Support for Adjustable Sampling Interval in ModelarDB

AALBORG UNIVERSITET
STUDENTERRAPPORT

P9 Project
Group cs-21-dt-9-02
Software
Aalborg University
January 2022

AALBORG UNIVERSITET
STUDENTERRAPPORT

Fifth Year Software
Software
Selma Lagerløfs Vej 300
9220 Aalborg Øst
https://www.aau.dk/

Title

Support for Adjustable Sampling Interval in ModelarDB

Theme

Database Technology

Project period

September 2021 - Jan 2022

Project group

cs-21-dt-9-02

Participants

Esben Kaa Nedergaard
Kenneth Ljunggren Nørholm
Simon Mathiasen
Simon Teodor Manojlovic

Supervisor

Christian Thomsen

Co-Supervisors

Torben Bach Pedersen
Søren Kejser Jensen

Page number: 50 pages
Appendices: 4 (5 pages)
Project finished: 14-01-2022

Abstract:

This report covers the design as well as testing of an extension of the Time Series Management System (TSMS) ModelarDB with support for adjustable sampling interval. This extension will be referred to as ModelarDB-Dynamic. This feature allows time series to have different sampling intervals at different times. Time series data can come from many sources e.g. from the monitoring system of a windmill park.

Monitoring systems are an important tool used to detect downtime or dangerous behavior. Today high-frequency sensors, which can produce large quantities of time series data, are often used. An example company with 192 windmills produces around 5.5 TiB of data each month. Because of this, it can become prohibitively expensive to store the raw data. Therefore, simple aggregates are often used instead of storing the raw data. These aggregates, however, can remove interesting outliers.

Tests showed that ModelarDB performs better than ModelarDB-Dynamic for cases where adjustable sampling interval is not necessary. However, in ModelarDB-Dynamic's intended use case it achieved up to 6 times faster ingestion speed compared to ModelarDB.

Preface

This project is the result of four 3rd semester master students from Aalborg University, of which one is studying computer science and the remaining three are studying software. The project has been supervised by Christian Thomsen with Torben Bach Pedersen as co. supervisor. Furthermore, Søren Kejser Jensen, a research assistant from the department of computer science at Aalborg University, has served as an excellent technical consultant as the original developer of ModelarDB.

The purpose of the semester project was to gain a broad understanding of the research area of time series databases. To achieve this we have worked with an implementation of an actual time series database developed at Aalborg University known as ModelarDB. In conjunction with this work, we have read relevant scientific papers to gain knowledge vital to the development of the project. By extending an existing time series database, ample knowledge of time series databases is demonstrated.

Contents

1 | Introduction

Telematics, i.e. monitoring and remote control of systems, is becoming more common [1]. This report focuses on storing time series data, produced by telematics devices e.g. sensors, efficiently. The sensors used today can produce a large amount of data, which is often stored as time series (a time series is a representation of how a value changes over time).

The large amounts of data produced by monitoring systems can be exemplified by the data generated by a windmill company. In [2] it is stated that a single windmill has 98 sensors that produce data at a speed of $10Hz$. This results in a single windmill generating 980 data points per second. An example company with 192 windmills would therefore need to store 5.5 TiB of data each month [2]. Storing the raw sensor data can therefore be infeasible or prohibitively expensive [2].

Therefore, telematic data providers have opted to store aggregates instead of storing the raw sensor data. Storing aggregates, however, removes valuable information from the data such as outliers, that could have been detected in the raw data [2]. Therefore, compressing the data in a manner that maintains a representation close to the original data is highly desirable and can lead to the following benefits compared to storing raw data points: [3].

- Reduce the amount of storage needed, which in turn also enables storage of an increased amount of information
  - The increased amount of data enables more extensive analysis of data
- Maximize utilization of limited communication bandwidth by compressing the data before transferring it
- Enable faster data processing by evaluating queries on the compressed data

One way to compress the data is using model-based compression[1], which utilizes the fact that a series of data points can be described using mathematical models. Instead of storing the individual data point only the parameters of a given model needs to be stored.

1.1 Model-based compression

Model-based compression defines a model fitted to a time series within some error. It is then possible to reconstruct approximations of the data points in the time series by using the model. This is done by applying the model to the stored timestamps.

To compress a time series with model-based compression it is often necessary to split the time series into segments that each is compressed by their own model, in order to satisfy an error-bound constraint [2]. A *segment*, therefore, represents a subset of continuous data points, from a given time series.

[1] Model-based compression is also known as summarized compression

An example of model-based compression is a set of data points fitted to a linear function. The example can be seen in **Figure 1.1** where the time series: $TS = \langle(1s, 1), (2s, 2), (3s, 4), (4s, 2), (5s, 2), (6s, 1)\rangle$ is segmented into $S_0 = \{(1s, 1), (2s, 2), (3s, 4)\}$ and $S_1 = \{(4s, 2), (5s, 2), (6s, 1)\}$. By splitting the times series into two segments and creating two models the original data points can be reconstructed with better accuracy compared to trying to fit the entire series in one linear model.

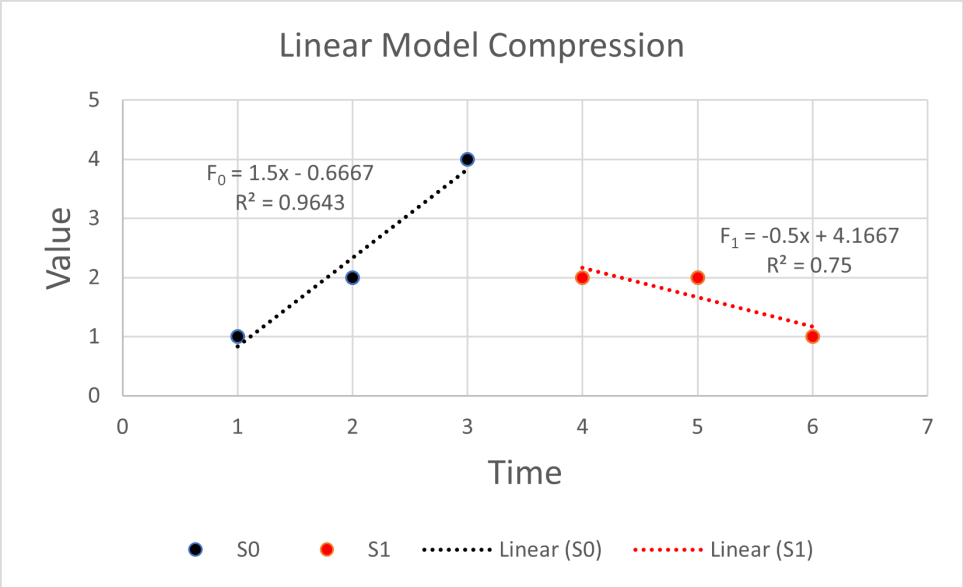


Figure 1.1: An example of models fitted to data points

The segments are compressed to the models $f_0(x) = 1.5x - 0.667$ and $f_1(x) = -0.5x + 4.1667$, respectively. The models are indicated by the dotted lines with a precision of 0.96 and 0.75, respectively. The parameters (e.g. $a = 1.5$ & $b = 0.667$ for S_0) as well as the sampling interval, start time, and end time can then be stored as opposed to storing the raw data points.

In this project, we will continue the development of ModelarDB which is a Time Series Management System (TSMS), which utilizes the model-based compression algorithms explained in the next section.

1.2 Compression model types

The theory behind the predefined model types used in ModelarDB will be described in this section.

1.2.1 PMC-mean model

The PMC(Poor man's compression)-mean model [4] is a simple model constructed as the average of a set of points [4]. Since the model represents the average, it is a constant model. New points are continuously sampled until the mean of all the current points is more than an error distance ϵ away from either the observed minimum or maximum point, and thus it is an online algorithm [4].

The PMC-mean model is different from the R-squared error method used in **Section 1.1**. The PMC-mean model considers each point's distance to the average model whereas the R-squared method considers all the points at once, therefore the R-squared method can return an r-value close to 1 even with few outliers. In the context of model-based compression, the PMC-mean approach is more sensible since we wish to be able to reconstruct the original points as close to their original value as possible. If using the R-squared error method, outlier data points could be included in a model and therefore not be detected as an outlier and not reconstructed accurately, meaning it would not represent the original values.

Example

As a simple example consider the following time series:

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

Say that the user defines an error-bound $\epsilon = 5\%$. When constructing segments from TS we calculate the average value and calculate the error for the min and max point in TS. In case both are within the error-bound of ϵ the model can be constructed.

To give an example of applying this model, the error for the **first five points** is calculated to see if a model can be constructed from 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 model 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 5% the data point at time $600ms$ cannot be included in the model.

1.2.2 Swing Filter

Swing Filter [5] 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 ϵ, 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 ← []
2  (t_0, v_0) ← getNext()
3  (t_1, v_1) ← getNext()
4  R_0 ← (t_0, v_0) // Make a recording
5  previous_recordings.add(R_0)
6  // Start a new filtering interval
7  lower_1 ← a line passing through: R_0 and (t_1, v_1 − ε)
8  upper_1 ← a line passing through: R_0 and (t_1, v_1 + ε)
9  k ← 1
10 while TRUE do
11   (t_next, v_next) ← getNext()
12   if (t_next, v_next) is NULL or (v_next < lower_k(t_next) − ε) or (v_next > upper_k(t_next) + ε)
13     // The point is null or outside allowed deviation
14     // Make a new recording
15     R_k ← (t_k, v_k), such that t_k ← 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) ← a line passing through: R_k and (t_next, v_next − ε)
21     upper_(k+1) ← a line passing through: R_k and (t_next, v_next + ε)
22     k ← k + 1
23   else // The point was inside allowed deviation
24     if v_next > lower_k(t_next) + ε
25       Swing lower_k up such that it passes through R_k−1 and (t_next, v_next − ε)
26     if v_next < upper_k(t_next) − ε
27       Swing upper_k down such that it passes through R_k−1 and (t_next, v_next + ε)
```

Listing 1.1: Swing Filter Algorithm, inspired by [5]

The algorithm for the Swing filter method can be seen in **Listing 1.1**. 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 ϵ. The same occurs for $lower_1$ but the ϵ 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 model. The equations can be found in [5]. 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 ϵ within the error bounds. If this is the case then the bounds are "swung" to fit them within the distance of ϵ, 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

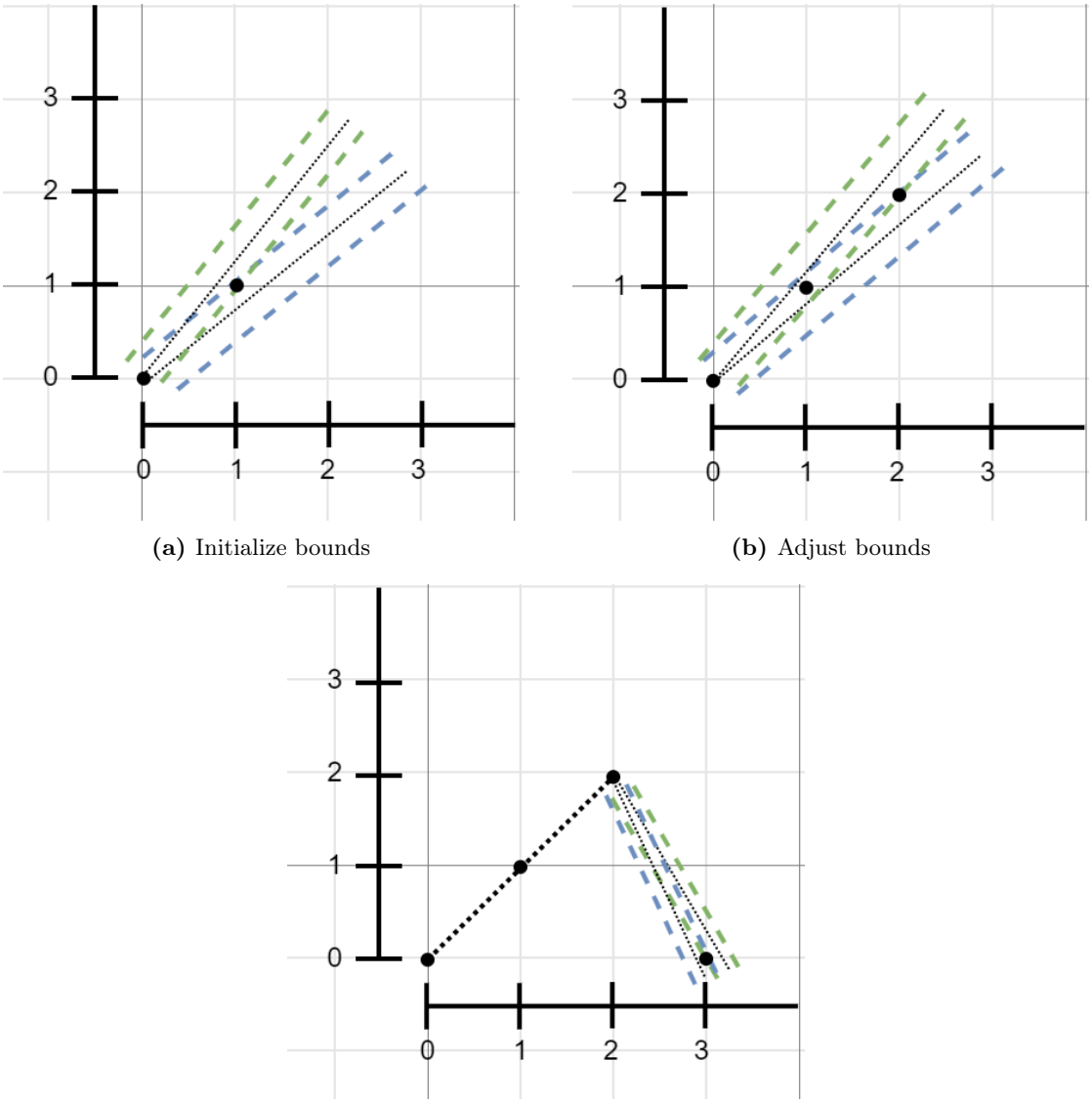
Consider the data points: $\langle(0,0),(1,1),(2,2),(3,0)\rangle$ and an error-bound of $\epsilon = 0.1$. The Swing Filter first makes a recording R_0 at $(0, 0)$. It then creates an lower bound $lower_0(t)$, which should pass through R_0 and $(1, 1 - 0.1)$ and a upper bound $upper_0(t)$, which passes 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 1.2a**, 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 1.2b**. The data point is within the upper and lower bounds as none of the conditions checked on line 12 are true because $v_{next} < lower_k(t_{next}) - \epsilon \Rightarrow 2 < (0.9 \cdot 2 + 0) - 0.1 \Rightarrow 2 < 1.7$ for the lower bound and $v_{next} > upper_k(t_{next}) + \epsilon \Rightarrow 2 > (1.1 \cdot 2 + 0) + 0.1 \Rightarrow 2 > 2.3$ 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 $v_{next} > lower_k(t_{next}) + \epsilon \Rightarrow 2 > (0.9 \cdot 2 + 0) + 0.1 \Rightarrow 2 > 1.9$ 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 $upper_0(t) = 1.05 \cdot t + 0$

Then the data point $(3, 0)$ is read as seen in **Figure 1.2c**. This point is not within the lower bound since $v_{next} < lower_k(t_{next}) - \epsilon \Rightarrow 0 < (0.95 \cdot 3 + 0) - 0.1 \Rightarrow 0 < 2.75$ making at least one of the conditions on line 12 true. The current segment is therefore finalized and a new

recording that minimizes the mean square error E_k is made before starting a new interval, which is illustrated on **Figure 1.2c**.



(a) Initialize bounds

(b) Adjust bounds

(c) Start new interval

Figure 1.2: Example of three steps of swing

In the case of ModelarDB, 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 ModelarDB uses multi-model compression it will create a line segment only for one interval. The compression ratio of this line segment is then compared to the other compression models.

1.2.3 Gorilla Compression

As part of ModelarDB, the compression model from Gorilla [6] is used. To get a better understanding of this compression model this section will describe the Gorilla algorithm conceptually and with an example.

Gorilla compression was developed as part of the application Gorilla that Facebook developed to handle their monitoring needs. Gorilla compression is a lossless compression algorithm 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. The key is ignored for the remainder of this description as it is not relevant for understanding the compression algorithm. Both the timestamp and value 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 compressions of timestamps and values utilize a 'variable length encoding' as a key feature in allowing a reduced amount of storage space to be used. First, the compression of timestamps is considered.

For compression of the timestamps, it was recognized that data often arrive at a relatively stable interval [6]. 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 1.2** the algorithm for performing gorilla compression with a variable-length encoding on timestamps is described. In **Listing 1.2** 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 Δ-of-Δ 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 1.2: An algorithm describing gorilla timestamp compression taken from [6]

Table 1.1 and **Table 1.2** gives an example of how the timestamps are encoded with the above algorithm. **Table 1.1** is the block header. The specific case applied from the algorithm in

Listing 1.2 can be seen from the "Case" column. In total (though excluding the header data) these 4·64-bit timestamps are stored in 40 bits (the encoding bits: 7 as well as the value bits: 33), which is a significant compression ratio. In the Gorilla paper [6] it is stated that an approximate 12 times reduction in storage required is achieved on their data by using Gorilla compression on both the timestamps and the values.

Time	Binary
0	0000 0000 0000 0000

Table 1.1: Block header for **Table 1.2**

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

Table 1.2: Example values for timestamp compression

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 1.3**.

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 in the next 5 bits, then store the length of the meaningful XORed value in the next 6 bits. Finally, store the meaningful bits of the XORed value.

Listing 1.3: An algorithm describing gorilla value compression taken from [6]

An example can be seen in **Table 1.3**. 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 1.3** 4 values are compressed using the different cases from the algorithm **Listing 1.3**. The first value is simply stored as-is. The second value of the example is stored using case 3B which is where the length of leading zeroes of the XOR'ed values is stored in the 5 bits following the control bits. From here the value continues with 6 bits that describe the length of the

meaningful digits (000111), followed by said meaningful digits of the XOR'ed value (1111111). For the third value, the amount of meaningful digits is the same as for the 2nd value, therefore this is case 3A and 11 bits can be saved as only the meaningful digits are stored. 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	Control bits	Value bits	Case
62	00111110	-	-	00111110	1
65	01000001	0111 1111	11	00001 000111 1111111	3B
62	00111110	0111 1111	10	1111111	3A
62	00111110	00000000	0	-	2

Table 1.3: Example of gorilla value compression

This section presented the three main compression models used in ModelarDB. The next section will describe ModelarDB's Multi-Model Group Compression (MMGC) using the compression models.

1.3 ModelarDB

ModelarDB [2] is a distributed TSMS that stores time series data using models. ModelarDB is implemented as a portable library to be used with existing systems for ingestion, query processing, and storage. The core focus of ModelarDB is fast ingestion of time series data, a high rate of compression, and online querying [2]. Online querying and fast ingestion are important to the stakeholders as they want to query as fast as possible, preferably in real-time. A high compression rate is important for the stakeholders as it can help drastically reduce the cost of storing the sensor data and in [2] it is stated that the stakeholders find it infeasible or prohibitively expensive to store the raw sensor data.

Currently, ModelarDB supports regular time series. A time series is regular if the time elapsed between the data points is the same for the entire time series. In other words, the **Sampling Interval (SI)**, which is the time between two data points in a time series, remains constant. As an example consider the following two time series:

- **Regular time series:** $TS_1 = \langle(100ms, 1.0), (200ms, 2.0), (300ms, 2.5), (400ms, 2.0)\rangle$, the SI remains the same i.e. $SI = 100ms$.
- **Irregular time series:** $TS_2 = \langle(100ms, 1.0), (250ms, 2.0), (300ms, 2.5), (400ms, 2.0)\rangle$, is irregular as the time elapsed between the different data points changes.

ModelarDB also supports **regular time series with gaps**. A gap is a period where no data points are available for the regular time series. An example of an regular time series with gaps could be: $TS_3 = \langle(100ms, 1.0), (200ms, \perp), (300ms, \perp), (400ms, 2.0)\rangle$ where $\perp$ denotes no value available.

1.3.1 Multi-Model Group Compression (MMGC)

One of the main contributions of ModelarDB is to use a **MMGC** method. MMGC is a combination of the following two methods [2]:

- **Multi-Model Compression (MMC):** Multi-model Compression defines picking the model with the best compression ratio from a set of models for a segment. This results in multiple models per time series, as a time series is made up of multiple segments.
- **Model-based Group Compression (MGC):** Model-based group compression means that correlated time series are grouped in a so-called *time series group* TSG. Data points from each time series' segment are compressed as a single segment. This is done to avoid storing similar models for multiple time series.

In previous TSMS systems, only one model type could be applied when using MGC. In ModelarDB multiple model types can be chosen for each segment, which means ModelarDB supports both Multi-Model Compression and Model-based Group Compression methods. The following two sections will go into more detail about how these methods are supported.

1.3.2 Supported model types

There exists no single model type that works well in all cases. ModelarDB therefore utilizes Multi-Model Compression (MMC). To support MMC ModelarDB needs to support multiple model types so that it can pick the model with the best compression ratio. Besides having the option to define custom models, ModelarDB currently supports the following predefined models: PMC-mean which is a constant model, Swing Filter which can model linear functions, and Gorilla/Facebook which uses a clever variable-length encoding compression scheme.

Model type	Works well with arbitrary value pattern	Storage usage per segment	Supports fast aggregates
PMC-mean	No	$\mathcal{O}(1)$ (32 bits)	Yes
Swing filter	No	$\mathcal{O}(1)$ (64 bits)	Yes
Gorilla	Yes	$\mathcal{O}(n)$ where n is amount of points	No

Table 1.4: Pros and cons of model types

Table 1.4 describes the pros and cons of the model types. PMC-mean and the Swing filter require certain value patterns to obtain good compression. Examples of the different value patterns required by the model types are shown in **Figure 1.3**. The red dotted lines in the figure illustrate possible models that could be generated using the model types.

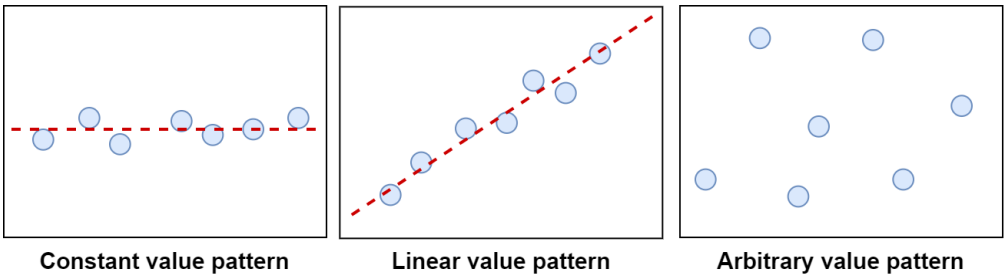


Figure 1.3: An example of different value patterns

Constant and linear value patterns are very similar as a constant value pattern can be represented using a linear model, where the slope of the curve is 0. However, in reality it makes sense to use two different model types as PMC-mean uses half the storage of the swing filter model.

Some of the models used in ModelarDB have the advantage of being able to calculate aggregates such as average in constant time as explained in **Section 2.1**.

ModelarDB uses a so called length bound constraint for the Gorilla compression model. The reason for this is that Gorilla is lossless and would otherwise consume all other data points as it would never exceed the error bound.

ModelarDB also contains a fourth fall-back model type that stores raw values. This model type is used when none of the other models can compress the data within the user-defined error-bound or the overhead of compressing the data is too high.

1.3.3 Gaps and derivations in time series group

When doing **Model-based Group Compression (MGC)** correlated time series are grouped and compressed using a single model. However, in a group, there can be gaps. Furthermore, the values from the time series in a group can diverge such that they no longer can be compressed using a single model. To handle the problem of gaps and derivations the storage schema of ModelarDB supports the storage of gaps, which enables storage of several segments for a group.

The effect of this can be seen in the example shown in **Figure 1.4**, where the segments for the time series group $TSG = \{TS_1, TS_2, TS_3\}$ are shown. At first, all of the three time series in the group can be represented using a single model so only one segment is stored and the $GAPS$ list is empty. Then, at t_A the time series TS_3 contains a gap so a new segment is generated for the group with $GAPS = [TS_3]$. At t_B all the time series produce data again, however, TS_1 deviates too much and therefore two segments are now created for the time series group TSG.

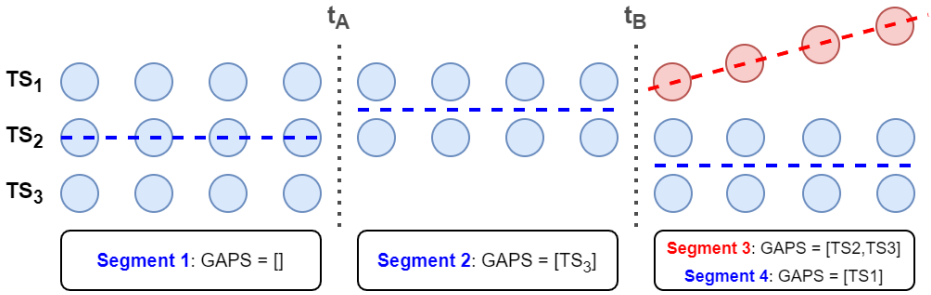


Figure 1.4: Segment examples for the time series group TSG

The following chapter will now look into some of the limitations of ModelarDB which will be the basis for possible focus areas of this project.

2 | ModelarDB limitations

ModelarDB has several limitations that originate from assumptions of the data, ModelarDB's current core focus, and the deployment of the system. These limitations are:

- **Manual view (segment vs data point view) querying:** additional effort is imposed on the users in order for them to be able to utilize the faster aggregates made possible by utilizing the models. The user currently has to direct queries to one of two views.
- **Network overload:** Currently, ModelarDB does not maximize the utilization of limited communication bandwidth because the data is not compressed before being sent to the data center. An option is to have edge devices stationed by the sensors which will compress the data on-site before sending the data to the data center for storage.
- **Fixed sampling interval:** Currently, only *a single fixed sampling interval* is used for all time series loaded into ModelarDB.
- **Only regular time series supported:** Currently, ModelarDB does not support use cases that require compression of irregular time series.

These limitations will be explained in more detail in the following sections.

2.1 Manual view (segment vs data point view) querying

Currently, the querying interface of ModelarDB provides two views [2]:

Data Point View: Enables ModelarDB to support querying of individual data points. This is done by reconstructing the data points of a segment using the model, start time, end time, and the sampling interval.

Segment View: Allows ModelarDB to do aggregate queries directly on the segments and their models without reconstructing the data points. Querying on the segment view is supported because some models such as PMC-mean and Swing enable the computing of many aggregates in constant time. For example, the average of a linear model such as Swing can be computed as $avg = \frac{min+max}{2}$. Computing aggregates in this way can greatly reduce query time as a model can represent many data points [2].

The user currently has to specify which of the two views they want to query. Instead, a single view could be provided to the users. Providing only a single view would help increase the usability of the system as users would not have to decide, which of the two views would be ideal for their query. The query processor would then be responsible for choosing the optimal view(s) to query.

2.2 Network overload

The bandwidth between the edge devices and the data center is a possible bottleneck. Therefore, it is sensible to perform the compression before transferring the data to the data center. The current setup of ModelarDB does not compress the data before transferring it from the edge device to the data center. ModelarDB is currently implemented with the goal of processing all data in a centralized location e.g. in a data center. Therefore, the implementation is built with Apache Spark [7] and Apache Cassandra [8], which enables the system to be distributed and scaled out, but this means that ModelarDB is not optimal for edge devices as some of the system resources is inevitably spent on maintaining the system across nodes. A different technology stack is therefore necessary for running ModelarDB's compression algorithm on edge devices.

Currently, development on a re-implementation of ModelarDB called *MiniModelarDB*[9] has been started. MiniModelarDB focuses on low-level optimization so that it can be used on edge devices. MiniModelarDB is implemented in Rust and uses a new technology stack with DataFusion as the query engine and Apache Arrow for in-memory storage and Apache Parquet for on-disk storage [9].

2.3 Fixed sampling interval

Currently, the ModelarDB implementation [10] does not support changing the sampling interval of a time series. This is not ideal. During critical events, it is of interest to the stakeholders to receive high-frequency data and then at non-critical periods receive low-frequency data. For example, the data from a windmill might measure with a higher frequency during start up and shutdown than during normal operations, as these periods are more likely to cause abnormalities.

Another problem is that the current implementation also does not support having two time series with different sampling intervals [10] as it was an assumption made during earlier development of ModelarDB to simplify implementation.

2.4 Only regular time series supported

ModelarDB assumes time series to be regular, meaning the data should be sampled at a fixed interval. This is a design decision that allows for further compression and optimization. Better compression is achieved because it is not necessary to store timestamps, as they can be recomputed from a segment given its SI and the start time. To save more space by removing timestamps, however, has the downside that ModelarDB cannot handle use-cases where the time series are irregular.

If it should be possible to support irregular time series in ModelarDB and still be possible to reconstruct data points, then ModelarDB would potentially need to store actual time stamps. Storing the timestamps could then be done using a method similar to the one used by Gorilla described in **Section 1.2.3**.

2.5 Problem Statement

The point of interest in this project is to handle the problem of **fixed sampling interval**. We chose this topic as we found it interesting and expect to get hands-on experience with many parts of ModelarDB.

In order to handle the fixed sampling interval problem the focus will be to add the feature adjustable sampling interval to ModelarDB, without notably decreasing the performance of its existing capabilities. Stated formally:

"*How should an adjustable sampling interval be implemented in ModelarDB while still offering a reasonable compression rate, ingestion rate, and query time?*"

3 | Design

This chapter presents the problems related to the adjustable sampling interval. This is done by presenting a description of the problems, then possible solutions, and in the end describing the chosen solution together with arguments for why a specific solution was chosen. The chapter is split into three parts. The first part will consist of an abstract design that will set the direction for the technical solution. This part spans **Sections 3.1 and 3.2**. The next part describes ModelarDB as it works today, as an understanding of the existing system is necessary to understand the changes made to it. This part consists of **Section 3.3**. Lastly, **Sections 3.4 to 3.8** will describe the concrete design of the technical solution.

3.1 Unsynchronized time series

Unsynchronized time series is a concern when using an adjustable sampling interval. Consider a time series group with two time series as seen in the example shown in **Figure 3.1**. In the figure the sampling interval of the blue time series is adjusted from 100 to 50 and then back to 100. The problem here is that the sampling interval was adjusted back to 100 at the time $T = 350$ meaning that the two time series became unsynced because even though they have the same sampling interval the time stamps will never align.

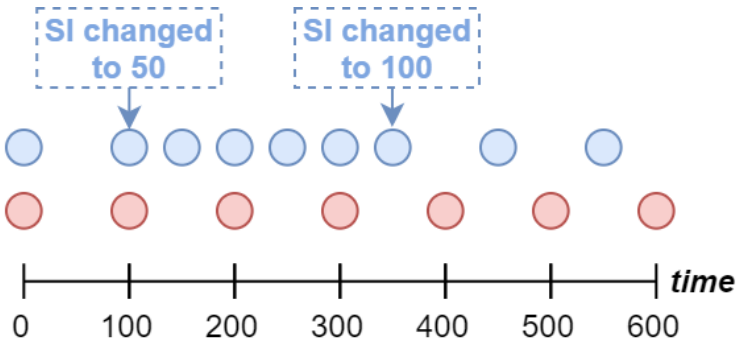


Figure 3.1: Unsynchronized time series example

3.1.1 Solution

Possible solutions to unsynchronized time series:

- **Storing deltas:** One way to handle the problem of unsynchronized time series is to use one of the time series as the base and then calculate the differences between when the base and the other time series are sampled in order to be able to reconstruct the data points for the other time series. For example if we for **Figure 3.1** use the red time series as the base then we would store $\Delta_{red} = 0$ and $\Delta_{blue} = 50$ after $T = 350$.

- **Requirement:** The problem of miss-aligned data points could be handled by the sender of the data to ModelarDB. It could be required that they always send data at timestamps such that $time$ **mod** $SI = 0$.

The problem with the first approach is that if the deltas become very large then the approach tries to group and represent data points that are in reality measured at different times. The second approach is chosen for this reason and because of its simplicity.

The time series shown in **Figure 3.2** illustrate the effect of the chosen approach. The assumption means that the data received after changing SI back to 100 now arrives at the **green** data point where $T = 400$. The reason for this is that it is the first timestamp after updating the SI where $time$ **mod** $SI = 0$.

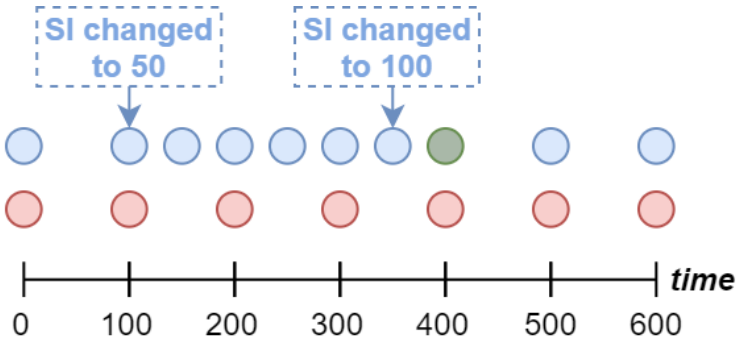


Figure 3.2: Solution to unsynchronized time series example

3.2 Definition updates

The current definition of a time series group presented in [2] does not allow adjustable sampling interval. The current definition is shown in **Definition 3.2.1**. The problem with this definition is that the first condition requires that all time series in the group should have the same sampling interval. The second condition is based on the assumption that the sampling interval remains constant, because then you only need to check the first data point of each time series in order to ensure that the time series are synchronized.

Definition 3.2.1 (Time Series Group)**:** A time series group is a set of regular time series (possibly with gaps): $TSG = \{TS_1, ..., TS_N\}$ where it holds that:

1. $\{TS_1, ..., TS_N\}$ have the same sampling interval SI
2. For all pairs $TS_A, TS_B \in TSG$ it should hold that:
   - $t_{1,A}$ **mod** $SI = t_{1,B}$ **mod** SI, where $t_{1,A}$ and $t_{1,B}$ are the first timestamp of TS_A and TS_B respectively.

3.2.1 Solution

One way to handle this problem is to remove both the conditions from the time series groups so all time series can be grouped. This leads to the updated time series group definition shown in **Definition 3.2.2**.

Definition 3.2.2 (Time Series Group)**:** A time series group is a set of regular time series (possibly with gaps): $TSG = \{TS_1, ..., TS_N\}$.

Now having removed these conditions from the time series group, the conditions now need to be managed by some other aspect in the system. Some possible solutions to handling this problem are:

- **Change the sampling interval of all time series in a group:** Instead of being able to update the SI of individual time series it could be changed so that the SI of all the time series in the group are updated thereby ensuring that they always have the same sampling interval. The idea behind this solution is that the time series in a group should be correlated, thereby implying that if the sampling interval of one of the time series should be updated then it could make sense to update the sampling interval of all of them.
- **Update the segment definition:** Ensuring that the time series represented by a model have the same SI could be done by imposing new conditions for when a segment can be created for a time series group by updating the segment definition.

We chose to go with the second solution as it would give the users more freedom as they could individually change the sampling interval of time series as needed instead of needing to update the sampling interval of the entire group. We, therefore, updated the definition for a segment to the one shown in **Definition 3.2.3**.

Definition 3.2.3 (Segment)**:** A segment S is a 5-tuple $S = (t_s, t_e, SI_S, GAPS, m)$ that represents the data points for a subset or all the time series in a time series group TSG. We have:

- t_s: is the start time
- t_e: is the end time
- SI_S: is the sampling interval of the segment.
- $GAPS$: set of time series from TSG not represented by S.
- m: is the model used to represent the data points.

We require that the following holds for the segment S:

- The sampling interval SI_{TS_i} for each time series TS_i represented by S should in the time period from t_s to t_e be equal to SI_S i.e.
  - $SI_{TS_i} = SI_S$ **for each** $\{TS_i \in TSG \mid TS_i \notin GAPS\}$

As stated the two conditions must be handled by changing the definition for a segment. This is done in the following way:

- **The first condition**, i.e. $\{TS_1, ..., TS_N\}$ have the same sampling interval SI, is handled by updating the segment definition such that time series in a segment must have the same SI.
- **The second condition**, i.e. $t_{1,A}$ **mod** $SI = t_{1,B}$ **mod** SI, is made irrelevant by the assumption made in **Section 3.1.1**, together with the updated segment definition given in **Definition 3.2.3**, that states that SI must be the same for all time series represented by a segment as the segment now holds a sampling interval.

Furthermore, we define a slice as a set of data points that each has a reference to the time series they are from as follows:

Definition 3.2.4 (Slice)**:** A slice s of a time series group TSG is a map from time series to data points (possibly with gaps), where there is one data point from each time series $TS \in TSG$. The following should hold for a slice s:
- All data points $p \in s$ need to have the same timestamp T
- The sampling interval SI of all time series represented by s should at timestamp T be the same.

3.3 Original ModelarDB's main components

Before going into details on the different design decisions that have been made to allow for adjustable sampling interval in ModelarDB some of the original core components of ModelarDB will be covered to ease the understanding of what has changed. An overview of the core components that will be discussed is shown in **Figure 3.3**.

Figure 3.3: Overview of main components that are related to ingestion. Arrows denote aggregation, '*' denotes multiplicity, and the dotted line indicates write actions

3.3.1 DataPoint & TimeSeries

A `DataPoint`[1] is a pair consisting of a timestamp and a value. Each data point belongs to a time series. The `TimeSeries`[2] class is responsible for supplying its sequence of `DataPoints` to a `TimeSeriesGroup` [3] by reading from a source (data stream) and returning the next data point when requested.

[1]**Direct link to source:** https://github.com/ModelarData/ModelarDB/blob/454ebac7f0f93943929ad4b198d7e14490ad342e/src/main/java/dk/aau/modelardb/core/DataPoint.java

[2]**Direct link to source:** https://github.com/ModelarData/ModelarDB/blob/454ebac7f0f93943929ad4b198d7e14490ad342e/src/main/java/dk/aau/modelardb/core/timeseries/TimeSeries.java

[3]**Direct link to source:** https://github.com/ModelarData/ModelarDB/blob/454ebac7f0f93943929ad4b198d7e14490ad342e/src/main/java/dk/aau/modelardb/core/TimeSeriesGroup.java

3.3.2 Time series group

A `TimeSeriesGroup` instance has a set of `TimeSeries`. The responsibility of the `TimeSeriesGroup` is to generate slices. These slices are created by taking the next `DataPoint` of each `TimeSeries` in a `TimeSeriesGroup`. Taking the next `DataPoint` from each `TimeSeries` to generate a slice can be done because time series have a constant sampling interval.

If a time series in the `TimeSeriesGroup` does not have a `DataPoint` for the current timestamp a gap data point is added to the slice. These gap data points are added by the `TimeSeriesGroup`, which detects gaps by maintaining a *nextExpectedTimestamp* value that stores the expected time for the next slice. That is to say every time a slice is returned the *nextExpectedTimestamp* is incremented with the sampling interval. This way if the next data point for a time series does not match the expected time a gap data point can be used in place of the actual data point.

3.3.3 SegmentGenerator

The `SegmentGenerator`[4] is the component responsible for performing the Multi-Model Group Compression (MMGC) and generating segments to be stored in the database. Currently, the `SegmentGenerator` is handed a reference to a `TimeSeriesGroup`, and then the work flow of the `SegmentGenerator` is as follows:

1. The `SegmentGenerator` **'pulls'** slices from the `TimeSeriesGroup` one at a time
2. The `SegmentGenerator` then tries to append the slice to the available compression models
3. If this append succeeds go back to **step 1** and pull a new slice
4. Else if the slice can not be appended to any of the models a segment is emitted. Then, if the compression ratio of the newly emitted segment becomes too low a **'split'** can occur because compression ratio is used as a *split-heuristic*.

The `DataPoints` of the `TimeSeries` can become temporarily uncorrelated due to various circumstances/events [2]. If the data is no longer within a user-defined error-bound a `SegmentGenerator` can 'split' the `TimeSeriesGroup` into several `TimeSeriesGroup` instances, where the data within these new groups is correlated. Some models, e.g. the Gorilla compression model [6], can represent the data no matter how uncorrelated it becomes. The reason for this is that the models are lossless therefore the error is always 0, but the cost of using these models is a poor compression ratio. Poor compression ratio is therefore used as the heuristic for when to perform a split of the time series, by having ModelarDB perform a split when the compression ratio is below some configurable fraction (default is 1/10.0) of the average compression ratio.

This split can happen all the way down to each individual `TimeSeries` being placed in their own group. It must be stressed that this does not create new actual `TimeSeriesGroups` with new group ids (GIDs), but rather clones of the original group are created. The clones have a subset of the original `TimeSeries`' ids (TIDs); the other TIDs are placed in a list of gaps for a given segment indicating that this specific segment does not apply to the time series with ids in the gaps list.

[4] **Direct link to source:** `https://github.com/ModelarData/ModelarDB/blob/454ebac7f0f93943929ad4b198d7e14490ad342e/src/main/java/dk/aau/modelardb/core/SegmentGenerator.java`

These split `TimeSeriesGroup`s are put in their own `SegmentGenerator` such that a `SegmentGenerator` instance only works with data that can be represented by a single model. This introduces some complexity into the `SegmentGenerator` in that these split `TimeSeriesGroup`s can also be joined again, and in all instances data needs to be pulled from the correct `TimeSeries` and the buffered data must be correctly copied and updated between the child `SegmentGenerator`s.

The following sections will now go in-depth with the design decisions that have been made for this project.

3.4 Reconstruction of data points from segments

As explained earlier in **Section 2.1**, ModelarDB provides users with a *data point view*, which allows the end-user to reconstruct data points from the segments by using the Sampling Interval (SI). The current storage schema used by ModelarDB stores the SI on the time series as it remains constant for the entire time series. However, this is not the case when working with adjustable SI as the SI can vary for a time series.

3.4.1 Solution

The chosen solution was to sacrifice some storage space by storing the SI on the segments instead of the time series to allow different segments for a time series to have different sampling intervals. This is done by updating the storage schema for ModelarDB described in [2] so that it stores the sampling interval on segments in order to fulfill the new definition of segments from **Definition 3.2.3**. The updated storage schema is shown in **Figure 3.4**.

In the example shown in **Figure 3.4** the changes are marked with red. In the example, we have four total segments. The first three segments are from the time series group 1. The first segment entry models data from both time series 10 and 11. After the first segment was finalized, the sampling interval for time series 11 changed from 100 to 500 resulting in the group splitting since the sampling intervals are different, and as such they can not be represented by the same segment as stated in **Definition 3.2.3**. Time series 10 continues generating segments with the same sampling interval (second entry) with time series 11 as gap, but time series 11 continues generating segments with 500 as sampling interval and with time series 10 as a gap (third entry).

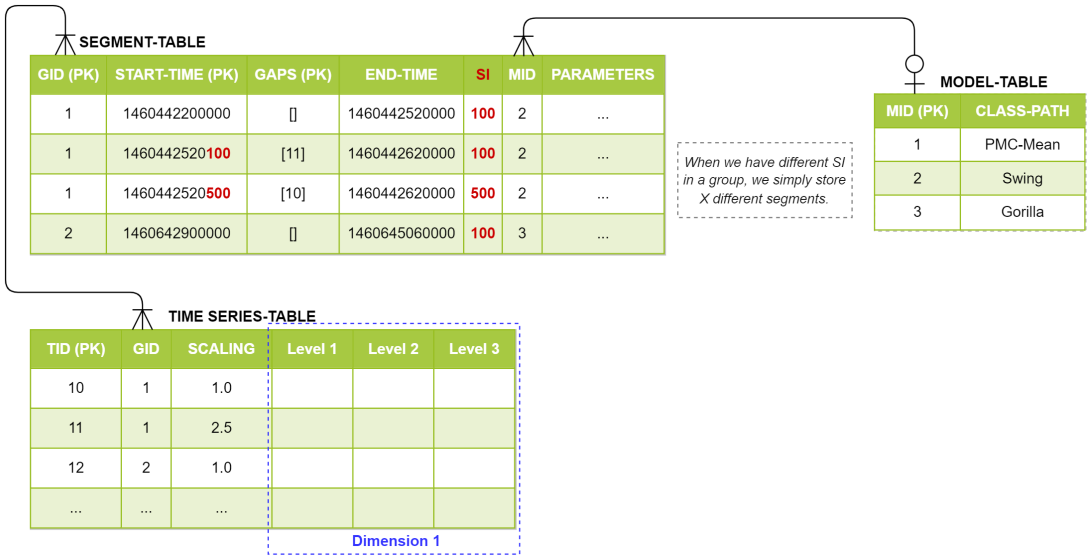


Figure 3.4: Updated storage schema

No changes were made to the model table. The model table still contains a model id (**MID**) and a path to the class containing the implementation of the model. The segment table now contains the columns described in **Table 3.1**. Furthermore, the time series table contains the columns described in **Table 3.2**.

Column	Description
GID	An integer representing the ID of the time series group that the segment represents.
START-TIME	The start time of the segment stored as a timestamp.
GAPS	List of the time series from the group, that are not represented in this segment.
END-TIME	The end time of the segment stored as a timestamp.
SI	New column used to store the sampling interval of the segment as an integer on the segments themselves.
MID	An integer pointing to the model type used for the segment.
PARAMET-ERS	The parameters for the model stored as a BLOB-object (e.g. $a = 1.5$ & $b = 0.667$ for S_0 shown in **Figure 1.1**)

Table 3.1: Segment storage schema description

Column	Description
TID	Unique integer used to identify the time series.
GID	Id of the group that the time series belongs to stored as an integer
SCALING	A double used to store the scaling factor that should be applied to the models in order to get data points for the time series. Is for example used during query processing.
DIMENSIONS	User-defined dimensions

Table 3.2: Time series storage schema description

3.5 How SI configurations are read

The sampling interval is currently static and defined in a configuration file. Therefore, it cannot be changed during run-time. How can we allow the SI to be adjustable for a `TimeSeries`[5]? Our proposed solutions for handling an adjustable sampling interval are:

- **Detecting change of sampling interval automatically**: Change of sampling interval can be detected automatically, a partial detection can be made by detecting the `DataPoint`s [6] that no longer conform to the rule of $(timestamp \textbf{ mod } si) = 0$. An example of this solution and its challenges is explained later in this section.
- **Sending a message over the data stream**: An outside mechanism could be responsible for sending a configuration point in the stream of data points. In this case, ModelarDB needs to be modified to be able to recognize the difference between `ValueDataPoint`s and `ConfigurationDataPoint`s. Whenever a `ConfigurationDataPoint` is recognized, ModelarDB will know to finish the current segment and start a new segment with the updated SI.
- **Message to ModelarDB outside the data stream**: Again an outside mechanism would be responsible for sending a message to ModelarDB conveying that a given `TimeSeries` will now send data points with an updated SI. In this case there is a synchronization problem in terms of getting the message of an updated SI and actually receiving `DataPoint`s with the updated SI. Consider that the message arrives after the data stream has begun sending data points with an updated SI.

Automatically detecting the changes in sampling intervals can be challenging since we also have the option for gaps. The question is whether a gap has occurred or the sampling interval was changed. Consider the time series $ts_1 = (0, 1), (50, 2), (100, 3), (200, 4), (300, 5)$. Rather than having a gap at $(150, value)$, a possible change in SI is detected from 50 to 100. However, to detect whether it is a gap or a change in SI, we need an assumption for the number of data points in a row with the same time interval between them. The assumption is needed because we can never be sure if it is an actual change or a bad luck of gaps. In the above example we could have that between the point at time 100 and time 200 there is a gap, and the same happened for the data points at time 200 and 300. This process as a whole would be more fitting if data points were ingested in batches. Then, we would have the option to look ahead in the data stream to decide if we should model gaps or change the sampling interval.

3.5.1 Chosen approach

It was chosen to model the change of sampling interval as a message over the data stream, where `ValueDataPoint`s are retrieved from. The primary reason for this is that it removes the problem of synchronization. As such the data stream can contain `ValueDataPoint`s on the form: (timestamp, value), and `ConfigurationDataPoint`s on the form: (configuration key, value). An

[5] **Direct link to updated source:** https://github.com/ModelarDB-Dynamic/ModelarDB-Dynamic/blob/SegmentGeneratorController/src/main/java/dk/aau/modelardb/core/timeseries/TimeSeries.java

[6] **Direct link to updated source:** https://github.com/ModelarDB-Dynamic/ModelarDB-Dynamic/blob/SegmentGeneratorController/src/main/java/dk/aau/modelardb/core/model/DataPoint.java

example of a `ConfigurationDataPoint` that would change the sampling interval to 500 is: ("SI", 500).

In the old system the `TimeSeriesGroup`[7] was responsible for handling the *nextExpectedTimestamp* for the next `DataPoint` as explained in **Section 3.3**. The *nextExpectedTimestamp* is used to determine if a gap has appeared, this is done by comparing the timestamp of an arriving data point with the *nextExpectedTimestamp*. If the data point has a timestamp that is bigger than the *nextExpectedTimestamp*, we know that there has been a gap point before the data point arrived.

We move this functionality to the `TimeSeries` class as `TimeSeries` in a group may have different SIs. When a `ConfigurationDataPoint` is received, the `TimeSeries`' current sampling interval and *nextExpectedTimestamp* is updated. The *nextExpectedTimestamp* points to the timestamp for the next expected `ValueDataPoint`. This timestamp therefore has to be updated to be able to handle the new sampling interval. The *nextExpectedTimestamp* is initialized as NULL before any `valueDataPoint` is read. The *nextExpectedTimestamp* is therefore updated following the approach shown in **Listing 3.1**:

```
1  if datapoint is ValueDataPoint
2      if nextExpectedTimestamp is NULL
3          nextExpectedTimestamp ← datapoint.getTimestamp()
4      nextExpectedTimestamp ← nextExpectedTimestamp + currentSamplingInterval
5
6  if datapoint is ConfigurationDataPoint
7      newSI ← datapoint.getNewSamplingInterval()
8      if nextExpectedTimestamp is not NULL
9          nextRolledBack ← nextExpectedTimestamp − currentSamplingInterval
10         difference ← newSI − (nextRolledBack  mod  newSI)
11         nextExpectedTimestamp ← nextRolledBack + difference
12     currentSamplingInterval ← newSI
```

Listing 3.1: Update of nextExpectedTimestamp and sampling interval

It is assumed that each `TimeSeries` starts their data stream with a configuration point to specify its sampling interval. A default sampling interval is therefore always set using the value from the configuration file before any data is read from the stream.

Since the *nextExpectedTimestamp* is computed for the next data point, we first have to roll the *nextExpectedTimestamp* back to the previous value whenever a `ConfigurationDataPoint` is met by subtracting the previous sampling interval (line 9). It is necessary to compute the difference (line 10) in order to avoid the time series becoming unsynchronized as mentioned in **Section 3.1**. Then in the end this difference is added to the rolled back expected timestamp to get the new expected timestamp (line 11).

As an example consider a previous timestamp of 150, $SI_{old} = 50$, and $SI_{new} = 75$. Since the *nextExpectedTimestamp* points ahead to the next expected `DataPoint`, it currently points to 200 i.e. $next_{old} = 200$. So the first action is to roll it back to $next_{rolled-back} = 200 - 50 = 150$. Next, the difference is calculated as $difference = 75 - (150 \textbf{ mod } 75) = 75 - 0 = 75$. Finally,

[7] **Direct link to updated source:** `https://github.com/ModelarDB-Dynamic/ModelarDB-Dynamic/blob/SegmentGeneratorController/src/main/java/dk/aau/modelardb/core/GroupBasedCompression/TimeSeriesGroup.java`

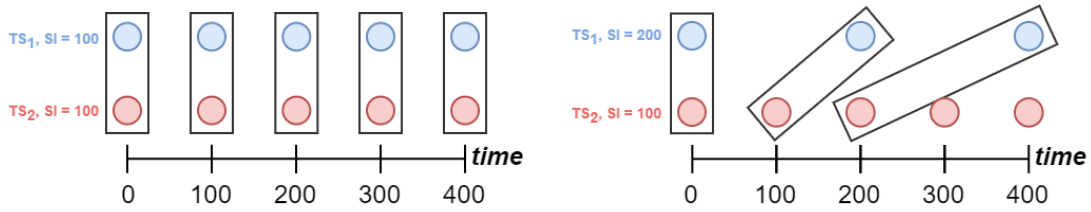
the difference can be added to the rolled back expected timestamp $next_{new} = 150 + 75 = 225$.

A limitation of this algorithm is that two configurations data points cannot be allowed to appear immediately after each other in the data stream. Given that the *nextExpectedTimestamp* is incremented with the current SI when a new `ValueDataPoint` is read, but recomputed as above when a configuration `DataPoint` is read, then the second configuration `DataPoint` will roll itself further back than the previous timestamp. This problem could be handled by an implementation workaround. Two `ConfigurationDataPoints` are currently only allowed if and only if no `ValueDataPoints` have been loaded, as it is dependent on *nextExpectedTimestamp* (line 8), which is first set when a `ValueDataPoint` is met (line 3).

3.6 Slice generation

In the old implementation of ModelarDB the `TimeSeriesGroup` is responsible for generating slices as explained in **Section 3.3.2** by taking the next `DataPoint` from each `TimeSeries`. **Figure 3.5a** shows an example of how the old slice generation works for two `TimeSeries` with the same sampling interval. The rectangles denote slices.

The old slice generation approach will, however, not work when using adjustable sampling interval, because the next `DataPoints` can get unsynchronised as seen on **Figure 3.5b**. This is because the old approach is made for a static sampling interval and can therefore assume that `DataPoints` always appear at the same timestamp. The figure shows how the old approach for slice generation would no longer work as two `TimeSeries` with different sampling intervals gets offset in regards to their timestamps. The generated slices would also not uphold the requirements for slices specified in **Definition 3.2.4** as the data points would have different timestamps and the SI of the time series represented by the slice would be different.



(a) Same SI for all `TimeSeries` **(b)** Adjustable SI enabling different SIs

Figure 3.5: Example of slice generation

The approach for generating slices should therefore be updated to ensure that the slice contains `DataPoints` from all `TimeSeries` that have their next data point at the same timestamp with the same sampling interval instead of containing a `DataPoint` from all `TimeSeries` in the `TimeSeriesGroup`. This means that the new approach should produce slices similar to the ones shown in **Figure 3.6**.

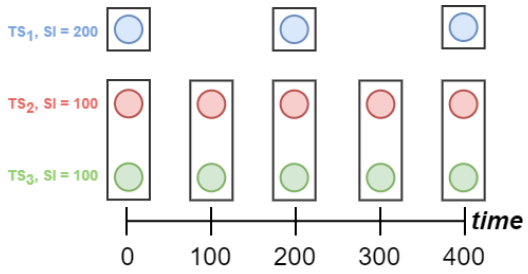


Figure 3.6: Expected slices

3.6.1 Priority Queue Solution

A priority queue can be used to ensure that the expected slices are constructed by sorting the `DataPoints` on timestamp and the sampling interval. This sorting makes it possible to generate slices that only contain `DataPoints` with the same timestamp and the same sampling interval.

Here, the steps of the algorithm, shown in **Listing 3.2**, for generating slices are described, with a reference to where this can be seen in the listing:

1. **Dequeue:** first an element from the priority queue is dequeued and added to the slice as shown on line **5-6**.
2. **Enqueue new point:** Whenever a data point is dequeued from a time series the next data point from that time series is immediately enqueued to maintain a priority queue consisting of one data point per time series as shown on line **8-10**.
3. **Loop step 1 - 2:** The slice is then constructed by continuing this dequeueing and enqueueing until a data point with a different timestamp or sampling interval than the current data point is met as shown on line **12-13**.

```
1  let slice be a list of data points initially empty
2  let queue be a priority queue of data points sorted on timestamp and sampling interval
3
4  while (!slice_done) do
5      x ← queue.dequeue()
6      slice.add(x)
7
8      next ← get_next_data_point_for_time_series(x.tid)
9      if next is !NULL then // next can be NULL if the time series is done
10         queue.enqueue(next)
11
12     if queue.empty() or has_different_si_or_time_stamp(queue.peek(), x) then
13         slice_done = TRUE
14
15 return slice
```

Listing 3.2: Pseudo code for constructing a slice

It was chosen to use a priority queue to generate slices because it is an efficient data structure. Priority queues are efficient as the complexity of enqueue and dequeue is $\Theta(\log(n))$ where n is the amount of elements in the queue [11, p.163-164]. The slices constructed using the priority queue are then returned to the `SegmentGeneratorController` described in **Section 3.8**.

3.7 Segment generation approach

The current implementation of the `SegmentGenerator` class is roughly 600 lines and consists of procedures that change state. Furthermore, the `SegmentGenerator` class is not only responsible for generating segments, but also for splitting and joining time series groups (described in **Section 3.3**). Having only procedures that modify state makes it challenging to understand the class and extend or change the functionality. Ideally, this functionality should be separated so that the `SegmentGenerator` is only responsible for generating segments and another component is responsible for joining and splitting time series groups. Also, given that the segment generator is a large complicated class, minimizing the changes in this class is desirable.

3.7.1 Idea to treat the SegmentGenerator as a black box

We initially decided to treat the `SegmentGenerator` class as a black box and not change the existing functionality. The idea was that the `SegmentGenerator` class should already be able to generate segments, split, and join as long as slices are correctly passed to it. The only difference would be to change the pull-based approach (**Section 3.3.3**) to a push-based approach. A new controller component would then distribute slices to each `SegmentGenerator` instance according to the sampling interval (one `SegmentGenerator` instance for each sampling interval that the time series currently have during ingestion). The huge advantage of this approach is to completely ignore the complexity of the `SegmentGenerator` class and instead add an abstraction layer in form of the controller class.

After a closer look at the `SegmentGenerator` class, in relation to implementing the push-based approach, we found that a larger refactor would be necessary anyway because the time series group and individual time series were parsed between all the procedures. This resulted in us needing a full understanding of the `SegmentGenerator` class anyway in order to refactor the class to the push-based approach. In the old system, the `SegmentGenerator` had an instance of a `TimeSeriesGroup` that was used to get the next data slice. Whenever this group was split, subsets of the `TimeSeriesGroup` were created and these instances were parsed between procedures. All places that used the `TimeSeriesGroup` had to be refactored to use a push-based approach.

3.7.2 Push-based SegmentGenerator

We decided to give slices to the `SegmentGenerator` instead of having the generator ask for slices as explained in **Section 3.3.3**. This was done to maintain a clear separation of concerns. When a `TimeSeriesGroup`'s `TimeSeries` become uncorrelated, the `SegmentGenerator` spawns leaf `SegmentGenerator`s. All `SegmentGenerator`s for a given `TimeSeriesGroup` have the same reference to each other through a list. If there is only one `SegmentGenerator` it will consume a slice itself

otherwise it partitions the slice into sub-slices and forwards the sub-slices to the leaf segment generators.

Each `SegmentGenerator` also maintains a set of permanent gaps. Permanent gaps refer to the time series for which a `SegmentGenerator` is not producing segments for i.e. when a split has happened in the `TimeSeriesGroup` then the time series ids that are assigned to a given leaf segment generator are assigned as permanent gaps to all other leaf segment generators for the `TimeSeriesGroup`.

3.8 Segment generator controller

The goal of this project is to have time series with possible different SIs and to store the SI on segments. To achieve this, the segment generator class must be concerned with SI. However, the current implementation of `SegmentGenerator` is not.

In an effort to keep the `SegmentGenerator` functionality the same we decided to separate the concern for handling change of SI from the `SegmentGenerator` to a new class, the `SegmentGeneratorController`, which maintains multiple `SegmentGenerator`s. Since a `SegmentGenerator` creates segments for a `TimeSeriesGroup`, a `SegmentGeneratorController` exists for every `TimeSeriesGroup`. Given the need to reconstruct data points, a single segment must have a single SI. The solution to this is that every `SegmentGenerator` for a `SegmentGeneratorController` has a unique SI, that can be saved with any generated segments.

The first purpose of the `SegmentGeneratorController` is to maintain a set of active `SegmentGenerator`s, their SI, and what `TimeSeries` ids (TIDs) they process. The second purpose is to delegate slices to the correct `SegmentGenerator`s. In order to maintain the set of `SegmentGenerator`s, the change of SI must be detected. In order to detect these changes in SI we came up with the following solutions of which one will be chosen:

- **Monitor slices for changed TIDs:** One possible way to handle changes in SI is by detecting changes in the TIDs in slices. This can be achieved by performing the following steps when receiving a slice:
  1. If no `SegmentGenerator` exists for the SI of the slice, a new is created. Skip to step 4.
  2. Compare TIDs of slice to TIDs of the `SegmentGenerator` with the same SI.
  3. If TIDs do not match
     (a) The `SegmentGenerator` finalizes the current segment.
     (b) Any new TID(s) are added to the `SegmentGenerator`.
     (c) Any TID(s) that were contained in the `SegmentGenerator`, but not in the slice is removed from the `SegmentGenerator`.
     (d) The `SegmentGenerator` starts a new segment.
  4. The slice is consumed by the `SegmentGenerator`.

  The method allows moving a `TimeSeries` to a different `SegmentGenerator` depending on its current SI.
- **Handle `ConfigurationDataPoints`:** Change in SI can be detected by passing `ConfigurationDataPoints` to the `SegmentGeneratorController`. When a `ConfigurationDataPoint`

is received the following is performed:

1. Finalize the `SegmentGenerator` responsible for the previous SI
2. Initialize a new `SegmentGenerator` with any remaining time series for the previous SI
3. If there exists a `SegmentGenerator` for the new SI then finalize it.
4. Initialize a new `SegmentGenerator` with all time series for the new SI

The first approach handles the change of SI by moving `TimeSeries` based on the assumption that a slice is delegated to both `SegmentGenerator`s affected by the move. However, this is not always the case as shown in **Figure 3.7**, where the only `TimeSeries` contained in SegmentGenerator1 is moved to SegmentGenerator4. This results in SegmentGenerator1 not being delegated a slice and as such not being finalized, as opposed to SegmentGenerator4. SegmentGenerator1 will be left hanging until a slice is delegated to it. A workaround for this is to look up which `SegmentGenerator` the `TimeSeries` is moved from, and finalize it if it no longer contains any TIDs. Overall the first approach has some overhead as each slice must be screened for its TIDs.

The second approach utilizing `ConfigurationDataPoint`s is simpler given that it relies on already available data being passed on to the `SegmentGeneratorController`, as opposed to dynamically computing it. As such we will continue with the second approach.

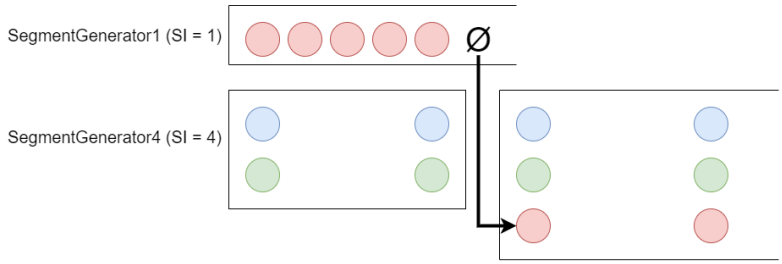


Figure 3.7: Example of a `TimeSeries` being moved to another `SegmentGenerator`, leaving the original empty

4 | Test

This chapter presents the data set used for testing, the tests themselves, and the results of the tests. We perform functionality tests that ensure that the functionality of the system is still working after having updated the implementation of multiple classes. We also conducted performance tests to identify the costs associated with implementing adjustable sampling interval. The following sections will further describe the test data as well as the tests.

4.1 Test data set

For the tests in this project, the REDD [12] time series data set, a public data set for energy research, was used. The data set consists of data for 6 houses each with up to 26 channels for different energy consumption sources, such as ovens or lighting. Specific channels are denoted by channel number and the channel name, i.e. 3.Oven. In total, the REDD data set contains $56,341,629$ data points.

To be able to use the data in both ModelarDB and ModelarDB-Dynamic, data preprocessing and cleanup is necessary as the data in the original REDD data set is not regular. Therefore, the data is made regular as both versions only support regular data. The prepossessing steps performed to make the data regular were the following:

1. The files have been individually sorted by timestamps as data points are sometimes out of order in the files.
2. Changed timestamps to always increment with one second from the previous measurement. A side effect of this is that all the time series no longer have any gaps, which will affect the absolute storage usage and ingestion speed. However, the tests are not used to demonstrate any absolute results but rather to compare the two versions of ModelarDB, and the same gap-less data is used for both systems.

4.1.1 Grouping the REDD dataset

In order to test the effects of grouped time series, we grouped together the following time series from House 1 of the REDD dataset:

- **Group One:** `3.Oven` and `4.Oven` are grouped together as they have similar values. It seems that when one of them is turned on the other will be turned on as well as seen in **Figure 4.1**.
- **Group Two:** `9.Lighting`, `17.Lighting`, and `18.Lighting` since their values are correlated in some parts of the data. A small part of the data points for the three time series can be seen plotted in **Figure 4.2**, where in the start all the time series are turned off at the same time and then `17` and `18` turn on at the same time and so on.

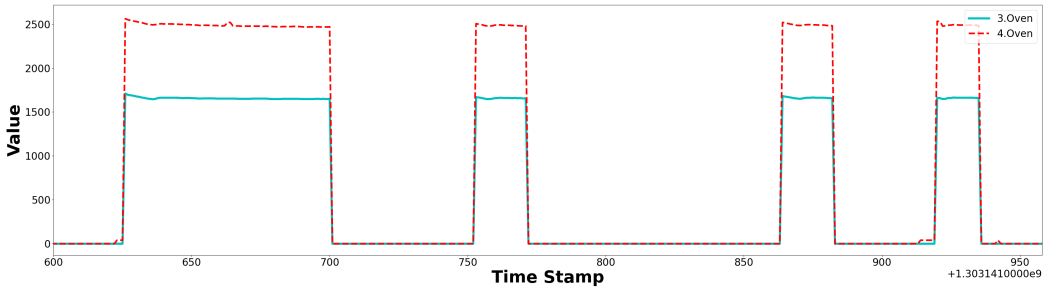


Figure 4.1: Small section of data points for time series `3.Oven` and `4.Oven`

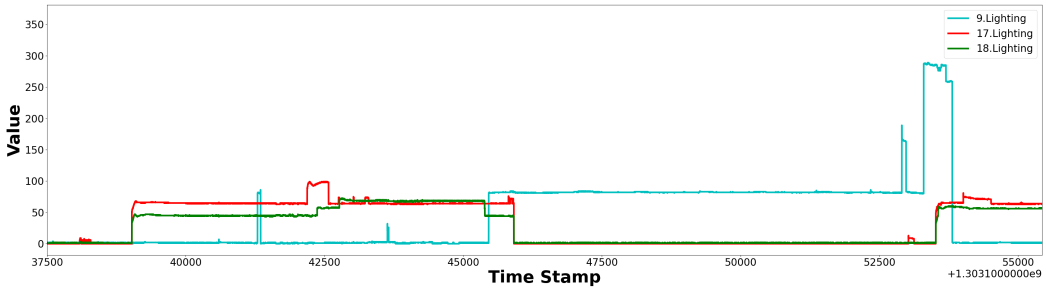


Figure 4.2: Small section of data points for the time series `9.Lighting`, `17.Lighting`, and `18.Lighting`

Scaling factors have to be calculated before data can be grouped, as two time series may follow the same pattern, but not values. Scaling factors are used to scale time series such that both values and patterns are similar.

In order to calculate the scaling factors, which should be multiplied onto the model's values for the different time series, we calculated the averages for the time series for data points that are not close to 0 e.g. when the lighting is turned on. It was chosen to ignore the values for when the devices are turned off as these are all very similar. The data points with values equal to 0, 1, and 2 are therefore ignored as it is assumed that 1 and 2 are noise values also symbolizing off.

The averages and scaling factors for the different time series are calculated to the following:

- **Group One:**
  - `3.Oven` = 195.56. Scaling Factor = 1.0
  - `4.Oven` = 318.99. Scaling Factor = $\frac{318.99}{195.56} = 1.63$
- **Group Two:**
  - `9.Lighting` = 71.78. Scaling Factor = $\frac{71.78}{45.70} = 1.57$
  - `17.Lighting` = 64.76. Scaling Factor = $\frac{64.76}{45.70} = 1.42$.
  - `18.Lighting` = 45.70. Scaling Factor = 1.0

4.2 Functionality tests

In this section, we describe the functionality tests that we created in order to verify the adjustable sampling interval feature was implemented correctly. The functionality tests are split into two parts:

- Unit tests
- Integration test

4.2.1 Unit Tests

The purpose of the unit tests is to ensure that basic functionality works as expected. The data set used for the unit tests is not the REDD data set, but some smaller manually created data set. As an example all the relevant data sets for the `SegmentGenerator` test can be seen in **Appendix B**. Using this manually created data makes it feasible for us to calculate the expected outcome so that we can compare it to the actual output. Four main components are tested using unit tests:

- **`TimeSeries`**: Since the data point reading was modified, unit tests for `TimeSeries` are created in order to be sure the data is read correctly before being passed to the ingestion part of the system. The `TimeSeries` tests are therefore a foundation for the rest of the tests. The tests were created by creating small time series that among other things tested that value data points, configuration points, and gap points are read correctly.
- **`TimeSeriesGroup`**: Tested that the priority queue solution discussed in **Section 3.6.1** generate correct slices. The testing is done by generating `TimeSeriesGroup`s with `TimeSeries` with different sampling intervals and extracting slices from the group and comparing them to the expected slices.
- **`SegmentGenerator`**: Tested that correct segments were generated (e.g. used PMC-mean for constant data) and that splitting and joining inside the `SegmentGenerator` worked as expected.
- **`SegmentGeneratorController`**: Tested that slices were delegated correctly to different `SegmentGenerator`s based on the sampling interval of the different `TimeSeries`. Also tested that receiving configuration data points move `TimeSeries`' to different `SegmentGenerator`s and close the `SegmentGenerator`s when we expect it.

Example Unit Test for SegmentGenerator

For the `SegmentGenerator` tests different `TimeSeries` were created and joined together in `TimeSeriesGroup`s. All the `TimeSeries` in these tests had the same sampling interval and their sampling interval did not change. This is because it is not the responsibility of the `SegmentGenerator` to handle different sampling intervals and changes in sampling interval (this is the responsibility of the `SegmentGeneratorController`).

As an example consider the two `TimeSeries` shown in **Table 4.1**, which were grouped together in a `TimeSeriesGroup`.

Time (ms)	100	200	300	400	500	600	700	800	900	1000	1100	...
TS1	1	1	1	1	1	999	999	999	999	999	999	...
TS2	1	1	1	1	1	1	1	1	1	1	1	...

Table 4.1: `TimeSeries` used for example test

For these two `TimeSeries` we would expect the `SegmentGenerator` to return the segments shown in **Figure 4.3**.

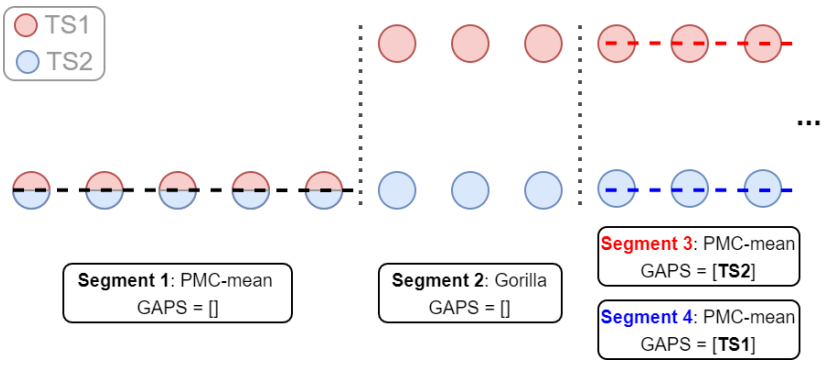


Figure 4.3: Expected segments of `SegmentGenerator` test

The reasons for why we expect the segments shown in **Figure 4.3** are:

- First a **PMC-mean segment** from 100ms to 500ms is returned that represent both `TimeSeries` as the value for both of them is 1 in this time period.
- Then, instead of splitting immediately at 600ms we expect a **Gorilla segment** with a length of 3 because the length bound used in this test is 3. A length bound is required for Gorilla because it is lossless and would otherwise consume all other data points. The reason for why this Gorilla segment is necessary is that ModelarDB-Dynamic still uses the split heuristic discussed in **Section 3.3.3**, which means that compression ratio is used as split heuristic and Gorilla has a worse compression ratio than PMC-mean.
- Then, after emitting the Gorilla segment the two `TimeSeries` should be split, because of the bad compression ratio, and therefore two **PMC-mean segments** should be emitted i.e. one for each `TimeSeries`.

The unit tests worked by feeding the slices generated by the `TimeSeriesGroup` to the `SegmentGenerator` and asserting that it returned the four expected segments.

4.2.2 Integration Test

In order to validate that the data point view works as expected in ModelarDB-Dynamic an integration test was conducted. The purpose of this test is to ensure that all the data points returned by the view are within the allowed error-bound. The steps for the test are:

1. The time series data is ingested by ModelarDB-Dynamic, using an error-bound of 10%

2. The data points for each time series are then extracted by querying the data point view to get the values represented by ModelarDB-Dynamic. This was done one time series at a time by using a shell script that for each time series created a CSV file and saved the query result sorted on timestamps to the file.

3. The extracted data points and original data were then compared using a Python script.

Equation (4.1) shows the comparison done in the Python script from **Step 3**. The comparison uses the same percentage error calculation used in ModelarDB-Dynamic. If this comparison at one point returns **FALSE** then at least one of the extracted data points was not within the maximum allowed error and the test should therefore fail.

$$maxAllowableError > \left(1 - \frac{originalValue - ||originalValue - extracted||}{originalValue}\right) \cdot 100\% \quad (4.1)$$

The data used for this integration test are the 20 time series from house 1 in the REDD dataset. Correlations for the groups discussed in **Section 4.1.1** were used when ingesting the data to ensure that the data point view also still works with groups. To illustrate the results of the comparison between the input data and the extracted data **Figure 4.4** has been created. In the figure, it can be seen that the extracted data points and the original data points are very similar.

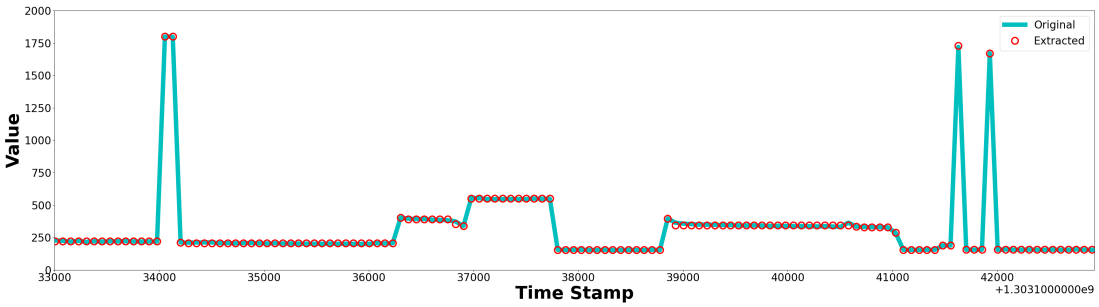


Figure 4.4: Comparison between original data and data extracted from data point view

Figure 4.4 only represents a fraction of the collective data from a single time series, and is only meant as a guiding illustration. The line on **Figure 4.4** illustrate the original data and the data illustrated with 'o' represent every 75'th data point extracted from ModelarDB-Dynamic.

This test successfully proved that the data that is extracted from the data point view is within the allowable error-bound as specified in the configuration for ModelarDB-Dynamic when performing the ingestion.

4.3 Performance tests

The following performance tests have been conducted:

- **Ingestion time:** As explained in **Sections 3.6 to 3.8** major changes have been made to the ingestion e.g. by making it push-based and adding a `SegmentGeneratorController`. The effect of these changes on ingestion speed is therefore tested.
- **Storage usage:** The sampling interval is stored on each segment to support adjustable sampling interval as discussed in **Section 3.4**. The precise effect of our changes in terms of storage space needed is tested.
- **Query speed:** It is expected that the implemented changes have a minor impact on query speed. To confirm this the query speed of the system is also tested.

We will be testing both the original ModelarDB (MDB) and the new ModelarDB-Dynamic (MDB-D) to compare their differences in performance on the same data. All testing was performed with the H2 database as this is the only supported database in ModelarDB-Dynamic and in regards to performance the comparison will be fair as both systems use the same database. Each performance test was run using an error-bound of 10%, and a length bound of 50 for Gorilla compression on a machine running Ubuntu 20.04.3 with the following hardware: CPU: i5-7200U @2.5GHz, RAM: 8GiB, 256 GB SSD.

For the ingestion time and storage usage tests two data sets were made:

- **Ungrouped:** The full REDD data set without any correlation added.
- **Grouped:** Only the grouped data i.e. only the following time series from house 1: `3.Oven`, `4.Oven`, `9.Lighting`, `17.Lighting`, and `18.Lighting` and correlations added for them.

These two splits of the test were made to make any difference between ungrouped or grouped data in the two systems explicit, as a combined test could have hidden interesting differences.

4.3.1 Ingestion time

We measure the ingestion time for both systems on both data sets (grouped and ungrouped). The ingestion time was measured 7 times for each data set for both ModelarDB and ModelarDB-Dynamic, where the fastest and slowest measurements are removed in an attempt to prevent outliers. All ingestion tests were run on the same machine with the same environment as previously mentioned. **Table 4.2** shows the results for the ingestion time, where the last row is the average.

Ungrouped		**Grouped**	
MDB	**MDB-D**	**MDB**	**MDB-D**
55.480s	71.496s	3.505s	7.184s
54.735s	71.365s	3.834s	7.423s
57.034s	71.563s	3.458s	7.482s
55.759s	72.006s	3.433s	7.278s
55.798s	72.351s	3.460s	7.296s
Average			
55.761s	71.756s	3.538s	7.333s

Table 4.2: Ingestion time results

The calculated percentage differences for the average ingestion times are calculated as follows:

- **Ungrouped:** $\frac{V_{new}-V_{old}}{V_{old}} \cdot 100\% = \frac{71.756s-55.761s}{55.761s} \cdot 100\% = 28.685\%$
- **Grouped:** $\frac{V_{new}-V_{old}}{V_{old}} \cdot 100\% = \frac{7.333s-3.538s}{3.538s} \cdot 100\% = 107.252\%$

From the results, it is clear that ModelarDB-Dynamic introduces a significantly larger ingestion time. For the ungrouped data, the ingestion time is increased by 28.7% and for the grouped data it is more than doubled at 107.3%.

Overhead Test

The data set used for grouped data (3, 729, 390 data points) is a lot smaller than the one used for ungrouped data (56, 341, 629 data points). A new data set is therefore created in order to test if the 107.3% increase in ingestion time for grouped data is due to some unknown overhead that would have greater effect on a smaller data set.

To test this we chose to extend the grouped data by copying the data from the original time series X times and appending it on to itself X times. For example, if the original time series was $TS = \langle(1s, 1), (2s, 5), (3s, 10)\rangle$. Then, if $X = 2$ we would get $TS_{extended} = \langle(1s, 1), (2s, 5), (3s, 10), (4s, 1), (5s, 5), (6s, 10)\rangle$.

For our test we chose $X = 15$ because $\frac{56{,}341{,}629}{3{,}729{,}390} \approx 15$. The results of ingesting these extended time series with correlation in both ModelarDB and ModelarDB-Dynamic are show in **Table 4.3**, where the data was ingested seven times and then the fastest and slowest measurements were discarded.

MDB	MDB-D
48.093s	101.898s
47.311s	98.615s
47.86s	97.137s
48.935s	97.564s
47.589s	97.999s
Average	
47.9576s	98.6426s

Table 4.3: Ingestion time results for 15 times extended grouped data

This gives the following percentage difference:

- **Grouped (15x):** $\frac{V_{new}-V_{old}}{V_{old}} \cdot 100\% = \frac{47.9576s-98.6426s}{47.9576s} \cdot 100\% = 105.687\%$

This percentage difference is very similar to the 107, 3% measured earlier. The results therefore suggest that there is no constant overhead that would affect smaller data sets. Other reasons for this larger increase in ingestion time for grouped data are therefore investigated in the following section.

Profiling

In order to identify the reason behind this increase in ingestion time Java Flight Recorder, a profiling tool included in the IntelliJ IDEA [13], was used when ingesting the data. Java Flight Recorder was used to profile the ingestion process 3 times and the averages of these three runs will be used when discussing their effects on the runtime. The following reasons behind the increase in ingestion time were identified:

- **Usage of Collection Classes:** In order to make the code more maintainable and readable it was chosen to introduce the usage of some of Java's collection classes such as Lists and Maps. The problem is, however, that we did not update all the code to use these new collections. For example, the slices maintain a list of the data points that are a part of the slice, however, the `SegmentGenerator` still expects arrays of data points. Therefore this list is converted to an array, which used 1.4% and 1.867% of the total runtime for ungrouped and grouped data respectively. This adds up to quite a bit as similar conversions happen in many places in ModelarDB-Dynamic.
- **Priority Queue:** A priority queue structure was added to ModelarDB-Dynamic to construct slices with only the same sampling interval and timestamp (as described in **Section 3.6.1**). Maintaining this priority queue obviously has some overhead. This overhead was measured to be 4.133% of the total runtime for ungrouped and 5.833% for grouped. The overhead is larger for the grouped data because more elements are kept in the queue at the same time when working with larger groups.
- **`SegmentGeneratorController`:** The primary reason for why ModelarDB-Dynamic performs significantly worse on grouped data compared to ungrouped data is the overhead introduced by the slice delegation from the `SegmentGeneratorController`. For ungrouped data, this overhead is only 2.267% of the total runtime but for grouped data, it is 39.067%. The reason for this is that the `SegmentGeneratorController` uses many look-ups in maps in order to ensure that the generated slices are split correctly and then delegated to the correct `SegmentGenerators`.

We will look at what the ingestion time should be for the grouped data without the overhead from the grouping, this is done to prove that the identified reasons sufficiently explain the 107.252% increase in ingestion time for the grouped data compared to the only 28.685% increase for ungrouped data. First, the expected cost of grouping is calculated by using **Equation (4.2)**. Where $\Delta_{ControllerOverhead}$ is the difference in the overhead added by the `SegmentGeneratorController` for grouped and ungrouped data and $\Delta_{QueueOverhead}$ is the difference in overhead added by the priority queue.

$$\begin{aligned} CostOfGrouping &= AVG_{grouped} \cdot \Delta_{ControllerOverhead} + AVG_{grouped} \cdot \Delta_{QueueOverhead} \\ &= 7.333s \cdot (39.067\% - 2.267\%) + 7.333s \cdot (5.833\% - 4.133\%) \\ &= 2.823s \end{aligned} \tag{4.2}$$

Using this cost of grouping an expected average time for the ingestion time can be calculated as: $AVG_{expected} = AVG_{grouped} - CostOfGrouping = 7.333s - 2.823s = 4.51s$. This gives an percentage increase of $\frac{AVG_{expected} - V_{old}}{V_{old}} \cdot 100\% = \frac{4.51s - 3.538s}{3.538s} \cdot 100\% = 27.473\%$, which is very similar to the 28.685% increase for ungrouped data.

In order to confirm that the calculated expected time of $4.51s$ is correct a test was conducted as explained in **Appendix C**, where the grouped data was ingested without any correlation. The result of this test is that the actual measured average time was $4.361s$, which is close to the expected time.

The focus for this project was to implement adjustable sampling interval in ModelarDB and ingestion speed was therefore not the primary concern. This meant that no optimizations in terms of ingestion speed were performed. However, if ModelarDB-Dynamic should be useful in practice this problem of slowing the ingestion speed to this degree would have to be addressed especially for grouped data. The focus during this optimization of ingestion speed should probably be the `SegmentGeneratorController` as it adds a significant overhead when ingesting grouped data.

Ingestion time on data with adjusted sampling interval

The previous ingestion time tests were clearly in favor of the classical ModelarDB since the data points were arriving with a constant sampling interval, which is what ModelarDB was implemented for. The interesting test case is when the sampling interval changes over time. This feature is not built into ModelarDB and thus it must receive data with the highest sampling interval throughout the whole period.

To test this case some test data was generated using the REDD data set, namely house 1 of the REDD dataset. Specifically, the data with the pre-processing described in **Section 4.1** already applied. The test data consist of two parts.

- The first part consists of x percent of the original data but with a different sampling interval than the original. The percentage values used for x can be seen in the first column of **Table 4.4**, and the sampling interval can be seen on the first row of the same table. The new sampling intervals were achieved by dropping rows from the original data set. As an example to achieve a sampling interval of $2s$ every second row was dropped. So to reiterate, the percentage of low-frequency data defines how much of the original data was transformed to use this new sampling interval.
- The second part of the data was then just the remaining data written to the new data file without any changes.

Table 4.4 shows the ingestion time results when ingested with ModelarDB-Dynamic.

	Sampling interval when low frequency (s)			
x i.e. % of low frequency data	60	10	5	2
95	**2.267s**	**3.447s**	**4.858s**	**9.118s**
90	**2.908s**	**4.228s**	**5.806s**	**11.820s**
80	**4.291s**	**5.554s**	**6.420s**	**12.181s**
70	**5.871s**	**6.678s**	**7.695s**	**13.481s**
50	**8.767s**	**11.475s**	**12.444s**	15.546s
30	14.369s	14.542s	15.59s	17.079s
10	18.737s	18.165s	18.325s	19.332s

Table 4.4: Ingestion time results for modified REDD data set where a percentage of the data points are low frequent. The leftmost column denotes the percentage of the original data set that is transformed into low frequency data. The horizontal topmost numbers denote the sampling interval in the low frequency region of the data. The results marked with **bold** are when ModelarDB-Dynamic performs better than ModelarDB

Ingestion time for classical ModelarDB was tested to be 13.899s where all data was high frequency. Comparing the ingestion time 13.899s with the ingestion times achieved in **Table 4.4** it can be seen when ModelarDB-Dynamic is favorable to use. ModelarDB-Dynamic outperforms classical ModelarDB when more than 50% of the data is low frequency. This was expected as there are far fewer data points to be ingested.

Table D.1 from **Appendix D** shows the percentage of the original data contained in the transformed data. Using this table it can be observed that ModelarDB-Dynamic has faster ingestion, when ingesting around 65% of the original data, which is the case when $x = 50$ and the sampling interval has been adjusted to $5s$ as here the ingestion time is 12.444s, which is less than the ingestion speed for ModelarDB i.e. 13.899s.

The biggest improvement in ingestion speed measured in our test is for the test case, where $x = 95$ and the sampling interval has been adjusted to $60s$. Here, ModelarDB-Dynamic turned out to be $\frac{13.899s}{2.267s} \approx 6.131$ times faster than ModelarDB.

4.3.2 Storage Usage

The storage usage and compression ratio of the systems are measured by taking the size of the database after ingesting the full REDD data set (ungrouped) and after ingesting the grouped data. The test was only run once for each data set as initial testing showed that the size of the database was consistent across runs.

As mentioned these tests are executed using the H2 database. H2 as with many other databases creates a lot of temporary data that hides the actual size of the data written to the database. To prevent this from obscuring the test results the database connection is closed using 'SHUTDOWN COMPACT' which is an H2 specific command that will remove all the temporary data and leave the H2 database file with only the actual data, thus showing the actual compressed data size.

The full REDD data set uses 988.9 MB (988,946,560 bytes) and the grouped uses 64.3 MB (64,294,867 bytes), which leads to the results shown in **Table 4.5**. The header size of the H2

database is measured at 12.3 KB (12,288 bytes) for both the new and old system, this is expected given we only moved SI from the `TimeSeries` table to the `Segment` table (as of **Section 3.4**) and as such maintain the same number of columns. This header value is relatively small compared to the full REDD data set size and is therefore ignored in these results.

Ungrouped		**Grouped**	
MDB	**MDB-D**	**MDB**	**MDB-D**
25.46 MB (25,464,832 bytes)	25.82 MB (25,821,184 bytes)	1.11 MB (1,105,920 bytes)	1.14 MB (1,138,688 bytes)
Compression Ratio			
38.836	38.300	58.137	56.464

Table 4.5: Total Storage results

The change in storage usage from MDB to MDB-D can be calculated to be as follows:

- **Ungrouped:** $Difference = V_{new} - V_{old} = 25.82MB - 25.46MB = 0.36MB$
  - **Percentage Difference:** $\frac{Difference}{V_{old}} \cdot 100\% = \frac{0.36MB}{25.46MB} \cdot 100\% = 1.399\%$
- **Grouped:** $Difference = V_{new} - V_{old} = 1.14MB - 1.11MB = 0.033MB$
  - **Percentage Difference:** $\frac{Difference}{V_{old}} \cdot 100\% = \frac{0.033MB}{1.11MB} \cdot 100\% = 2.963\%$

It could here be assumed that the difference in storage space between the old and the new system would be equal to the size of an integer i.e. 4 *bytes* multiplied onto the number of segments as this is the extra amount of storage space used to store the sampling interval as an integer for each segment.

However, the results show that it is not that simple because as seen in **Table 4.6** ModelarDB-Dynamic generates $363,413$ segments for the ungrouped data, which gives a difference of $363,413 \cdot 4\ bytes = 1.45MB$, which is different from the measured difference of $0.36MB$. The same is true for the grouped data, where we expect a difference off $17,182 \cdot 4\ bytes = 0.069MB$ but the actual difference is $0.033MB$.

Data set	**Amount Segments**			**Amount Data Points**		
	MDB	**MDB-D**	**Difference**	**MDB**	**MDB-D**	**Difference**
Ungrouped	364,841	363,413	-1,428	56,341,629	56,341,629	0
Grouped	16,827	17,182	355	3,729,390	3,729,390	0

Table 4.6: Amount of segments and data points for the two data sets

Two primary reasons for why the actual storage usage differs, which we did not expect, have been identified:

- **Different Amount of Segments:** As seen in **Table 4.6** both ModelarDB and ModelarDB-Dynamic generate the same amount of total data points and therefore both represent all the data. However, ModelarDB and ModelarDB-Dynamic generate different amounts of segments when compressing the data. The reason for this is that even though we tried to keep the `SegmentGenerator` a black box as much as possible some logic was slightly updated when changing it from being pull-based to push-based as explained in **Section 3.7**. Apparently, the changes made in ModelarDB-Dynamic seem to make the segment generation

work better for ungrouped data as it creates 1,428 fewer segments. But the changes to the split and join logic has made it worse for grouped data as it creates 355 more segments for the smaller grouped data set.

- **H2 compression:** H2 uses a compression technique when storing the data, and when using 'SHUTDOWN COMPACT' as we do in our tests then the compression technique runs until completion instead of the standard 200 ms [14]. This compression technique could potentially identify that the sampling interval is the same for all 363413 segments and thereby compress this added sampling interval data for ModelarDB-Dynamic.

In conclusion, as regards storage usage the overall storage cost of implementing adjustable sampling interval was minimal as ModelarDB-Dynamic only use 1.399% more storage for ungrouped data and 2.963% for the grouped data than ModelarDB in our tests. Therefore further investigation into the exact reasons behind these results will be left as future work.

Storage usage on data with adjusted sampling interval

Similar to the last test conducted in **Section 4.3.1**, it is interesting what effect adjusting the sampling interval has on storage usage. To test this the same methodology and data is used, meaning the first x percent of the data uses a different sampling interval than originally. The specific percentage (i.e. x), sampling interval, and storage usage results can be observed in **Table 4.7**. For comparison, the storage space used by the original ModelarDB, with the data having 0% of low frequency, is 5.92MB.

	Sampling interval when low frequency (s)			
x i.e. % of low frequency data	60	10	5	2
95	0.61 MB	1.31 MB	1.93 MB	3.53 MB
90	0.93 MB	1.59 MB	2.17 MB	3.68 MB
80	1.43 MB	2.03 MB	2.55 MB	3.91 MB
70	2.04 MB	2.56 MB	3.00 MB	4.19 MB
50	3.15 MB	3.52 MB	3.84 MB	4.69 MB
30	4.37 MB	4.59 MB	4.78 MB	5.26 MB
10	5.44 MB	5.52 MB	5.58 MB	5.75 MB

Table 4.7: Storage usage results for modified REDD data set where a percentage of the data points are low frequent. The leftmost column denotes the percentage of the original data set that is transformed into low frequency data. The horizontal topmost numbers denote the sampling interval in the low frequency region of the data.

As can be observed in **Table 4.7**, ModelarDB-Dynamic in our tests has a lower storage usage than ModelarDB when an adjustable sampling interval is used in all cases. The percentage of the original data contained in the transformed data shown in **Table D.1** from **Appendix D**. If the data from both tables are combined then it can be observed that ModelarDB-Dynamic performs better even when it is still ingesting 95% of the original data, which is the case when $x = 10$ and the sampling interval has been adjusted to $2s$ as it here uses 5.75 MB, which is less than the 5.92MB used by ModelarDB.

In our test cases ModelarDB-Dynamic used up to $\frac{5.92MB}{0.61MB} \approx 9.705$ times less storage than ModelarDB, which was in the test case, where $x = 95$ and the sampling interval has been adjusted to $60s$.

4.3.3 Query speed

To test the difference in query speed, three queries are run against ModelarDB (MDB) and ModelarDB-Dynamic (MDB-D) where data has been ingested for house 1 in the REDD data set. The three queries can be seen in **Listing 4.1**.

```
Query 1:
    SELECT * FROM datapoint WHERE tid = 1

Query 2:
    SELECT tid, start_time, avg_s(#) FROM segment GROUP BY tid, start_time

Query 3:
    SELECT * FROM datapoint WHERE
        '2011-04-18 16:08:52.0' < timestamp AND timestamp < '2011-04-19 16:08:52.0'
```

Listing 4.1: Queries used to test the query speed

The first query simply asks for all data points from one time series. The second query performs an average aggregate function on all time series by using the user-defined function avg_s. The third query is a range query, selecting all data points measured between two timestamps. Since the three queries are different types of queries, different result times are expected for each type.

The following was the procedure used for obtaining the test results for both systems for the three queries:

1. Ingest house 1 data
2. Execute 7 warm-up queries of query n
3. Measure time for query n 7 times
4. Discard the highest and the lowest reading
5. Increment n

The results are shown in **Table 4.8**.

Query 1 (s)		Query 2 (s)		Query 3 (s)	
MDB	**MDB-D**	**MDB**	**MDB-D**	**MDB**	**MDB-D**
0.608	0.669	0.461	0.489	1.866	1.792
0.640	0.678	0.460	0.457	1.813	1.835
0.618	0.663	0.447	0.411	1.873	1.803
0.594	0.764	0.469	0.406	1.805	1.837
0.653	0.652	0.474	0.461	1.856	1.817
Average (s)					
0.623	0.685	0.462	0.445	1.843	1.817

Table 4.8: Query speed test results. All measurements are in seconds.

From the query time results, there are generally not any substantial differences in query speeds. This was expected since no major changes were made to the database apart from adding an extra sampling interval column to the segment table.

5 | Discussion

This chapter discusses the big picture as a continuation of the discussion of the individual test results in **Chapter 4**. The discussion presents the opportunities that have become possible given the added functionality in ModelarDB-Dynamic. The discussion also highlights some of the costs that have been found to be associated with the implemented feature.

5.1 Benefits of adjustable sampling interval

ModelarDB-Dynamic has the additional functionality of allowing users to adjust the sampling interval of a specific time series during ingestion. This allows users to reduce the load on ingestion, by choosing a higher (less frequent data) sampling interval during non-critical times. Oppositely the sampling interval can be lower (more frequent data) during critical times to collect more data, which is preferred as it allows more detailed analysis. This is in contrast to ModelarDB where the sampling interval must remain constant. So to state it clearly this added feature allows the system to be configured to use cases where data for some periods are more critical than for others. This can lead to a higher value to cost ratio for gathering and storing time series data.

Initially, it may seem as if less data would lead to less storage required. However, as ModelarDB and ModelarDB-Dynamic use model compression this is not necessarily the case. As the storage required depends on the number of segments and their model type. The major difference is that fewer data points can lead to different segments, that potentially span longer time periods, as models will be fitted differently.

5.2 Drawbacks of an adjustable sampling interval

ModelarDB-Dynamic's additional functionality does, however, come at a cost as highlighted in **Section 4.3**. Utilizing the same configuration with a constant sampling interval for both systems, ModelarDB-Dynamic's ingestion speed is approximately 28.7% and 107.3% slower for ungrouped and grouped data respectively. Furthermore, ModelarDB-Dynamic used 1.4% and 3% more storage space for ungrouped and grouped data respectively, which, is not much and even expected since we had to update the database to store SI for each segment. ModelarDB-Dynamic had a similar query speed when compared to ModelarDB. This is expected given that we only made small changes to the views such as adding sampling interval to the segment view. The reconstruction of data points should also not take much longer in the ModelarDB-Dynamic as we only moved SI from time series to segments. The effect of this moving of SI for the data point view is that the segments now have to be queried to get the SI used to reconstruct the data points. This, however, makes little difference as segments are already queried to reconstruct data points.

An important mention with regards to the ingestion speed is that the main focus of ModelarDB-Dynamic has been to demonstrate the capability of adjustable sampling interval as an extension

of the original ModelarDB. There has not been a major focus on optimizing ingestion speed, however, a profiler revealed a few areas where improvements could be made e.g. `SegmentGeneratorController`, which if addressed could lead to improvements in ingestion speed.

6 | Conclusion

This project has worked with ingestion, storage, and querying of telematics data stored as time series. Telematics data is known to be high in frequency and data volume, therefore, storing the raw data can therefore be infeasible. Traditional solutions to this problem are to produce and store aggregates of the values instead. This, however, has the potential to hide important fluctuations or outliers that will be obscured by the smoothing effect of averages. To combat this loss of detail, model-based compression can be used as an effective way of compressing time series data. Model-based compression makes it possible to achieve a compression ratio that makes it possible to store all the data without an insurmountable cost, especially if some amount of error is tolerable or if data can be grouped.

ModelarDB is a Time Series Management System (TSMS) that makes use of Multi-Model Group Compression (MMGC) and as such is a step towards being able to store more detailed data. However, ModelarDB currently lacks some very desirable features. One feature that is lacking is to allow for different time series to have different sampling intervals as well as having a time series' sampling interval vary over time. This would enable the possibility to store data with higher frequency during critical periods and to be able to store less frequent data during periods of normal operation. Working with high-frequency data is costly, but more accurate, and is therefore not always desirable. Given these considerations, we arrived at the following problem formulation: "*How should an adjustable sampling interval be implemented in ModelarDB while still offering a reasonable compression rate, ingestion rate, and query time?*"

During this project, many different design considerations were made in order to implement an adjustable sampling interval. The adjustable sampling interval was made possible by allowing configuration messages to be sent over the data stream. Additionally, the database is updated such that each generated segment is stored with its respective sampling interval, which leads to an expected lower compression ratio. The measured required storage was 1.4% more for ungrouped data and 2.96% more for grouped data. The updated database could potentially also have affected query speed when reconstructing data points from segments. However, the measured query time differed by a negligible amount when comparing ModelarDB-Dynamic to ModelarDB.

For our changes to be usable by real-world projects, the support for an adjustable sampling interval needs to be extended to the other data sources and storage engines, as this project has only focused on a single source and storage engine, namely CSV as a source and H2 storage. Furthermore, some work needs to be done to improve the ingestion rate as having a 28.7% (ungrouped data) and 107.3% (grouped data) increase in ingestion time which is not ideal in practice. However, we assess that the ingestion speed could be improved with additional effort as discovered by the profiler. It is, however, not expected to be able to hit the levels of ingestion speed offered by ModelarDB as ModelarDB-Dynamic has some ingestion overhead to allow for the adjustable sampling interval.

The tests for ingestion time conducted showed that ModelarDB-Dynamic is 28.7% and 107.3% slower for ungrouped and grouped data respectively when ingesting with a fixed sampling interval. However, when an adjustable sampling interval is applied for ModelarDB-Dynamic it performs up to 6x faster in the very best case. ModelarDB-Dynamic has the same ingestion speed as

ModelarDB when around ingesting around 65% of the original data.

In the end, we can conclude that we have managed to implement the feature of adjustable sampling interval into ModelarDB without meaningfully increasing the compression ratio or query time. The ingestion time on the other hand has been increased by quite a bit but it is expected that this increase can be brought down to a manageable level. Lastly for this to be usable in real big data settings the support for adjustable sampling interval will have to be implemented into the remaining sources and storage engines namely Spark and Cassandra.

Glossary

MDB ModelarDB. 35, 36, 40, 43

MDB-D ModelarDB-Dynamic. 35, 36, 40, 43, 54

MGC Model-based Group Compression. 10

MMC Multi-Model Compression. 10

MMGC Multi-Model Group Compression. 9, 10, 20, 46

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, 2, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21, 23, 25, 30, 35, 36, 38, 39, 40, 41, 42, 44, 45, 46, 47, 51

ModelarDB-Dynamic ModelarDB but with support for an adjustable sampling interval. i, 30, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 44, 46, 54

SI Sampling Interval. 9, 14, 17, 18, 19, 21, 23, 24, 25, 28, 29, 40, 44

TSMS Time Series Management System. i, 2, 9, 10, 46

Bibliography

[1] Shafiq Dharani. *Telematics: Poised for strong global growth.* 2018.
URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/telematics-poised-for-strong-global-growth (visited on 13/09/2021).

[2] 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.

[3] Nguyen Quoc Viet Hung, Hoyoung Jeung and Karl Aberer. 'An Evaluation of Model-Based Approaches to Sensor Data Compression'. eng. In: *IEEE transactions on knowledge and data engineering* 25.11 (2013), pp. 2434–2447. ISSN: 1041-4347.
URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6378372.

[4] 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.

[5] 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.

[6] 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.

[7] Michael Armbrust et al. 'Spark SQL: Relational Data Processing in Spark'. In: SIGMOD '15.
Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1383–1394. ISBN: 9781450327589. DOI: 10.1145/2723372.2742797.
URL: https://doi.org/10.1145/2723372.2742797.

[8] Avinash Lakshman and Prashant Malik. 'Cassandra — A Decentralized Structured Storage System'. In: *Operating Systems Review* 44 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922.

[9] Søren Kejser Jensen. *ModelarData / MiniModelarDB.*
URL: https://github.com/ModelarData/MiniModelarDB (visited on 20/10/2021).

[10] Søren Kejser Jensen and Jason Zhang. *skejserjensen/ModelarDB - repository.*
URL: https://github.com/skejserjensen/ModelarDB (visited on 01/10/2021).

[11] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition.* 3rd. The MIT Press, 2009. ISBN: 0262033844.

[12] J. Zico Kolter and Matthew J. Johnson. *REDD: The Reference Energy Disaggregation Data Set.*
URL: http://redd.csail.mit.edu/ (visited on 15/11/2021).

[13] JetBrains. *Java Flight Recorder.* URL: https://www.jetbrains.com/help/idea/java-flight-recorder.html#jfr-configurations (visited on 13/12/2021).

[14] H2. *Compacting a Database*. URL: http://www.h2database.com/html/features.html#compacting (visited on 09/12/2021).

A | Dependency diagram

The dependency diagram was made to kick start implementation of ModelarDB by giving an overview of the different components in the system. The purpose of the diagram is mainly for us to better understand the system.

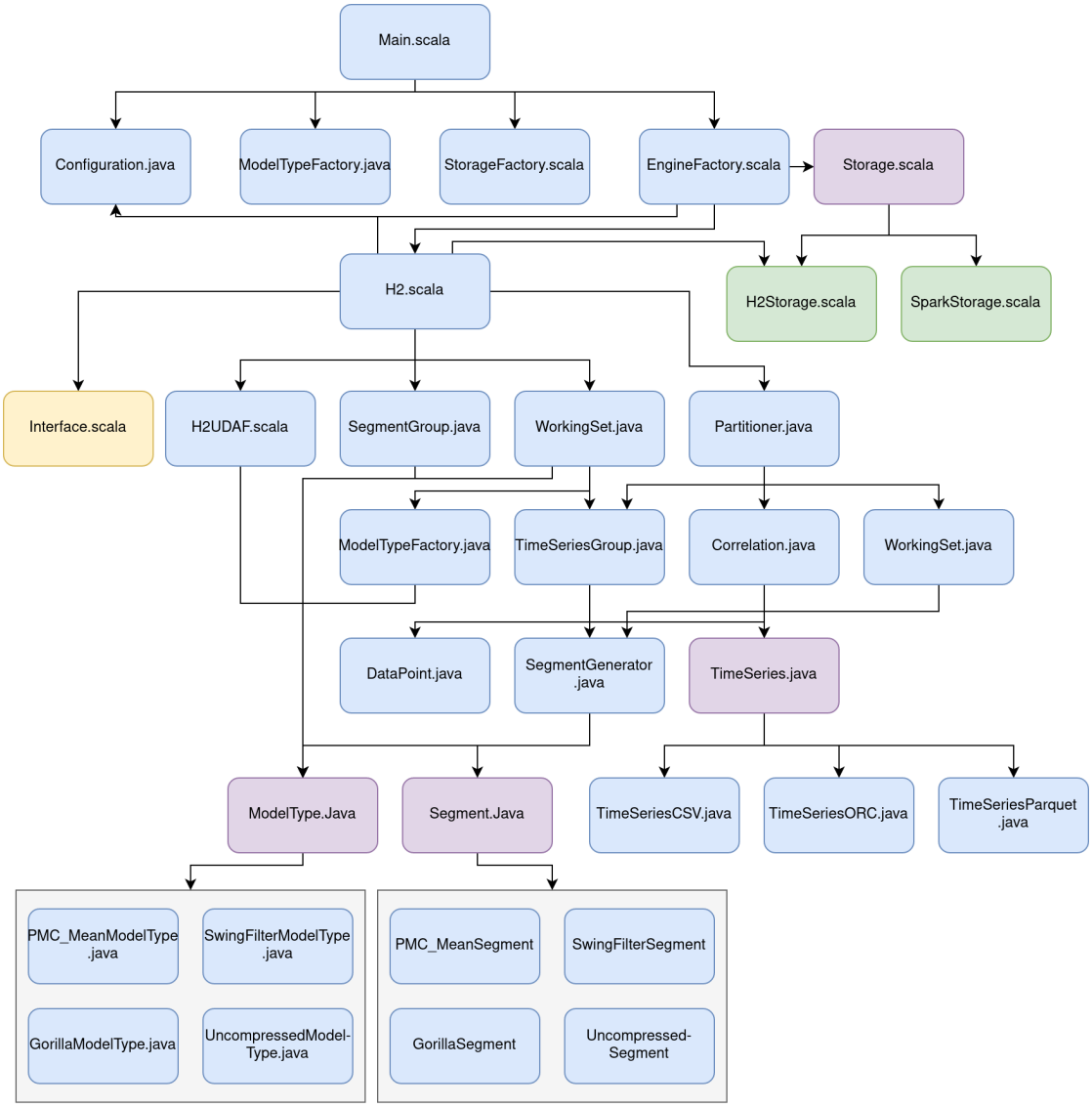


Figure A.1: Overview of the code flow when using H2 storage. The arrows symbolize a dependency from an instance to another instance.

B | Unit test time series data

The time series data manually created for unit testing the `SegmentGenerator` are presented here together with a small description of what it is used to test. Not all data points fit neatly in the tables and are therefore, when needed, represented with trailing dots.

Table B.1 is regular and has timestamps up to 1,500, all values are 1. This data set is used to test if a constant segment (PMC-Mean) is created with the data points.

Timestamp	100	200	300	400	500	600	700	800	900	1000	1100	...
Values	1	1	1	1	1	1	1	1	1	1	1	...

Table B.1: `TimeSeries` data set 1

Table B.2 is also regular and has timestamps up to 1,500 all values incremented by 1. The data set is used to test that a linear segment (Swing) can be created.

Timestamp	100	200	300	400	500	600	700	800	900	1000	1100	...
Values	1	2	3	4	5	6	7	8	9	10	11	...

Table B.2: `TimeSeries` data set 2

Table B.3 is a time series with gaps. Used to test that two segments are created correctly when the time series has gaps.

Timestamp	100	200	300	400	500	1100	1200	1300	1400	1500
Values	1	1	1	1	1	1	1	1	1	1

Table B.3: `TimeSeries` data set 3

Table B.4 used to test that ingestion is working when ingesting two time series (in this case together with **Table B.1**) and one time series ends early.

Timestamp	100	200	300	400	500	600	700	800	900	1000
Values	1	1	1	1	1	1	1	1	1	1

Table B.4: `TimeSeries` data set 4

Table B.5 continues with timestamps up to 1,500 with regular sampling interval and no gaps with value 999 for all data points not shown. Used to test if splitting works as expected by grouping it together with **Table B.1**. This is the example discussed in **Section 4.2.1**.

Timestamp	100	200	300	400	500	600	700	800	900	1000	1100	...
Values	1	1	1	1	1	999	999	999	999	999	999	...

Table B.5: `TimeSeries` data set 5

Table B.6 changes to value 999 at timestamp 600 then at 1,200 changes back to value of 1. This time series is used to check if joining works after splitting by grouping it together with **Table B.1**.

Timestamp	100	...	500	600	...	1100	1200	1300	1400	1500
Values	1	...	1	999	...	999	1	.1	1	1

Table B.6: `TimeSeries` data set 6

C | Uncorrelated grouped data

The data for the time series `3.Oven`, `4.Oven`, `9.Lighting`, `17.Lighting`, and `18.Lighting` from house 1 **without any correlation** was ingested 7 times by ModelarDB-Dynamic, and the ingestion speed was measured. The fastest and slowest measurements are removed in an attempt to prevent outliers. The results of this test are shown **Table C.1**.

MDB-D
4.191s
4.441s
4.463s
4.345s
4.365s
Average
4.361s

Table C.1: Ingestion time results for the grouped data but ingested as uncorrelated

D | Storage usage with adjustable SI

Table D.1 shows how many percent of the data points are still in the data after applying the transformation.

	Sampling interval when low frequency (s)			
x i.e. % of low frequency data	60	10	5	2
95	6.58 %	14.50 %	24.00 %	52.50 %
90	11.50 %	19.00 %	28.00 %	55.00 %
80	21.33 %	28.00 %	36.00 %	60.00 %
70	31.17 %	37.00 %	44.00 %	65.00 %
50	50.83 %	55.00 %	60.00 %	75.00 %
30	70.50 %	73.00 %	76.00 %	85.00 %
10	90.17 %	91.00 %	92.00 %	95.00 %

Table D.1: The percentage of the original data that is in the files for the given cell.