before reader threads are told to sleep.	1 000 000

receiver.throttle_sleep_time	Non-negative integer	How many milliseconds receivers should sleep after max buffer size is reached.	50
database.jdbc.batch_size	Non-negative integer	The amount of segments batched before writing to the database	100
segment.max_length	Non-negative integer	The max length that all model types have to respect	40 000

**Table C.1:** Detailed overview of user-configurable parameters

Individual time series can override the default threshold and error bound values by adding the tag on the key e.g. `'model.timestamp.threshold.{timeSeriesTag}=10'`.

For configuration keys where several values can be specified the values must be comma separated. Currently the three configuration keys where this is allowed are `'source.csv'`, `'model.timestamp.types'` and `'model.value.types'`.

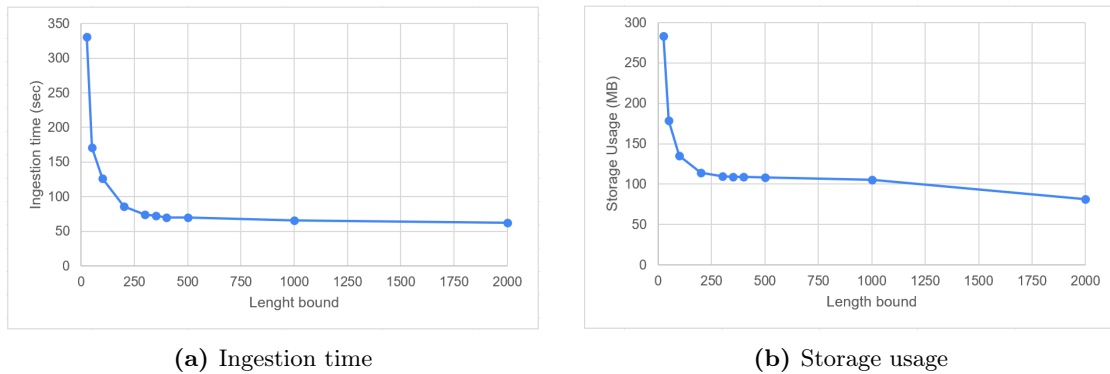
Only a database connection string is required for the system to run. CSV sources are optional and naturally do not have a default value. All other settings have default values.

# D | Initial Testing

This chapter describes some of the initial tests that were conducted in order to determine, which values to use for some of the different configuration parameters described in **Appendix C**.

## D.1 Length bound test

An initial test was done in order to determine a default value for the length bound used in *IrregularDB*. This testing was done by ingesting the entire REDD dataset (see **Section 5.2.1** for more info on REDD) and measuring the ingestion speed and compression ratio. Measuring these two metrics allows us to determine, at which point choosing a higher length bound would provide very little increase in compression ratio and ingestion speed. The results of this test can be seen in **Figure D.1**.



**Figure D.1:** Results of initial testing of different length bounds

A default value of  $L = 400$  was chosen for the length bound as this seems to be around where the curve flattens out. Because the ingestion speed for  $L = 350$  was measured to be  $72.413s$ , whereas it for  $L = 400$  was  $70.266s$  giving an improvement of  $2.147s$ . Then further increasing the length bound to  $L = 500$  only gives an improvement of  $0.416s$  as the ingestion speed for  $L = 500$  is  $69.85s$ . Similar results can be observed for the compression ratio.

## D.2 Brute Force vs. Greedy Model Picker test

A test was conducted in order to test the two different model selection approaches discussed in **Section 3.4.3**. In this test we chose to ingest the entire REDD dataset (see **Section 5.2.1**) using the two different model pickers. The following relevant configuration parameters were used when ingesting the data:

- **Model Types:** All `TIMESTAMP MODEL TYPES` and `VALUE MODEL TYPES`.
- **Error bound:** 10%

- **Threshold:** 100
- **Length bound:** 50

The ingestion speed of two runs for each model picker can be seen in **Table D.1**. Their storage usage can be seen in **Table D.2**.

Brute Force	Greedy
172.736 s	170.911 s
170.411 s	170.196 s
Average	
171.574 s	170.554 s

**Table D.1:** Ingestion time results

Brute Force	Greedy
193 569 571 B s	194 175 491 B

**Table D.2:** Storage usage results

As seen in the tables using the greedy approach leads to slightly faster ingestion speed at the cost of compression. However, the differences between the two approaches is almost indistinguishable as there is the following percentage differences between their values:

- **Ingestion Speed:**  $\frac{\|V_{brute} - V_{greedy}\|}{(V_{brute} + V_{greedy})/2} \cdot 100\% = \frac{\|171.574s - 170.554s\|}{(171.574s + 170.554s)/2} \cdot 100\% = 0.596\%$
- **Storage Usage:**  $\frac{\|V_{brute} - V_{greedy}\|}{(V_{brute} + V_{greedy})/2} \cdot 100\% = \frac{\|193569571B - 194175491B\|}{(193569571B + 194175491B)/2} \cdot 100\% = 0.31\%$