Model-Based Time Series Management at Scale

Jensen, Søren Kejser

Publication date:
2019

Document Version
Publisher's PDF, also known as Version of record

MODEL-BASED TIME SERIES MANAGEMENT AT SCALE

BY
SØREN KEJSER JENSEN

DISSERTATION SUBMITTED 2019

AALBORG UNIVERSITY
DENMARK

Model-Based Time Series Management at Scale

PhD Dissertation

Søren Kejser Jensen

Dissertation submitted August, 2019

Dissertation submitted: August, 2019

PhD supervisor: Professor Torben Bach Pedersen
Aalborg University

PhD Co-Supervisor: Associate Professor Christian Thomsen
Aalborg University

PhD committee: Associate Professor Simonas Saltenis (chairman)
Aalborg University

Professor, Klemens Böhm
Karlsruhe Institute of Technology

Associate Professor Yongluan Zhou
Copenhagen University

PhD Series: Technical Faculty of IT and Design, Aalborg University

Department: Department of Computer Science

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-473-7

Published by:
Aalborg University Press
Langagervej 2
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Rosendahls, 2019

Abstract

Sensor networks are increasingly deployed to monitor and simplify management of physical entities such as planes and wind turbines. As the produced amount of sensor data increases, so do the requirements for methods and systems that can store and analyze the vast quantities of sensor data being collected. However, the systems used in industry are in general not designed to manage sensor data at large scale. This forces practitioners to only store simple aggregates, for example averages over a 10-minute window, instead of the raw time series. As a remedy, this thesis proposes a model-based Time Series Management System (TSMS) named *ModelarDB* that supports ingestion, storage, and multi-dimensional analysis of time series at scale. As part of ModelarDB, this thesis proposes methods for model-based management of time series:

Firstly, the thesis provides a comprehensive literature survey of TSMSs proposed by academia and through industrial research. A set of classification criteria are proposed that illustrate the commonalities and differences of each system. The surveyed systems are grouped based on these criteria and the user-facing functionality they provided to give a simple to consume overview of the TSMSs. The survey shows that systems combining data storage and query processing in one application tend to be research prototypes designed to evaluate new methods, algorithms, and data structures. TSMSs that use separate applications for storage and query processing tend to reuse existing systems for at least one of these, and are deployed to solve real-world problems at scale. TSMSs developed as extensions to Relational Database Management Systems (RDBMSs) are designed to evaluate new methods for model-based query processing of time series. Based on the analyzed TSMSs and future research directions proposed by experts in the field, the survey proposes that research into a model-based approach for time series management should be performed. The survey should significantly reduce the work required for researchers entering this topic of research and allow practitioners to easily compare the benefits and drawbacks of each TSMS.

Secondly, the thesis addresses the challenge of using models for storage and analysis of individual time series. As a result, the thesis proposes several methods for model-based management of individual times series. A model in this context is any representation from which a time series can be reconstructed within a user-defined error bound (potentially zero). A model-agnostic ingestion method is proposed that split each time series into dynamically sized sub-sequences and represents each using a model

from a set of model types. These are either included with the system or user-defined using a proposed API. Methods for model-based query processing are proposed to efficiently execute aggregate queries directly on models instead of on reconstructed data points. In addition, methods for predicate push-down and for improving the performance of projections using static code-generation are proposed. A general schema is proposed for storing time series as models, with specific optimizations proposed for using the schema with a distributed key-value store. ModelarDB is implemented based on a new general architecture for model-based TSMSs and the proposed methods are implemented as part of the system. An evaluation demonstrates that the system hits a sweet spot and provides fast ingestion, state-of-the-art compression, and fast scalable query processing for large scale aggregate queries, while also being competitive with existing storage solutions for small scale aggregate and time point-range queries. The proposed methods for model-based management of time series allow storage and analysis of time series at larger scale than other solutions by significantly decreasing the amount of storage required.

Thirdly, the thesis addresses the challenge of using models for storage and analysis of correlated time series. As a result, the thesis proposes extensions to ModelarDB's ingestion method, the included model types, and its model-based query processing methods. In addition, a method for grouping correlated time series is proposed. This method creates groups of correlated time series based on user-hints, given using a set of primitives that efficiently allows domain experts to describe how time series are correlated using their domain knowledge or by analyzing historical data. As correlation is determined based on metadata and not ingested data points, the large runtime overhead required to compute which time series are correlated is removed. This increases the scalability of the system. Regarding query processing, the thesis proposes methods for performing multi-dimensional analytics directly on models using metadata stored as a data cube. An evaluation demonstrates that the proposed methods increase ModelarDB's ingestion rate, compression ratio, and query processing speed for large scale queries, while keeping the system scalable and competitive with existing storage solutions for small scale aggregate and time point-range queries on data sets with many short time series. As such, the proposed methods for management of correlated time series further increases the scale at which time series can be managed.

Finally, the thesis provides a demonstration of the current version of ModelarDB. The system's architecture and how it can be deployed using existing infrastructure is presented. An updated API is described for implementing user-defined model types without modifying ModelarDB. A method combining static and dynamic code-generation to more efficiently perform projections is proposed. How to use the system from a user-perspective is shown as examples regarding the system's configuration format, the proposed primitives for describing correlation, and the system's extended SQL query language. A graphical user interface is presented which illustrates both the functionality provided by ModelarDB and the underlying implementation. A demonstration scenario is proposed that allows users to experience how ModelarDB supports model-based time series management at scale.

Resumé

Sensornetværk bliver i stigende grad anvendt til at overvåge og simplificerer administrationen af fysiske enheder, som for eksempel fly og vindmøller. I takt med at mere sensordata produceres, stiger behovet for metoder og systemer, der kan lagre og analysere de enorme mængder sensordata som bliver indsamlet. Dog er systemerne, som bliver brugt industrielt, generelt ikke designet til at håndtere sensordata i stor skala. Dette tvinger folk i industrien til blot at gemme simple aggregater, for eksempel et gennemsnit for hver 10. minut, fremfor de rå tidsserier. Denne afhandling foreslår en løsning på problemet i form af det modelbaserede Time Series Management System (TSMS) *ModelarDB*, som kan indlæse, lagre og foretage multidimensionel analyse af tidsserier i stor skala. Som en del af ModelarDB foreslår afhandlingen metoder til modelbaseret håndtering af tidsserier.

For det første indeholder afhandlingen en omfattende litteraturundersøgelse af TSMSs, der blive foreslået af akademikere og forskere i industrien. Et sæt klassificeringskriterier er foreslået, der viser fællestræk og forskelle ved de analyserede TSMSs. De undersøgte systemer er grupperet ud fra de foreslåede kriterier, og den funktionalitet som hvert system tilbyder, for at give en overskuelige oversigt af dem. Litteraturundersøgelsen viser, at systemer, der kombinerer komponenterne til at lagre og forespørge på tidsserier i ét program, generelt er udviklet til at evaluerer nye metoder, algoritmer og datastrukturer. TSMSs, hvor to forskellige programmer bruges til at lagre og forespørge på tidsserier, genbruger ofte eksisterende systemer til mindst en af disse komponenter. Disse systemer kan skalere, og bliver anvendt til løse problemer i industrien. Systemer, der er udviklet som udvidelser til Relational Database Management Systems (RDBMSs), anvendes til at evaluerer nye metoder til at forespørge på tidsserier ved brug af modeller. Baseret på analysen af TSMSs og forslag til ny forskning fra eksperter inden for feltet, konkluderer litteraturundersøgelsen, at forskning i modelbaserede TSMSs bør foretages. Litteraturundersøgelsen bør reducere arbejdsbyrden for forskere, som begynder at arbejde med TSMSs, og gør det nemmere for folk i industrien at sammenligne fordele og ulemper ved hver TSMS.

For det andet adresserer afhandlingen problemet med at bruge modeller til at lagre og analyserer individuelle tidsserier, og præsenterer metoder til modelbaseret håndtering af individuelle tidsserier. En model i denne kontekst er enhver repræsentation, hvorfra en tidsserie kan blive genskabt indenfor en brugerdefineret fejlgrænse, som kan

være nul. En modelagnostisk indlæsningsmetode bliver foreslået, som opdeler hver tidsserie i delsekvenser af dynamisk længde og repræsenterer hver med en model fra et set af modeltyper. Disse modeltyper er enten inkluderet i systemet eller implementeret af brugere gennem et foreslået API. Metoder til at udfører forespørgsler, der beregner aggregater direkte på modeller fremfor datapunkter rekonstrueret fra modeller, bliver også foreslået. Derudover bliver der foreslået metoder til at videregive prædikater til det underliggende lager, samt metoder til at forøge hastigheden af projekteringer ved brug af dynamisk kodegenerering. Et generelt skema til at lagre tidsserier som modeller bliver foreslået, samt specifikke optimeringer, der kan anvendes hvis skemaet bliver implementeret i en distribueret key-value database. ModelarDB er implementeret baseret på en ny general arkitektur for modelbaserede TSMSs, og de foreslåede metoder er implementeret som en del af systemet. En evaluering viser, at systemet er hurtigt til at indlæse tidsserier, har meget effektiv komprimering, og kan effektivt udføre forespørgsler der beregner aggregater over store dataset. Samtidigt kan systemet udfører forespørgsler der beregner aggregater over få tidsserier eller udtrækker enkelte datapunkter med omtrent samme hastighed som for eksisterende dataformater. De foreslåede metoder for modelbaseret håndtering af tidsserier gør det muligt at lagre og analyserer tidsserier i større skala end de eksisterende løsninger, ved at reducere det påkrævede lager markant.

For det tredje håndterer afhandlingen problemet med at bruge modeller til at lagre og analyserer korrelerede tidsserier, og foreslår udvidelser til ModelarDB's indlæsningsmetode, de inkluderede modeltyper, og dets metoder til at lave forespørgsler på modeller. Derudover præsenteres en metode til at gruppere korrelerede tidsserier. Metoden grupperer korrelerede tidsserier ud fra user-hints givet ved brug af et nyt set primitiver. Disse primitiver gør det nemt for brugere at beskrive hvilke tidsserier der er korrelerede ud fra deres domæneviden eller analyse af eksisterende data. Eftersom hvilke tidsserier der er korrelerede, er beskrevet ud fra metadata fremfor datapunkter, har metoden intet ressourceforbrug under indlæsning af tidsserierne. Dette gør systemet mere skalerbart. Med hensyn til forespørgsler, foreslår afhandlingen metoder til at udfører multidimensionel analyse direkte på modeller ved brug af metadata lagret som en data cube. En evaluering viser, at de foreslåede metoder forbedrer ModelarDB's indlæsningshastighed, komprimering, og hastigheden af forespørgsler i stor skala. Samtidig viser den, at ModelarDB stadig skalerer, og kan udføre forespørgsler der beregner aggregater over få tidssier eller udtrækker enkelte datapunkter med omtrent samme hastighed som eksisterende dataformater for datasæt med mange korte tidsserier. De foreslåede metoder for modelbaseret håndtering af korrelerede tidsserier forøger yderligere den størrelsesorden, som det er muligt at lagre og analyserer tidsserier for.

Endeligt indeholder afhandlingen en demonstration af den nyeste udgave af ModelarDB. Systemets arkitektur, og hvordan det kan genbruge eksisterende infrastruktur, bliver beskrevet. Et opdateret API, der gør det muligt at implementerer nye modeltyper uden at ændre ModelarDB, er beskrevet. En metode til hurtigere at udføre projekteringer ved at kombinerer statisk og dynamisk kodegenerering bliver præsenteret. Hvordan systemet kan anvendes af brugere, bliver beskrevet med eksempler på

ModelarDB's konfigurationsformat, de foreslåede primitiver til at beskrive korrelation, og systemets SQL-sprog til forespørgsler. En grafisk brugergrænseflade bliver beskrevet, der viser funktionaliteten, som ModelarDB tilbyder, samt hvordan denne er implementeret. Et scenarie er beskrevet, der overfor brugere demonstrer hvordan ModelarDB understøtter modelbaseret håndtering af tidsserier i stor skala.

Acknowledgement

I would like to thank the following people for their support during my PhD studies:

My supervisors Torben Bach Pedersen and Christian Thomsen deserve thanks for teaching me how to do research, for efficiently providing helpful feedback on drafts and new ideas, and for their patience despite my many questions and mistakes. In addition, I would like to thank them for providing me with encouragement during the difficult parts of my PhD studies.

All of our partners from the industry also deserve acknowledgment for providing us with real-life data sets and for answering our questions regarding the energy domain and wind turbines in particular. In addition, I want to thank the Innovation Fund Denmark, the European Union, and Microsoft for providing various degrees of funding.

I would also like to thank everybody at the Database, Programming and Web Technologies (DPW) Group at Cassiopeia for creating a pleasant work environment with room for both cozy social interactions and challenging research. In addition, I would like to provide a special thanks to Ilkcan Keles for helping me get accustomed to the life of a PhD student when I started, for great professional and social discussions, and for providing feedback on the summary of this thesis. I would also like to thank Robert Waury for our great discussions on many and often obscure topics, both work-related and otherwise, and for being the facilitator of so many social activities throughout the years. A special thanks must also be provided to Arne Joachim Skou, Christian Thomsen, Bent Thomsen, Lone Leth Thomsen, Kurt Nørmark, and René Rydhof Hansen for helping me improve my teaching skills. I would also like to thank the administrative staff at Aalborg University for all of their assistance and for being patient despite my many questions and mistakes.

During my PhD studies I spend six months vising the Data Intensive and Knowledge Oriented Systems (diNo) Group at Paris Descartes University. I would like to thank Themis Palpanas for hosting me, as well as everybody at diNo for being welcoming and for all of their support during my stay. While in Paris, I lived in the Danish Student House managed by Fondation Danoise and I would like to thank everybody at the house for creating a pleasant environment with so many different people living under one roof. The administrative staff at both Fondation Danoise and Paris Descartes University also deserve a special thanks for answering my many questions and for providing me with assistance both before and during my stay.

Last, I would like to thank my family for all of their support during my PhD studies, for their patience and understanding despite my both long and often strange working hours, for their visits during my stay in Paris, and for generally being accommodating during my PhD studies at Aalborg University.

Søren Kejser Jensen
Aalborg University, August 14, 2019

Contents

Thesis Details

Thesis Title:	Model-Based Time Series Management at Scale
PhD Student:	Søren Kejser Jensen
Supervisor:	Professor Torben Bach Pedersen, Aalborg University
Co-Supervisor:	Associate Professor Christian Thomsen, Aalborg University

The main body of this thesis consists of the following papers.

[A] Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, "Time Series Management Systems: A Survey". *In IEEE Transactions on Knowledge and Data Engineering (TKDE), Volume 29, Number 11, Pages 2581–2600, November, 2017*.

[B] Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, "ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra". *In Proceedings of the VLDB Endowment (PVLDB), Volume 11, Number 11, Pages 1688–1701, July, 2018*.

[C] Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, "Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB". *Unpublished Manuscript*.

[D] Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, "Demonstration of ModelarDB: Model-Based Management of Dimensional Time Series". *In Proceedings of the International Conference on Management of Data (SIGMOD), Pages 1933–1936, 2019*.

This thesis has been submitted for assessment in partial fulfillment of the PhD degree. The thesis is based on the scientific papers which are listed above. Parts of the papers are used directly or indirectly in the extended summary of the thesis. As part of the assessment, co-author statements have been made available to the assessment committee and are also available at the Faculty. The permission for using the published and accepted articles in the thesis has been obtained from the corresponding publishers with the conditions that they are cited and DOI pointers and/or copyright/credits are placed prominently in the references. In reference to IEEE copyrighted material which

is used with permission in this thesis, the IEEE does not endorse any of Aalborg University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink.

Part I

Thesis Summary

Thesis Summary

1 Introduction

1.1 Motivation and General Approach

Large amounts of sensor data are being collected in almost every domain. For example, a Boeing 787 can produce half a terabyte of sensor data per flight [71], while Facebook twice in just 18 months had to double the size of their cluster which caches the data points collected by their monitoring infrastructure [66]. Despite the different physical entities being monitored, all sensor data can be represented as *time series*. A time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ is a sequence of data points in increasing order by time where each data point is a pair of a timestamp t_i and a value v_i where $1 \leq i$. Many Time Series Management Systems (TSMSs) have been proposed in recent years due to the increased amount of time series data being produced. These systems are also known as Time Series Databases or Data Series Management Systems if the ordering of the sequences managed are generalized beyond time [60]. However, most of these TSMSs focus on specific niches [7, 16, 42] and few support high-level analytics such as similarity search [60]. The high-level abstractions provided by Relational Database Management Systems (RDBMSs) greatly simplifies management and analysis of structured relational data. However, no such widespread general purpose high-level abstraction exists for time series [60]. Therefore, to fully benefit from the amount of time series data being collected, a general purpose TSMS must be designed with a storage model, data structures, algorithms, and optimization methods specifically designed for management and analysis of time series at scale [60, 61, 75].

Many existing storage solutions have been used for time series. However, existing formats and systems provides much less compression compared to TSMSs as shown in Table 1. In this table, we compare the amount of storage required to store time series from wind turbines collected with a sampling rate of 100 milliseconds. The compared storage solutions are text files (CSV), an open-source RDBMS (PostgreSQL), a commonly used proprietary RDBMS with a license that prevents us from using its name (RDBMS-X), a distributed key-value store (Apache Cassandra), binary files with a columnar layout (Apache Parquet and Apache ORC), an open-source TSMS (InfluxDB), and our model-based TSMS (ModelarDB). The comparison was performed

as part of Paper B [43], and the selection of storage solutions were based on discussions with companies in the energy sector, the survey conducted in Paper A [42], and DB-Engines Ranking [16]. The results show that RDBMSs are not the best choice for storing time series as PostgreSQL increases the storage requirement by 1.34 times compared to the CSV files, while the proprietary RDBMS-X only reduces the storage required by 3.49 times after switching to a clustered index with a columnar layout. Apache Cassandra performs slightly better as it reduces the amount of storage required by 5.21 times, but provides a more restricted query model compared to the RDBMSs by lacking, for example, an `OR` operator. While Apache Parquet performs similar to Apache Cassandra, Apache ORC reduces the amount of storage required 43.16 times compared to the CSV files. However, Apache Parquet and Apache ORC are not well suited for online analytics as an entire file must be written before the data it contains can be queried. While the TSMS InfluxDB reduces the amount of storage required by 134.57 times compared to the CSV files, it provides a limited query model. For example, it has no `DATEPART` functionality and cannot perform aggregation in the time domain for intervals of varying sizes, such as months [37, 38, 40]. Our system *ModelarDB* reduces the storage required by at least 174.46 times compared to CSV. When compared to existing storage solutions, it requires 1.3–391.44 times less storage.

Ad-hoc solutions have been created as workarounds for the limitation of the existing storage solutions when storing time series. In the energy domain, high quality sensors with wired power and connectivity are used to monitor modern wind turbines, solar panels and power plants. The high quality time series produced by these sensors have a regular sampling interval, and the few incorrect and out-of-order data points that do occur are corrected by existing cleaning procedures. However, while the owners and

Table 1: Comparison of common storage solutions and ModelarDB (Updated) [43]

Storage Method	**Size (GiB)**
CSV Files	582.68
PostgreSQL 10.1	782.87
RDBMS-X – Row	367.89
RDBMS-X – Column	166.83
Apache Cassandra 3.9	111.89
Apache Parquet Files	106.94
Apache ORC Files	13.50
InfluxDB 1.4.2 – Tags	4.33
InfluxDB 1.4.2 – Measurements	4.33
ModelarDB	2–3.34

manufacturers of wind turbines would like to sample their sensors at high frequency to improve the monitoring and allow for more detailed analysis, it is currently infeasible to do so due to the amount of storage required. Currently, they only store simple aggregates over a fixed window size, for example 1–10 minute averages. An example is shown in Figure 1, where a sequence of twelve data points are reduced to just a single average. It is clear from this example that reducing each sub-sequence to just a single data point removes all outliers and fluctuations in the sequence which could have indicated problems with the monitored entity.

A dynamic sampling interval can also be used in order to reduce the storage requirement. By using a low sampling interval by default and then increasing it when a critical event occurs, the storage requirement for the time series would be decreased. However, it is not known what constitutes a critical event based on discussions with owners and manufacturers of wind turbines. A constant high sampling rate is required for this reason. Another method for reducing both the amount of storage required and the query processing time is Approximate Query Processing (AQP). In this context we use the term AQP for both approximate storage and approximate query processing. AQP allows a TSMS to store time series and provide query results within a guaranteed error bound. This error bound is often set by the user as required for their domain. Some systems, such as BlinkDB [1] and SnappyData [58], use sampling for AQP to reduce query processing time. However, sampling does not reduce the storage requirement and can increase it if the samples are pre-computed and stored [14]. To reduce both the amount of storage required and the query processing time, a model-based approach can be used where time series are stored as models. We use the term model for any representation from which a time series can be reconstructed within a user-defined error bound (possibly zero). As an example, a linear function given by the equation $v = a \times t + b$ can efficiently represent an increasing, decreasing or constant sub-sequence of a time series using only two floating-point values a and b. However, as the structure of time series often change over time the most appropriate model type to use can be different for each sub-sequence, so multiple model types should be used even for a single time series. In addition, time series are often correlated. For example, two co-located temperature sensors should produce similar values. These

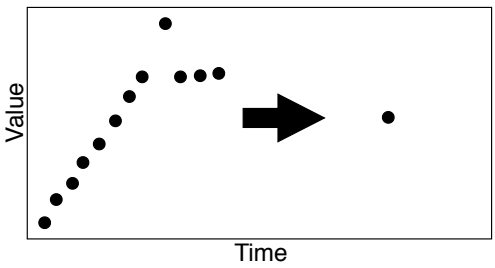

Figure 1: Compression of time series using simple aggregates

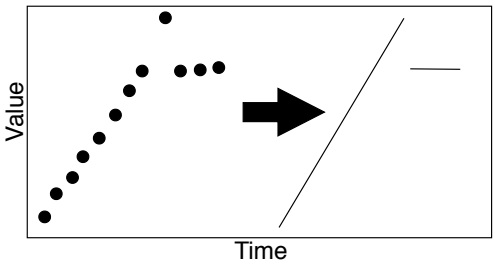

Figure 2: Model-based compression of time series

correlated time series should be modeled together to further reduce the amount of storage required. The benefit of model-based storage for time series is illustrated in Figure 2. The values of these time series can be efficiently represented using a linear function and a constant function which in total only requires three floating-point values while preserving the structure of the time series. Many aggregate queries can be answered directly from models, for example, `MIN`, `MAX`, `SUM` and `AVG` can all be computed in constant time for both constant and linear functions. As such, a TSMS using a model-based approach for time series management should require less storage and provide faster query processing compared to existing TSMSs that store time series as raw data points. This thesis explores the idea of employing a model-based approach for times series management and proposes the distributed TSMS ModelarDB which compared to existing storage solutions achieves faster ingestion speeds, better compression, and faster query performance on aggregate queries at scale.

1.2 Thesis Structure

The thesis is organized as follows:

Part I provides the motivation for the thesis and a summary of the four papers included therein. Section 2 summarizes Paper A [42] and presents a set of paramount properties for a TSMS to efficiently manage high quality time series, provides an overview of TSMSs proposed by academia and through industrial research, and presents examples of TSMSs with unique functionalities. In addition, it summarizes our proposal for research into a model-based TSMS which is based on the surveyed systems and future research directions proposed by other researches in the field. Section 3 summarizes Paper B [43] and describes methods and algorithms for model-based management of individual time series as implemented in ModelarDB. Section 4 summarizes Paper C [41] and describes methods and algorithms for model-based management of correlated time series as implemented in ModelarDB. Section 5 provides a description of the general architecture for a model-based TSMS underlying ModelarDB, how model-based storage and query processing are implemented in ModelarDB at the time of writing, and summarizes Paper D [44] to describe the functionality provided by ModelarDB from a user's perspective. Finally, Section 6 provides a summary of the

contributions made by the thesis, while Section 7 proposes directions for future work.

Part II reproduces the four papers in full with only their layout revised to fit the format of this thesis. As each paper builds on the previous, it is recommended to read the papers sequentially from Paper A to Paper D.

2 Existing Time Series Management Systems

2.1 Motivation and Problem Statement

As shown in Table 1, TSMSs are more suitable for storing large quantities of time series than general purpose RDBMSs, as TSMSs are specialized systems optimized for a specific use-cases and as a result capable of achieving greater performance [79]. However, as TSMSs focus on time series management, their performance often comes at the cost of functionality required for general purpose RDBMSs, such as support for transactions and the ability to update existing data. As the existing storage solutions used in industry cannot manage the large quantities of high quality sensor data produced by entities in the energy domain, we perform a thorough literature survey of TSMSs. As such, Paper A [42] addresses the problem of determining the current state-of-the-art in regard to TSMSs and if a TSMS suitable for managing large quantities of high quality time series exists.

2.2 Paramount Properties and Classification Criteria

Based on existing literature and discussions with owners and manufacturers of wind turbines, we propose the following set of paramount properties [43] for a TSMS to efficiently manage large quantities of high quality sensor data: (i) **Distribution:** a distributed architecture is required due to the large amounts of sensor data being produced. (ii) **Stream Processing:** data points received by the system must be available for query processing with only a small amount of lag when monitoring physical entities. (iii) **Compression:** as analysis of historical data can reveal performance degradation or indicate problems regarding the monitored entities, the sensors should be sampled at a high frequency and the raw data should be stored instead of simple aggregates. However, due to the amount of storage required to do so, state-of-the-art compression must be provided. Compression also reduces query processing time due to a reduction in disk I/O. (iv) **Efficient Retrieval:** partitioning, time-ordered storage, or indexing is required to ensure a low query response time when analyzing a subset of a large data set. (v) **AQP:** approximation is required to both decrease query response time and enable use of lossy compression which generally outperforms lossless compression [36]. (vi) **Extensibility:** as researchers and domain experts continue to develop new and improved model types, the system must be easily extensible without requiring users to change the TSMS itself and the system should be able to automatically use the most appropriate model type for each sub-sequence of a time series.

Based on these paramount properties and how the surveyed TSMSs differ, we define the following set of classification criteria [42]: (i) **Architecture:** is the primary classification criteria due to the impact it has on the system if it uses an *internal* data store, an *external* data store, or is an *RDBMS extension*. (ii) **Year:** is the publication year of the most recent publication about the system and is included to simplify comparing systems from the same time period. (iii) **Purpose:** indicates what trade-offs have been performed as this differs depending on the system's intended usage, which is either *monitoring* entities, receiving data from *IoT* devices, performing *data analytics* on time series, or performing an *evaluation* of new research. (iv) **Motivational Use Case:** is the use case driving development of the system and is another indicator of the trade-offs that were made. (v) **Distributed:** systems are classified based on their capability to scale through *distributed* computing or if they only support running *centralized* on a single machine. (vi) **Maturity:** describes if a TSMS is a *proof-of-concept implementation* of new research, a *demonstration system* that implements functionality so users can interact with it, or is a *mature system* that is deployed to solve real-life problems and is supported by an open-source project, a company, or both. (vii) **Scale Shown:** is an indicator of how large deployments the system have been evaluated with, measured in the *number of nodes* deployed and the *size of the largest data set* used. (viii) **Processing Engine:** is either an *existing system* or is *implemented* from scratch for a TSMS. (ix) **API:** is the primary methods for using a TSMS and are *query languages*, *extended query languages*, *client libraries*, a *web service*, or a *web interface*. (x) **Approximation:** is methods provided by a TSMS for approximating time series, if any, using either simple *aggregates* that provide no error guarantees, or AQP with error guarantees provided using either *sampling* or *mathematical models*. (xi) **Stream Processing:** methods are provided by each system, if any, through either *user-defined functions*, as *functionality part of the TSMS*, or as a *query interface based on streams*. (xii) **Storage Engine:** is the component of the system used for data storage and can be *implemented* for the TSMS or be an *existing system*. (xiii) **Storage Layout** is the representation a TSMS uses internally to store time series. Each system is categorized with regard to these criteria and a summary is shown in Table 2.

2.3 Internal Data Stores

Some systems implement both the data storage and the query processing components in one application. This results in tight integration between these components. As no external application has direct access to the data store, it can be optimized exclusively to fit the needs of the query processing component, and the query processing component can dictate what interface should be provided unless an embedded Database Management System (DBMS) with a pre-defined interface is used. In addition, these systems are not required to use an existing method for transferring data between the data storage and query processing components, such as JSON or XML. Instead, if data serialization is required, a system can define and optimize a data serialization method for the TSMS's intended workload. No dependencies have to be deployed and

Table 2: Overview of the surveyed systems using the proposed classification criteria [42]

	Year	Purpose	Motivationa Use Case	Distributed	Maturity	Scale Shown	Processing Engine	API	Approximation	Stream Processing	Storage Engine	Storage Layout
Internal Stores												
tsdb [17]	2012	Monitoring	Monitoring of large computer networks at high resolution.	Centralized	Mature System	6.3 GB 1 Node	Implemented (C)	Client Library (C)	No support for approximation of time series.	No stream processing.	BerkeleyDB	Fixed sized arrays stored as values in BerkeleyDB.
FAQ [47]	2014	Evaluation	Efficient approximate queries on time series of histograms.	Centralized	Proof-of-Concept Implementation	1.9 GB 1 Node	Implemented (Java)	Unknown	Model-based AQP	No stream processing.	KyotoCabinet	Sketches and histograms organized in a range tree.
WearDrive [34, 53]	2015	IoT	Power efficient storage of data produced by wearable sensors.	Distributed	Demonstration System	Unknown 2 Nodes	Implemented (C, Java, JNI)	Client Library (C, Java, JNI)	No support for approximation of time series.	Register callbacks waiting for sensor readings.	Implemented (C, Java, JNI)	In-memory log of KV-pairs indexed by a hash table.
RINSE [86–88]	2015	Data Analytics	Querying time series without constructing a full index first.	Centralized	Demonstration System	1 TB 1 Node	Implemented (C)	Drawn nearest neighbor search.	Model-based AQP	No stream processing.	Implemented (C)	ADS+ tree indexing an unspecified ASCII format.
Unnamed [67]	2015	Evaluation	Fast approximated queries for decision support systems.	Centralized	Proof-of-Concept Implementation	Unknown 1 Node	Implemented (R)	Extended SQL	Model-based AQP used for aggregate queries.	No stream processing.	Implemented (R)	Separated storage of both raw data and models.
Plato [45]	2015	Data Analytics	Simple analysis of spatiotemporal data with signal processing.	Centralized	Demonstration System	Unknown 1 Node	Implemented (Unknown)	Extended SQL	Model-based AQP	No stream processing.	Implemented (Unknown)	Tables with models stored as a built-in data type.
Chronos [12, 13]	2016	Monitoring	Monitor hydroelectric plants using PCs with flash memory.	Centralized	Demonstration System	11.5 GB 1 Node	Implemented (C++)	Client Library (C++)	No support for approximation of time series.	Out-of-order inserts.	Implemented (C++)	B-Tree index over quasi-sequential data points.
Pytsms [73, 74]	2016	Evaluation	Reference implementation of two formalisms for time series.	Centralized	Proof-of-Concept Implementation	Unknown 1 Node	Implemented (Python)	Client Library (Python)	User-defined aggregates at multiple resolutions.	No stream processing.	Implemented (Python)	Python objects pickled to a file or serialized to CSV.
PhilDB [54]	2016	Data Analytics	Storage and analysis of versioned mutable time series.	Centralized	Demonstration System	119.12 MB 1 Node	Implemented (Python), Pandas	Client Library (Python)	No support for approximation of time series.	No stream processing.	Implemented (Python), SQLite	Triples stored as binary files, updates as HDF5.

Table 2: (Continued) [42]

	Year	Purpose	Motivationa Use Case	Distributed	Maturity	Scale Shown	Processing Engine	API	Approximation	Stream Processing	Storage Engine	Storage Layout
External Stores												
TSDS [83]	2010	Data Analytics	Querying data in multiple formats from one end-point.	Centralized	Mature System	Unknown 1 Node	Implemented (Java)	REST API serving multiple formats.	AQP using sampling.	No stream processing.	Implemented (Java) for caching.	A binary file with 64-Bit values, metadata as NcML.
SciDB [77, 78, 80]	2013	Data Analytics	Storage and querying of data from scientific instruments.	Distributed	Mature System	Unknown 16 Nodes	Implemented (C++), ScaLAPACK	Array languages AFL and AQP.	AQP using sampling.	Arrays piped through external processes.	Implemented (C++), PostgreSQL	N-Dimensional arrays of tuples, stored in chunks.
Respawn [10, 59]	2013	Monitoring	Executing low latency queries across cloud and sensor nodes.	Distributed	Demonstration System	21 GB 10,000 Nodes	Bodytrack Datastore	REST API serving JSON.	Predefined aggregates at multiple resolutions.	No stream processing.	Bodytrack Datastore	Bodytrack Datastore's compressed binary format.
SensorGrid [15]	2013	Data Analytics	OLAP of sensor data through AQP and grid computing.	Distributed	Demonstration System	Unknown 5 Nodes	Implemented (Unknown)	Web Interface, SQL	User-defined aggregates at multiple resolutions.	Window queries.	RDBMS	Data points, aggregates in two-dimensional array.
Unnamed [28–30]	2014	Evaluation	Indexing mathematical models in a distributed KV-store.	Distributed	Proof-of-Concept Implementation	12 GB 9 Nodes	Apache Hadoop	Unknown	Model-based AQP	Real-time modelling of segmented time series.	Apache HBase	In-memory binary trees indexing models in HBase.
Tristan [55, 56]	2014	Data Analytics	Enable creation of analytical applications using sensor data.	Centralized	Demonstration System	Unknown 1 Node	HYRISE	Unknown	Model-based AQP	Real-time modelling of segmented time series.	HYRISE	Sparse representation after dictionary compression.
Druid [85]	2014	Data Analytics	Ingestion and exploration of complex events from log data.	Distributed	Mature System	100 GB 6 Nodes	Implemented (Java)	REST API serving JSON.	User-defined aggregates and model-based AQP.	No stream processing.	Implemented (Java), DFS	Immutable columns encoded based on their type.
Unnamed [35]	2014	Monitoring	Query both time series and relational data through SQL.	Distributed	Mature System	0.90 TB 1 Node	IBM Informix	SQL	Model-based AQP	Real-time modelling of segmented time series.	IBM Informix	Blobs of values, or timestamp deltas and values.
Unnamed [84]	2014	Monitoring	Remote monitoring of industrial installations in real-time.	Distributed	Mature System	5 TB 46 Nodes	GE Streaming Engine, Pivotal GemFire	OQL and Client Library (Java)	No support for approximation of time series.	Real-time data transfor-mations and analytics.	Pivotal Gemfire	Ordered KV-pairs storing segments as linked lists.

Table 2: (Continued) [42]

	Year	Purpose	Motivationa Use Case	Distributed	Maturity	Scale Shown	Processing Engine	API	Approximation	Stream Processing	Storage Engine	Storage Layout
Bolt [31]	2014	IoT	Simplify management of IoT data from connected homes.	Distributed	Demonstration System	37.89 MB 1 Nodes	Implemented (C#)	Client Library (C#)	AQP using sampling.	Data is written to and read from streams.	Implemented (C#), S3, Azure	In-memory index of continuous chunks on disk.
Storacle [11, 22]	2015	Monitoring	Enable local processing of data at the edge of sensor networks.	Distributed	Demonstration System	541 MB 1 Node	Implemented (Java), Cloud	Unknown	Predefined aggregates.	Online computation of simple aggregates.	Implemented (Java), Cloud	Unknown in-memory format and Protocol Buffers.
Gorilla [66]	2015	Monitoring	Reducing query latency for a real-time monitoring system.	Distributed	Mature System	1.3 TB 20 Nodes	Implemented (C++)	Client Library (Unknown)	No support for approximation of time series.	No Stream processing.	Implemented (C++), DFS, HBase.	Blocks of deltas prefixed by a single timestamp.
Unnamed [57]	2015	Data Analytics	Execute aggregate queries on resource constrained systems.	Distributed	Mature System	3 TB Unknown	Implemented (Java), MySQL	SQL	Model-based AQP used for aggregate queries.	Online computation of aggregate models.	MySQL	Binary tree storing aggregates as models or sums.
servIoTicy [68, 81]	2015	IoT	Integration of stream processing and data storage for IoT.	Distributed	Demonstration System	Unknown 16 Nodes	Couchbase, Apache Storm, Elasticsearch	REST API serving JSON.	No support for approximation of time series.	Apache Storm extended with versioning of bolts.	Couchbase	JSON documents with ids indexed by Elasticsearch.
BTrDB [2, 3]	2016	Monitoring	Analyzing data with ns timestamps at multiple resolutions.	Distributed	Mature System	2.757 TB 2 Nodes	Implemented (Go)	Client Library (Go, Python)	Predefined aggregates at multiple resolutions.	Data is written to and read from streams.	DFS, MongoDB	Versioned tree with aggregates in internal nodes.
RDBMS Extensions												
TimeTravel [23, 46]	2012	Data Analytics	Continues forecasting of power consumption in a smart grid.	Centralized	Demonstration System	Unknown 1 Node	PostgreSQL	Extended SQL	Model-based AQP	No stream processing.	PostgreSQL	Data points in arrays with layers of models on top.
F^2DB [24, 25]	2012	Data Analytics	Forecasting directly integrated as part of a data warehouse.	Centralized	Demonstration System	Unknown 1 Node	PostgreSQL	Extended SQL	Model-based AQP for forecast queries.	No stream processing.	PostgreSQL	Tables and a hierarchy of models built on top.
Unamed [8, 9, 50]	2016	Evaluation	OLAP analysis of time series as ordered sequences of events.	Centralized	Proof-of-Concept Implementation	23 MB 1 Node	Oracle RDBMS	Extended SQL	Model-based AQP for interpolated data.	No stream processing.	Oracle RDBMS	Tables for both raw data and model parameters.

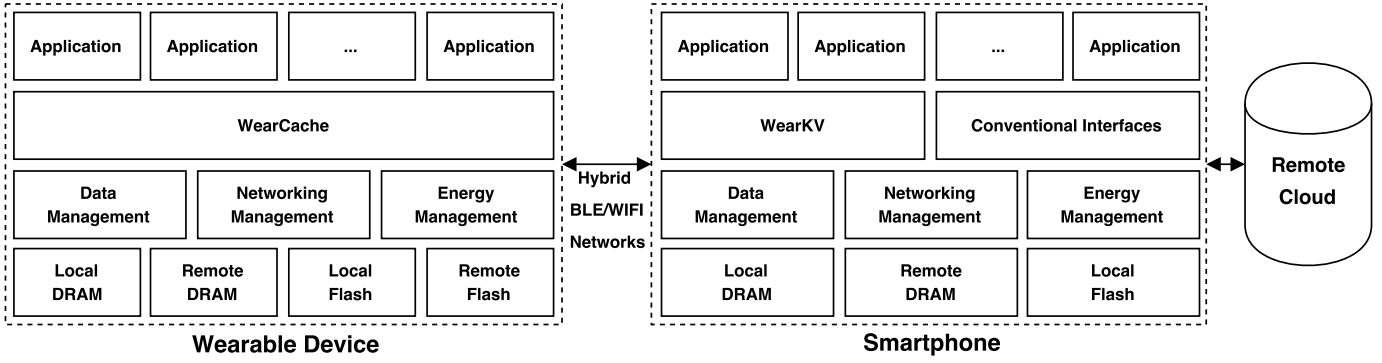

Figure 3: WearDrive uses a split architecture with remote resources located on the other device [42]. The figure was redrawn from [34]

configured as these systems are self-contained, simplifying their deployment. On the other hand, these TSMSs cannot reuse existing infrastructure already deployed, such as a Distributed File System (DFS) or a distributed DBMS. In addition, if a new data store is developed, effort will have to be spent ensuring the system's durability and fault-tolerance, and administrators will have to learn how to tune it for each use-case.

TSMSs using an internal data store are generally developed as research prototypes and most them are not distributed. Of the nine systems surveyed, six are research prototypes created to evaluate new theoretical methods while only three are developed as general purpose systems and have mature enough implementations to fulfill this task based on the surveyed papers. As such, systems in this category are generally small scale prototypes instead of mature systems intended for actual deployment. This category includes two unique systems; WearDrive [34] and Plato [45]. WearDrive [34] is a TSMS designed to reduce the power consumption of wearable devices, like a smart watch, communicating with a smartphone. Previous work show that the main consumer of power in these smart devices are the flash storage [53]. As such, WearDrive demonstrates that the battery life of wearable devices can be extended by keeping all data in memory and persisting data by transferring it to a smartphone instead of using local flash storage. The system is developed as two separate applications; one running on a smartphone and the other running on a wearable device. This architecture is shown in Figure 3. By modifying the firmware of the wearable device, WearDrive provides the same persistence guarantees as using local flash storage would and increases the battery life of the wearable device at the same time, despite using remote instead of local flash storage. Plato [45] is a TSMS designed as a new RDBMS integrating methods from signal processing to support data cleaning and data analysis. The TSMS allows users to create and query model-based representations of time series instead of the raw data. Plato uses a three layer architecture as shown in Figure 4 with each layer providing

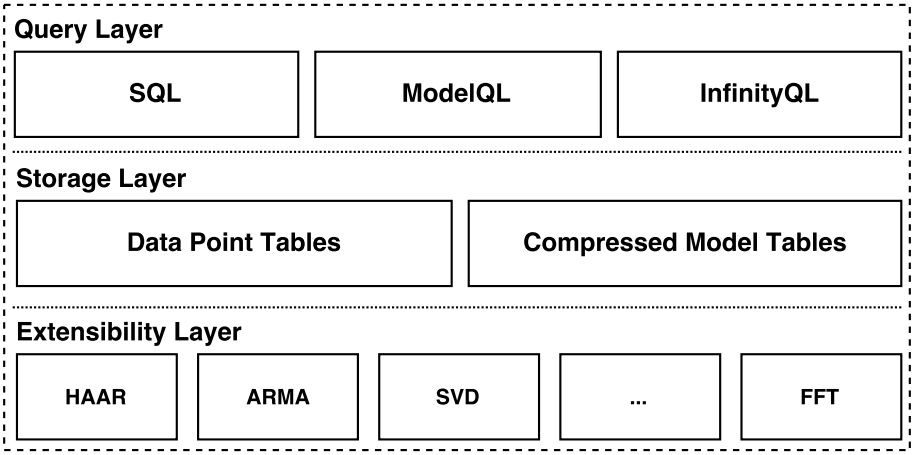

Figure 4: Plato's architecture is split into three layers [42], redrawn from [45]

abstractions for implementation, fitting, or querying models. Users can implement model types through the Extensibility Layer. Users can then select which model types to use for representing each time series using the Storage Layer. The Query Layer allow users to query the data sets using either the raw data or the model-based representation. To accommodate users with different backgrounds, Plato provides two query languages; ModelQL and InfinityQL. ModelQL is designed for statisticians, treats models as black boxes, and allows users to apply functions on the models. InfinityQL is designed for users with a database background and allows users to query the models as if they are relations with an infinite number of tuples. By integrating models directly into the TSMS, Plato allows time series data to be analyzed in the system without first transferring the data set to an external tool for time series analysis like sktime [76] or R [70].

2.4 External Data Stores

TSMSs that rely on an *external* storage solution must implement methods for transferring data between the data storage and the query processing application. A TSMS reusing an external data store benefits from the engineering time that is spent on optimizing and making these data stores durable. Administrators can also tune these systems based on experience gained from existing deployments. However, splitting a TSMS into a separate data storage and query processing application complicates deployment as both now have to be managed and tuned for a particular workload.

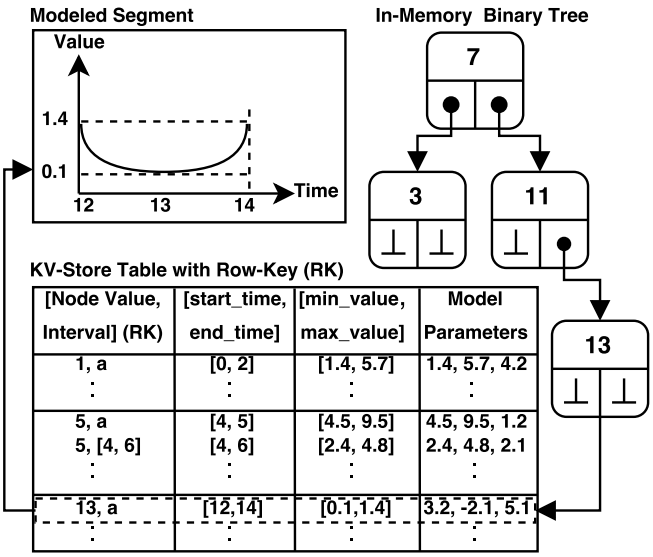

Figure 5: Time series are stored as models in a distributed key-value store and indexed by an in-memory tree [42], redrawn from [29]

In addition, the query processing component is restricted to the existing data model provided by the data store and can only access the data through the interfaces provided.

There are fifteen systems in this category, and they are generally mature distributed systems deployed to solve real-life tasks. Only two systems are not distributed. All the systems make use of an existing data store such as Apache HBase, HYRISE, MySQL or Couchbase. Twelve of the systems provide some form of approximation of the time series. However, most of the systems that provides AQP have limited support for adding additional models or are simple research prototypes. Few systems in this category are truly unique as they are primarily developed to solve real-life problems instead of functioning as research prototypes. As such, these systems generally represent time series as ordered sequence of data points stored with some form of lossless compression. However, the TSMS proposed by Guo et al. [28–30], Tristan [55, 56], and BTrDB [2, 3] present interesting approaches for time series management. The TSMS proposed by Guo et al. [28–30] stores time series as models to achieve high compression and fast query processing performance through model-based AQP. The TSMS store the models in Apache HBase using one table ordered by time and another ordered by value. In-memory binary trees index both tables, allowing a MapReduce based query processing algorithm to efficiently extract the models required by each query and reconstruct the necessary data points. Query processing is shown in Figure 5 and starts with traversing the relevant binary tree to determine which models to extract. Mappers then read these models from Apache HBase and prune models irrelevant to

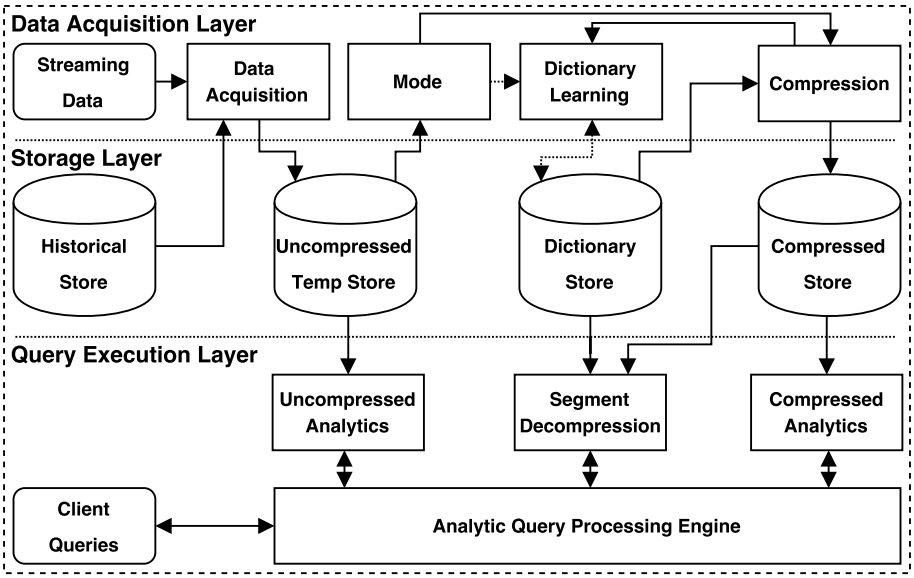

Figure 6: Tristan is based on the MiSTRAL architecture. The dotted arrows indicate data flow only present when creating dictionaries in offline mode [42]. The figure is redrawn from [55]

the query. Reducers then reconstruct the data points represented by each model to create an approximation of the original time series. This TSMS proposes an interesting model-based approach to store time series, however, the system is limited by only representing time series as constant functions and by always reconstructing data points instead of computing aggregates from the models. Tristan [56] is a TSMS based on the MiSTRAL architecture [55] and implemented using HYRISE. The system stores time series using dictionary compression and supports executing queries on top of the compressed representation. The MiSTRAL architecture is shown in Figure 6. The Data Acquisition Layer combined with the Storage Layer ingests data points into statically sized segments which are then represented as a sequence of smaller fixed size time series using dictionary compression. Dictionaries with smaller fixed size time series appropriate for compressing a particular data set are created offline. The system then adjusts the dictionaries at specific time intervals during run-time. The Query Layer extracts data from the Storage Layer to answer queries using one of three options; data points from the Temporary Store, compressed segments, or decompressed segments depending on the query and if the data have been compressed. Tristan proposes an interesting method for AQP with the downside that the dictionary must be trained offline. BTrDB [2, 3] is designed to manage large quantities of time series with nanosecond resolution timestamps produced by high precision power meters. The system stores time series as copy-on-write k-ary trees as shown in Figure 7. Data points are efficiently stored sequentially in the leaves, while the internal nodes store statistics about the data to reduce query processing time. As the tree is copy-on-write the internal nodes also

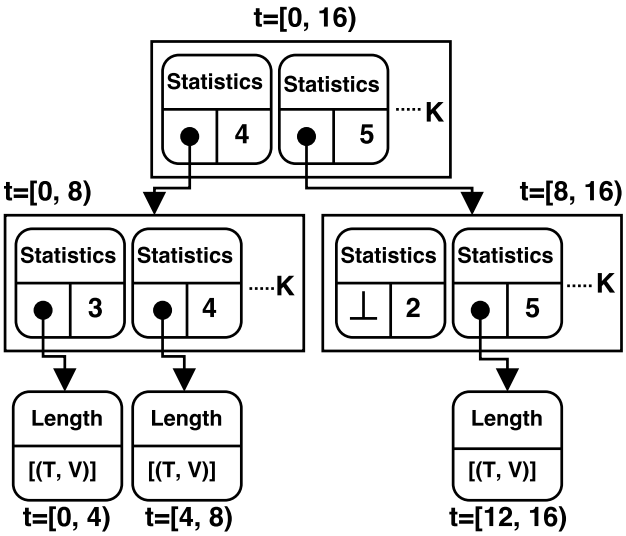

Figure 7: BTrDB uses a copy-on-write K-ary tree storing data points in the leaves with statistics and version numbers stored in internal nodes [42], redrawn from [2]

store version numbers to support queries on older versions of the data. While BTrDB provides fast ingestion rates and query performance at multiple resolutions, its use of two iterations of delta-encoding combined with Huffman encoding for compression only achieves a 2.93 times reduction in the amount of storage required compared to storing the data without compression using 16 bytes per data point.

2.5 RDBMS Extensions

Implementing a TSMS as an *RDBMS extension* allows reuse of existing functionality from the RDBMS without requiring the data to be transferred between two separate applications. In addition, there exists a substantial amount of work on how to tune most RDBMSs, simplifying the work of the administrator. Extending a RDBMS comes with the trade-off that the TSMS must adhere to the limitations of the RDBMS in terms of data model and extensibility, in addition to the performance overhead and added complexity of features irrelevant for a time series workload, such as transactions.

Out of the three TSMSs implemented as RDBMS extensions, both TimeTravel [23, 46] and F^{2}DB [24, 25] present interesting ideas for integrating forecasting into a RDBMS instead of relying on external tools such sktime [76] or R [70]. However, they are only mature enough to be considered demonstration systems that show the effectiveness of such functionality with a real-life use-case. To support forecasting, TimeTravel [46] extends PostgreSQL with the ability to construct a hierarchy of forecasting models which the system then indexes and uses for query processing. The architecture of TimeTravel is shown in Figure 8. The model hierarchy is constructed

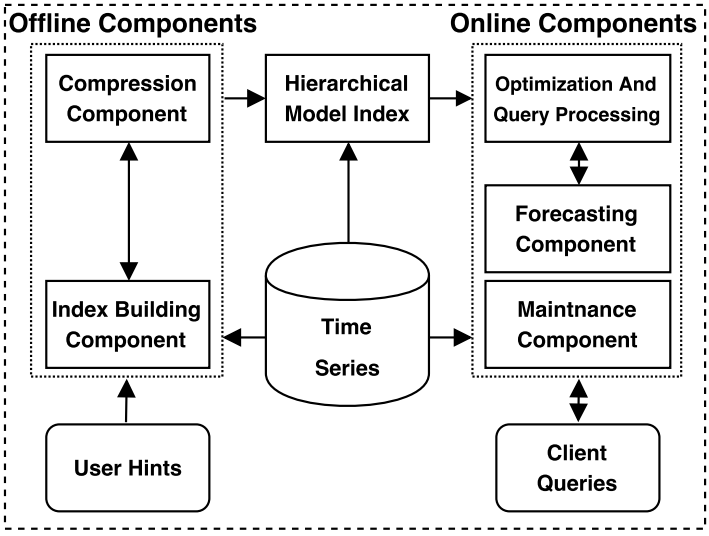

Figure 8: TimeTravel allows a hierarchy of model for forecasting being constructed offline and then maintains them online [42]. The figure was redrawn from [46]

offline and then maintained online by the system. A hierarchy of models at different granularities is used to answer queries with different error bounds efficiently. To build the hierarchy the user must provide the error bound guarantees required, which model types to use for forecasting, and hints about the seasonality of the time series. Constructing the model hierarchy enables AQP, as TimeTravel supports both exact and approximate queries on historical data and approximate queries on forecasted values. TimeTravel is limited in terms of scalability due to it being a single node system.

F^2DB [24] also extends PostgreSQL with support for forecasting but focuses on data warehouse style analytics. In contrast to TimeTravel, F^2DB [24] does not require user-hints to construct models from data points but can use a set of evaluation criteria to determine the most appropriate model for a particular time series. Users can also manually choose to represent a time series with a specific model type. The system automatically maintains the created models. As F^2DB uses a generic interface for model types, domain experts can optionally implement more. A model advisor assists the user by suggesting alternative configurations of models for a query workload [24, 25]. The model advisor is a separate component from which model configuration can be loaded into F^2DB, as shown in the architecture in Figure 9. Like TimeTravel, F^2DB is a single node system and limited in terms of scalability.

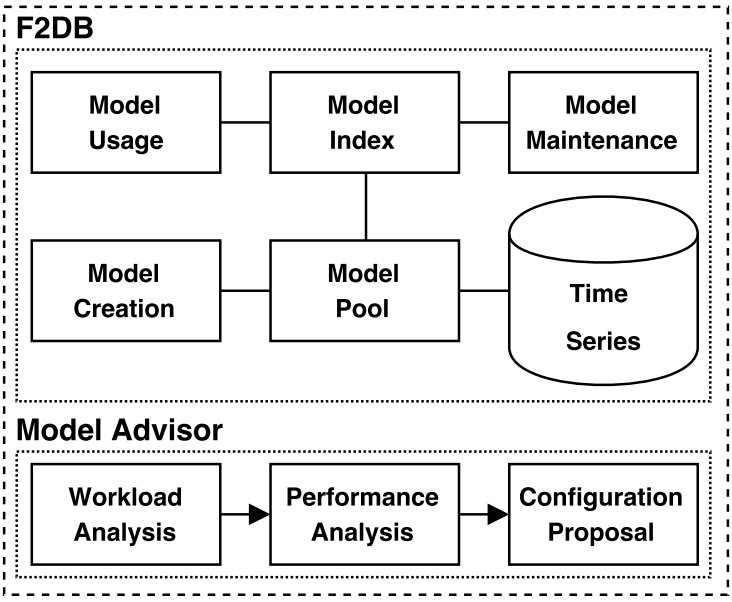

Figure 9: F^2DB TSMS allow forecasting models to be constructed from data stored in relations, which the system then indexes and use for query processing [42]. The figure was redrawn from [24]

2.6 Future Research Directions

Based on the survey, it is clear that none of the systems fulfill the paramount properties for managing high quality time series. The existing TSMSs generally lack the capability to scale through distributed storage and distributed query processing, have limited support for AQP, or cannot be extended by domain experts to support new approximation methods optimized for a particular domain without changing the TSMS. As such, we propose that research into a purely *model-based TSMS* be conducted, with the goal of developing a TSMS utilizing a model-based physical storage layer instead of storing raw data points. The complete TSMS should be *scalable through distributed storage and query processing*, and provide *automatic model management* to automatically store each time series using the model types providing the highest degree of compression so both the storage requirement and query processing time is reduced. Query processing time should be further reduced through efficient retrieval of time ordered or indexed models, and by *answering queries directly from the stored models using model-based AQP* instead of reconstructed data points. To ensure received data points are available for query processing with only a small amount of lag, the system must allow *queries to be executed online* while data is being ingested. A robust notion of metadata about each time series must be provided so the TSMS can function as a *model-based sensor data warehouse* and support multi-dimensional analysis of time series. Finally, the system should be *extensible* so that domain experts can define new model types that are optimized for their specific use-cases.

3 Management of Individual Time Series

3.1 Motivation and Problem Statement

As none of the existing systems fulfill all of the paramount properties required for a TSMS to efficiently manage high quality time series, a new TSMS is required [7, 42]. Since the compression ratio provided by model-based representations of time series generally is higher than general purpose lossless compression [72], a model-based approach with a user-defined error bound (possibly zero) should be considered for time series management. Different model types have been developed for specific use-case, but each of them also have drawbacks [36, 72, 82]. As time series change over time, each model type is only efficient for representing specific sub-sequences of a time series. As an example, a constant function only requires one floating-point value to represent a sequence of approximately constant values but deteriorates into a stair-step pattern for increasing or decreasing sequences. Such sequences can be efficiently represented by a linear function using only two floating-point values. However, if an approximately constant sequence is represented using a linear function, it also requires two floating-point values. In other words, using a linear function requires an unnecessary floating-point value compared to using a constant function for this sequence. To address this issue, multiple methods for Multi-model Compression

(MMC) have been proposed. MMC represents a time series using multiple different model types so each sub-sequence of a time series is represented by the model type that provides the highest compression ratio [18, 64, 65, 69]. However, these methods either limit the model types that can be used with the method [18, 69], provide no upper-bound for the latency of ingested data points [18, 69], or require that the user make a trade-off between low latency and high compression ratio [64, 65]. As such, Paper B [43] addresses two major problems. Firstly, how can a MMC method be created that removes the latency limitations of existing methods without compromising the compression ratio and supports user-defined model types? Finally, how can a TSMS be created that fulfill all the paramount properties and use a model-based physical storage layer instead of storing raw data points?

3.2 Preliminaries

As stated, the high quality time series produced in the energy domain have a regular sampling interval, and the few incorrect or out-of-order data points that do occur are corrected using existing cleaning procedures. For this reason, the only anomaly that must be considered when ingesting these time series are missing data points. While interpolation can be utilized to approximate the missing values, their absence can indicate problems with the monitored entity. For this reason, interpolation of missing values should be an explicit decision performed by a data analyst, and not an implicit action performed by the TSMS. These regular time series are defined as follows using definitions reproduced with modifications from Paper B [43]:

Definition 1 (Time Series)
A *time series* TS is a sequence of *data points*, in the form of timestamp and value pairs, ordered by time in increasing order $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$. For each pair (t_i, v_i), $1 \leq i$, the timestamp t_i represents the time when the value $v_i \in \mathbb{R}$ was recorded. A time series $TS = \langle (t_1, v_1), \ldots, (t_n, v_n) \rangle$ consisting of a fixed number of n data points is a *bounded time series*.

Definition 2 (Regular Time Series)
A time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ is considered *regular* if the time elapsed between each data point is always the same, i.e., $t_{i+1} - t_i = t_{i+2} - t_{i+1}$ for $1 \leq i$ and *irregular* otherwise.

Definition 3 (Sampling Interval)
The *sampling interval* of a regular time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ is the time elapsed between each pair of data points in the time series $SI = t_{i+1} - t_i$ for $1 \leq i$.

As an example, the time series $TS_e = \langle (100, 0.156), (200, 0.139), (300, 0.122), (400, 0.106), (500, 0.92) \ldots \rangle$ is a regular unbounded time series with a 100 milliseconds sampling interval. A bounded time series can be constructed from TS_e by extracting the data points with the timestamps $100 \leq t \leq 500$.

Definition 4 (Gap)
A *gap* between a regular bounded time series $TS_1 = \langle (t_1, v_1), \ldots, (t_s, v_s) \rangle$ and a regular time series $TS_2 = \langle (t_e, v_e), (t_{e+1}, v_{e+1}), \ldots \rangle$ with the same sampling interval SI and recorded from the same source, is a pair of timestamps $G = (t_s, t_e)$ with $t_e = t_s + m \times SI$, $m \in \mathbb{N}_{\geq 2}$, and where no data points exist between t_s and t_e.

Definition 5 (Regular Time Series with Gaps)
A *regular time series with gaps* is a regular time series, $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ where $v_i \in \mathbb{R} \cup \{\perp\}$ for $1 \leq i$. For a regular time series with gaps, a gap $G = (t_s, t_e)$ is a sub-sequence where $v_i = \perp$ for $t_s < t_i < t_e$.

No gaps are present in TS_e as a value exist for all timestamps matching the sampling interval. However, $TS_g = \langle (100, 0.156), (200, 0.139), (400, 0.106), (500, 0.92) \ldots \rangle$ have no value for $t = 300$ and is two regular time series separated by a gap. For simplicity, we say that multiple time series from the same source separated by gaps is one time series containing gaps. As the sampling interval is undefined for a time series with gaps, we define regular time series with gaps as a time series where all data points in a gap have the special value $\perp$ indicating that no values were collected. As an example, $TS_{rg} = \langle (100, 0.156), (200, 0.139), (300, \perp), (400, 0.106), (500, 0.92) \ldots \rangle$ have a sampling interval of 100 milliseconds.

Definition 6 (Model)
A *model* is a representation of a time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ using a pair of functions $M = (m_{est}, m_{err})$. For each t_i, $1 \leq i$, the function m_{est} is a real-valued mapping from t_i to an estimate of the value for the corresponding data point in TS. m_{err} is a mapping from a time series TS and the corresponding m_{est} to a positive real value representing the error of the values estimated by m_{est}.

Definition 7 (Segment)
For a bounded regular time series with gaps $TS = \langle (t_s, v_s), \ldots, (t_e, v_e) \rangle$ with sampling interval SI, a *segment* is a 6-tuple $S = (t_s, t_e, SI, G_{ts}, M, \epsilon)$ where G_{ts} is a set of timestamps for which $v = \perp$ and where the values of all other timestamps for TS are defined by the model M within the error bound ϵ.

A model type determines the set of parameters required to create a specific model of that type for approximating the values of a time series. Models represent the values of a time series as the function m_{est} with the error of the representation given by the error function m_{err}. We say that a model is fitted to a bounded regular time series, for example $TS_b = \langle (100, 0.156), (200, 0.139), (300, 0.122), (400, 0.106), (500, 0.92) \rangle$, when determining the parameters of a model using a model type. To ensure the error bound of the model-based representation is upheld for all data points in a time series, it can be necessary to split the time series into smaller segments each represented by a different model, and possibly a different model type if MMC is used.

3.3 Model-Agnostic Time Series Compression

To represent time series using multiple different model types online, we propose a method that evaluates different model types in sequence. It switches to the next model type when the current one can no longer fit a model to the ingested data points within the user-defined error bound. This approach is similar to the one proposed in [18], but our approach is model-agnostic and provides latency guarantees. Our method discards the timestamps as they can be reconstructed from the information stored as part of each segment. An example is shown in Figure 10.

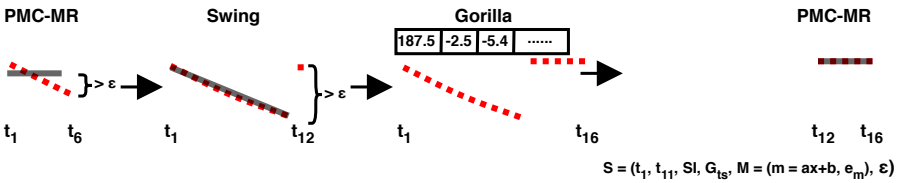

Figure 10: Model-agnostic time series compression using MMC (Updated) [41]

In this example, three model types are used: the PMC-MR model type that represents time series as constant functions [52], the Swing model type that represents time series as linear functions [21], and a model type using the lossless floating-point compression algorithm proposed by Facebook for their Gorilla TSMS [66]. From t_1 to t_5 PMC-MR can fit a model to the ingested data points within the user-defined error bound ϵ, but at t_6 a data point is ingested for which PMC-MR would exceed the error bound. As such, our method switches to the Swing model type which can represent the ingested data points within the error bound until t_{12}. To accommodate this, our method escalates to the Gorilla model type, which is the last model type in this example, and continues ingesting. As Gorilla uses lossless compression it cannot exceed the error bound, instead it is limited by a user-configurable length limit, 15 in this example. As a result, at t_{16} all model types have exceeded their respective bounds and the model type with the best compression ratio is selected, in this example Swing. As such, a model of type Swing is used to represent the sub-sequence from t_1 to t_{11} as part of a segment. Afterwards, ingestion is restarted with PMC-MR from the first data point not part of the previous segment, which is t_{12} in this example. As a model type might represent all data points in a time series, the guaranteed latency is unbounded with this method. Expanding the user-configurable length limit to apply to all model types will provide a guaranteed upper bound for this latency. However, this reduces the compression ratio for lossy models that can be fitted to more data point than the set limit. As an alternative, queries can be executed on the raw data points but this approach lacks the benefits provided by a model-based representation. To solve this problem, we propose two types of segments; *finalized segments* and *temporary segments*.

A finalized segment is persisted to disk when all model types cannot represent the latest data point, as shown at t_{16} in Figure 10. Temporary segments are only

Algorithm 1 Our MMC algorithm that is model-agnotic, as it supports lossy and lossless model types, and provides high compression and latency guarantees [43]

```
1:  Let ts be the time series of data points.
2:  Let models be the list of models to select from.
3:  Let error be the user defined error bound.
4:  Let limit be the limit on the length of each segment.
5:  Let latency be the latency in not emitted data points.
6:  Let interval be the sampling interval of the time series.
7:
8:  model ← head(models)
9:  buffer ← create_list()
10: yet_emitted ← 0
11: previous ← nil
12: while has_next(ts) do
13:     data_point = retrieve_next_data_point(ts)
14:     if time_diff(previous, data_point) > interval then
15:         flush_buffer(buffer)
16:     end if
17:     append_data_point_to_buffer(data_point, buffer)
18:     previous ← data_point
19:     if append_data_point_to_model(data_point, model, error, limit) then
20:         yet_emitted ← yet_emitted + 1
21:         if yet_emitted = latency then
22:             emit_temporary_segment(model, buffer)
23:             yet_emitted ← 0
24:         end if
25:     else if has_next(models) then
26:         model ← next(models)
27:         initialize(model, buffer)
28:     else
29:         emit_finalized_segment(models, buffer)
30:         model ← head(models)
31:         initialize(model, buffer)
32:         yet_emitted ← min(yet_emitted, lengh(buffer))
33:     end if
34: end while
35: flush_buffer(buffer)
```

stored in-memory and contain models that can represent additional data points, but are constructed if the latency exceeds a user-defined limit in terms of data points not available for query processing. Using these two segment types, our method removes

the trade-off between high compression and low latency exhibited by other MMC methods. This is done by storing time series as models with high compression in finalized segments, while also incrementally making temporary segments available for query processing to satisfy a user-defined latency threshold. Our model-agnostic MMC algorithm is shown in Algorithm 1.

The algorithm ingests the time series *ts* one data point at a time and verifies that a gap is not present in Line 12–15. If there is no gap, the data point is stored in *buffer* and the current model is fitted to the new data point in Line 17–19. As the current model now represent another data point yet to be emitted for query processing, the variable *yet_emitted* is incremented in Line 20. When the current model type represents more data points yet to be emitted than the user-defined *latency* limit, a temporary segment is emitted on Line 21–23. If the data point cannot be represented by the current model type, the next model type is selected in Line 25–27, corresponding to t_6 and t_{12} in Figure 10. When the last model type exceeds its bound, a finalized segment is emitted as shown in Line 28–32, corresponding to t_{16} in Figure 10. The number of yet to be emitted data points represented by a finalized segment are subtracted from *yet_emitted* in Line 32. This reduces the number of temporary segments emitted while still providing the required latency guarantee. If a gap is present in the time series, all data points in *buffer* are emitted as finalized segments before ingestion restarts with the remaining data points. While gaps could also be stored in segments as pairs of start time and end time, we found no significant difference in the compression ratio when evaluating the two methods. As such, we decided that no segments should contain missing data points. This simplifies implementation of user-defined model types significantly.

3.4 Model-Based Query Processing

Since all models can reconstruct the data points they represent within the user-defined error bound by definition, any SQL query can be executed on these reconstructed data points as if they were stored directly. However, this adds unnecessary overhead as many aggregate queries can be answered directly from the models.

```
1 public double sum() {
2   int timespan = this.endTime - this.startTime;
3   int size = (timespan / this.SI) + 1;
4   double first = this.a * this.startTime + this.b;
5   double last = this.a * this.endTime + this.b;
6   double average = (first + last) / 2;
7   return average * size;
8 }
```

Listing 1: Implementation of `SUM` for a linear function [43]

As an example, `SUM` can be calculated in constant time for each segment that represents values using a linear function. The procedure implemented for the model type Swing is shown in Listing 1. The number of data points represented by the model is calculated in Line 2–3, while the average is calculated in Line 4–6. Finally, in Line 7

the `SUM` is calculated by multiplying the average with the number of data points the model represents.

Model-based aggregation methods can directly substitute aggregation of reconstructed data points and utilizing them in a TSMS requires only a simple substitution of the operators by the system's dynamic optimizer. The only complication is if a `WHERE` clause restricts an aggregate to a time interval that includes anything but full segments. For such queries, the start time, end time, or both of the relevant segments must be temporarily changed. However, TSMSs built using existing systems for query processing such as Apache Spark or Apache Flink, can be limited by the public APIs that provides access to the query processing system's query plans and dynamic optimizer. For such systems, we propose using a split approach for the query interface by defining both a *Data Point View* and a *Segment View*. The Data Point View allows queries to be executed on reconstructed data points as if they were stored directly by the system using the schema `(Tid int, TS timestamp, Value float)` where `Tid` is the time series identifier, `TS` is the timestamp of the data point, and `Value` is the value of the data point. The Segment View allows queries to be executed at the segment level to support model-based aggregate queries. The Segment View uses the schema `(Tid int, StartTime timestamp, EndTime timestamp, SI int, Mid int, Parameters blob)`, where `StartTime` and `EndTime` correspond to the start and the end time of the segment respectively, `SI` is the sampling interval of the time series the segment was created from, `Mid` is the identifier of the model type used to represent the values of the segment, and `Parameters` are the model parameters. The model parameters are represented as a blob instead of separate columns to make it possible to have model types with arbitrary numbers of parameters. On the segment view, model-based aggregates can be implemented as a set of User-Defined Aggregate Functions (UDAFs). The User-Defined Functions (UDFs) `START`, `END`, and `INTERVAL` must also be implemented so that users can change start time and end time of a segment to allow aggregate queries on the Segment View at the same granularity as provided by the Data Point View. By combining the Segment View extended with UDAFs and the Data Point View, any SQL query can be efficiently executed without knowledge of the underlying model-based implementation. Model-based aggregate queries can be executed on both views with the same granularity, and point-range queries can be executed on the Data Point View.

3.5 Experimental Evaluation

Evaluation Environment

All the proposed methods were implemented as a new distributed TSMS we named ModelarDB. The system uses Apache Spark for query processing and Apache Cassandra for storage. We evaluate the system on a seven node cluster consisting of one master node and six worker nodes. Three real-life data sets are used for the evaluation. Firstly, the data set EH is 582.68 GiB when stored as CSV files and contains high frequency

measurements from wind turbines with a 100 millisecond sampling interval. Secondly, the data set EP is 339 GiB when stored as CSV files and contains measurements from entities in the energy domain such as wind turbines, solar panels, power plants, etc. with a 60 second sampling interval. Finally, the data set ER is 487.52 GiB when stored as CSV files and is an extended version of the REDD data set [49] with each file duplicated and scaled with a random value in the range [0.001, 1.001) 2,500 times.

We compare ModelarDB to popular storage solutions for time series management used in industry: the TSMS InfluxDB, the distributed key-value store Apache Cassandra, and the big data columnar file formats Apache Parquet and Apache ORC. The storage solutions are compared in terms of ingestion rate, compression, scalability, and query performance for multiple query workloads. Queries on ModelarDB are executed on time series ingested with a 10% error bound using the Segment View (SV) and the Data Point View (DPV). All queries on InfluxDB are executed on a single node using its command-line interface (CLI) as the open-source version does not support distribution. For all other storage solutions, an Apache Spark DataFrame (DF) or an Apache Spark Cached DataFrames (DFC) are used for executing the queries across all six worker nodes in the cluster. The number of worker nodes utilized is shown as suffixes to the query interface on the relevant figures.

Ingestion Rate

We evaluate the ingestion rate for all storage solutions on a single worker node for a direct comparison. The experiments are performed using `spark-shell` and the necessary Apache Spark connectors for all formats except InfluxDB, as no mature Apache Spark connector exists for this system. A Python application using Pandas [63] and the client library InfluxDB-Python are used instead [39]. The ingestion rate of ModelarDB is also measured on six worker nodes to evaluate its scalability. Depending on the system, the experiments are performed when Bulk Loading (BL) without any queries being executed or when performing Online Analytics (OA) with small aggregate queries continuously being executed on random time series using the Segment View.

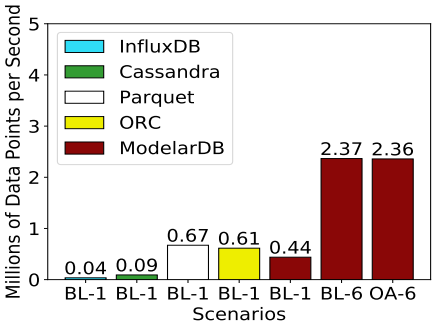

Figure 11: Ingestion rate, ER (Subset) [43]

The results for the ingestion experiment are shown in Figure 11. Only InfluxDB, Apache Cassandra, and ModelarDB support online analytics as an entire Apache Parquet and Apache ORC file must be written before these can be queried. ModelarDB provides 4.89–11 times faster ingestion rates compared to the other systems supporting online analytics. ModelarDB is only 1.52 times slower than Apache Parquet although the latter does not support online analytics. When ingesting the same data set using all six worker nodes, ModelarDB is 5.39 times faster for the bulk loading case and 5.36 times faster for the online analytics case.

Effect of Error Bound

The compression ratio of all storage solutions are compared using all three data sets in Paper B [43], with EH and EP shown in this summary. In addition, the effect of changing the error bound is evaluated for ModelarDB.

The amount of storage required for EH and EP with different error bounds can be seen in Figure 12 and Figure 13. For both data sets, ModelarDB provides better compression than all of the existing storage solutions when using a 0% error bound. Moreover, ModelarDB requires less storage as the error bound is increased. With a 10% error bound, ModelarDB provides at least a 1.8 times increase in compression for EH with an actual average error of only 0.005% and provides at least a 1.6 times increase for EP with an actual average error of only 0.73%. This high level of compression is due to ModelarDB's use of MMC as different combinations of model types are used for each error bound and data set pair. The model type distribution in segments can be seen for EH in Figure 14 and for EP in Figure 15. For EH, ModelarDB primarily uses the constant PMC-MR constant model type as the high sampling interval creates many data points with similar values. However, for EP, ModelarDB primarily uses the lossless Gorilla model type for the 0% error bound, but uses the two lossy models more when the error bound is increased.

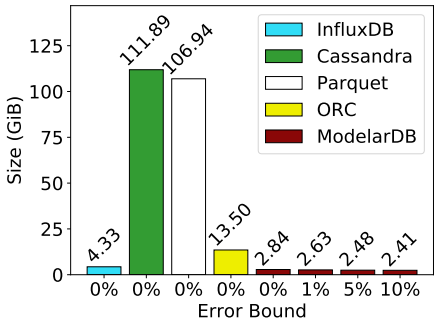

Figure 12: The amount of storage required, EH [43]

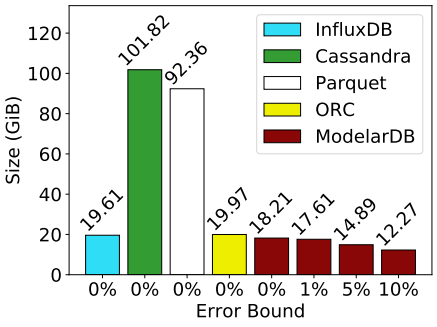

Figure 13: The amount of storage required, EP [43]

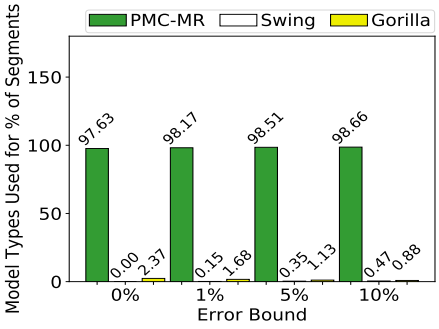

Figure 14: Model types used, EH (Updated) [43]

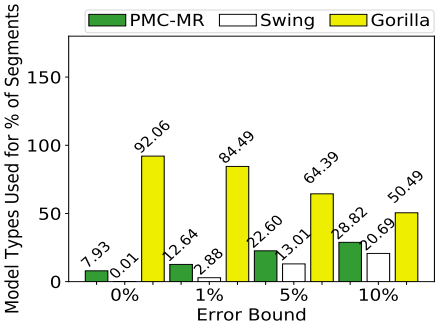

Figure 15: Model types used, EP (Updated) [43]

Scale-out

To evaluate the performance of ModelarDB at scale, we compare the query processing time for all storage solutions when executing large scale aggregate queries on ER using our seven node cluster. For ER, ModelarDB provides up to 27.4 times better compression than the existing storage solutions by combining PMC-MR (82.86%) and to a lesser degree Gorilla (16.09%). To evaluate the ability of ModelarDB to scale-out, we also execute the large scale aggregate queries on Microsoft Azure using between 1 and 32 Standard_D8_v3 nodes. We use this node type as its hardware matches the recommendations provided by the documentation for Apache Spark, Apace Cassandra and Microsoft Azure [4–6].

The results of executing the queries on the local cluster can be seen in Figure 16. Using the Segment View to perform model-based aggregation, ModelarDB is 2.95 times faster than InfluxDB with both only having access to the resources of one worker node. When all six worker nodes are utilized ModelarDB is 1.52 times faster than the closest competitor Apache Parquet. The results for Microsoft Azure can be seen in Figure 17 and demonstrates that ModelarDB scales linearly to at least 32 nodes. This

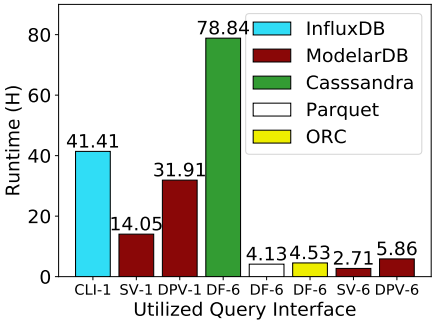

Figure 16: Large aggregate queries, ER [43]

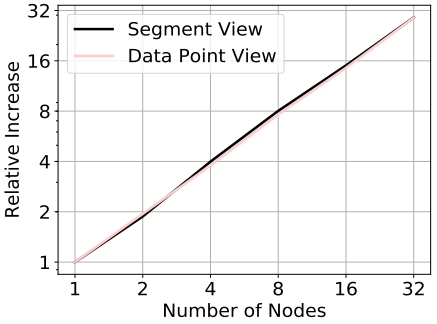

Figure 17: How ModelarDB scales [43]

is expected as each time series is managed by a single node and not distributed across the cluster.

Further Query Processing Performance

To further evaluate the query performance of ModelarDB, we execute a set of small scale aggregate queries and, despite not being ModelarDB's intended use-case, a set of time point-range queries on all storage solutions using our seven node cluster. While we only observed reduced performance when an Apache Spark Cached Data Frame (DFC) were used instead of an Apache Spark Data Frame (DF), Figure 19 shows results with caching both enabled and disabled to demonstrate its effect.

The results for executing small time point-range queries on EH and EP can be seen in Figure 18 and Figure 19 respectively. For these queries, both timestamps and values must be read to extract either specific data point or small sub-sequences. To do this, ModelarDB must reconstruct the data point instead of executing the queries directly on the models. For both EH and EP, InfluxDB, Apache Cassandra, and Apache ORC provides the lowest query processing times. ModelarDB is competitive for EP which consists of many short time series. It is important to note that time point-range queries are not part of ModelarDB's design focus.

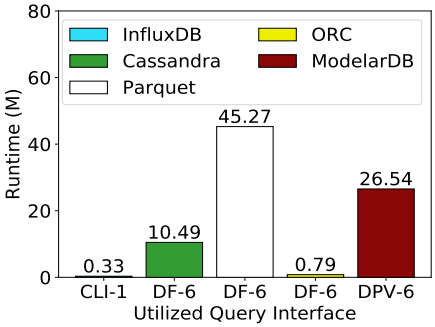

Figure 18: Time point-range queries, EH [43]

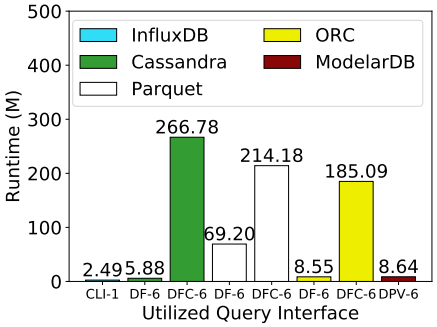

Figure 19: Time point-range queries, EP [43]

The results for executing small scale aggregate queries on EH and EP can be seen in Figure 20 and Figure 21 respectively. For EH, Apache Parquet provides the lowest query processing time. However, despite its columnar format being optimized specifically for simple aggregate queries on a single column, ModelarDB is only 5.96 minutes slower than Apache Parquet and faster than all other competitors. As EP consists of small time series, InfluxDB provides the lowest query processing time. However, ModelarDB is only 1.54 times slower.

Conclusion

The evaluation shows that using the proposed methods ModelarDB provides a combination of properties not present in current storage solutions. The system provides

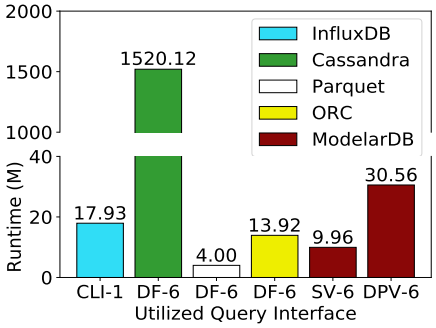

Figure 20: Small aggregate queries, EH [43]

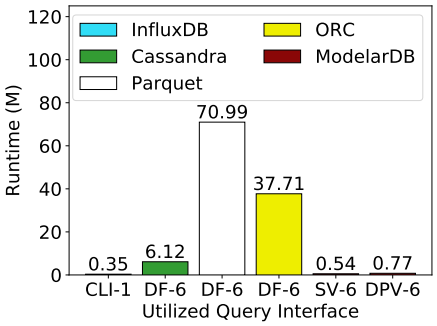

Figure 21: Small aggregate queries, EP [43]

fast ingestion due to its state-of-the-art compression using our proposed MMC method. ModelarDB also scales linearly with the number of nodes added to a cluster. Finally, the system provides state-of-the-art query performance for large scale aggregate queries while also being competitive for small scale aggregate queries and even time point-range queries, despite not being a focus of the design.

4 Management of Correlated Time Series

4.1 Motivation and Problem Statement

As demonstrated in Section 3, utilizing a model-based approach with MMC for time series management can yield state-of-the-art performance. Existing methods for MMC [18, 64, 65, 69], including our own [43], only exploit that data points in a time series are correlated by time, for example that the values of a temperature sensor at t and $t + SI$ will be similar. However, multiple time series in a data set are often correlated. As an example, multiple temperature sensors placed in proximity should generate similar values. Model-based Group Compression (MGC) methods exploit this correlation between time series by compressing multiple time series together in a group. An example of MGC is shown in Figure 22 where three correlated time series are represented by three linear functions when compressed separately, but can be represented as a single linear function if compressed together. However, the existing MGC methods only use one model type for all time series within a group [26, 62]. In other words, it is not possible for them to use a different model type for each sub-sequence of the time series in a group. As such, Paper C [41] addresses multiple problems. Firstly, how can MMC and MGC be combined into a Multi-model Group Compression (MMGC) method with the benefit of both methods? Secondly, how should time series be partitioned across a cluster so that correlated time series are co-located and can be compressed together online without requiring data points to be transferred between nodes and limiting scalability of the MMGC method? Thirdly,

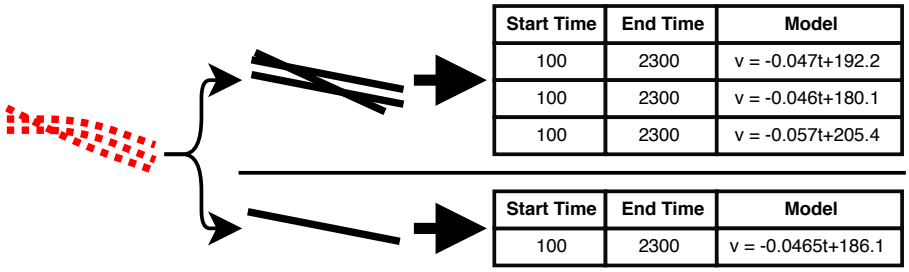

Figure 22: Model-based compression of singular time series (Top) compared to model-based compression of multiple correlated time series using MGC (Bottom) [41]

how can existing model types be extended to represent multiple correlated time series instead of just a single time series? Finally, how can methods for model-based query processing be extended to support executing queries on models representing multiple time series?

4.2 Preliminaries

While ingesting the high quality time series produced in the energy domain individually, identifying each time series using a `Tid` is sufficient. However, these identifiers do not provide any information about the relationship between the time series, such as location or type of entity from which the data points are collected. For this, a robust notation of metadata and the connection between these pieces of metadata is required. For this purpose, we propose using a data cube [27] and extend the definitions from Paper B [43] to accommodate segments representing groups of time series. All definitions are reused from Paper C [41].

Definition 8 (Dimension)
A *dimension* with members M is a 3-tuple $D = (member : \mathbb{TS} \rightarrow M, level : M \rightarrow \{0,1,\ldots,n\}, parent : M \rightarrow M)$ where *(i)* M is hierarchically organized descriptions of the time series in the set of time series $\mathbb{TS}$ with the special value $\top \in M$ as the top element of the hierarchy; *(ii)* *level* is surjective; *(iii)* For $TS \in \mathbb{TS}$, $level(member(TS)) = n$ and $\nexists m \in M$ where $level(m) > n$; *(iv)* For $TS \in \mathbb{TS}$, $m \in M$ and $k \neq \top$, if $level(m) = k$ then $level(parent(member(TS))) = k - 1$; *(v)* $parent(\top) = \top$; *(vi)* $level(\top) = 0$.

A data set of time series is described by a set of members hierarchically organized into dimensions. Each time series has a member at the lowest level of each dimension, meaning that all dimensions have a balanced hierarchy. Each member except the special member $\top$ has a parent. To traverse the hierarchy, the function *member* returns the member for a time series at the lowest level of the hierarchy, the function *parent* returns the parent of each member until the value $\top$ is returned at the top of the hierarchy. Instead of using numbers for the levels of a dimension, we use named

levels to make their connection to real-world entities clear. As an example, given a time series with $Tid = 7337$ collected from one out of four wind turbines located in a park near Vester Hornum in Denmark and a Location dimension defined as $Turbine \rightarrow Park \rightarrow Region \rightarrow Country \rightarrow \top$, this wind turbine would be described with the members $7337 \rightarrow Vester\ Hornum \rightarrow Vesthimmerland \rightarrow Denmark \rightarrow \top$.

Definition 9 (Model)
A *model* of a time series $TS = \langle (t_1, v_1), \ldots \rangle$ is a pair of functions $M = (m, e_m)$. For each t_i, $1 \leq i$, m is a real-valued mapping from t_i to an estimate of the value v_i for the corresponding data point in TS. e_m is a mapping from TS and the corresponding m to a non-negative real value representing the error of the values estimated by m.

Definition 10 (Model Type)
A *model type* is a partial function $m_t(TS, \epsilon)$, which when defined for a bounded time series TS and a non-negative real number ϵ returns a model $M = (m, e_m)$ of TS such that $e_m(TS, m) \leq \epsilon$. We call ϵ the *error bound*.

A definition of a model type is introduced to complement the definition of a model, and make the difference between a model and a model type explicit. As previously stated, we say that a model type fits a model to the data points of a time series when the parameters for a model is computed.

Definition 11 (Time Series Group)
A *time series group* is a set of regular time series, possibly with gaps, $TSG = \{TS_1, \ldots, TS_n\}$, where for $TS_i, TS_j \in TSG$ it holds that they have the same sampling interval SI and that $t_{1i} \bmod SI = t_{1j} \bmod SI$ where t_{1i} and t_{1j} are the first timestamp of TS_i and TS_j, respectively.

A time series group is restricted to only containing time series with the same sampling interval and aligned timestamps. This ensures that a data point is received from all time series in a group at each sampling interval unless gaps occur. If the correlated time series do not consist of approximately the same values, scaling can be applied to allow a single stream of models to represent the values of all time series in the group. By representing the data points from a time series group with only one model per dynamically sized sub-sequence, the compression ratio can be significantly increased compared to a model-based representation of each individual time series.

Definition 12 (Segment)
A *segment* is a 5-tuple $S = (t_s, t_e, SI, G_{ts} : TSG \rightarrow 2^{\{t_s, t_s+SI, \ldots, t_e\}}, M)$ representing the data points for a bounded time interval of a time series group TSG. The 5-tuple consists of start time t_s, end time t_e, sampling interval SI, a function G_{ts} which for the $TS \in TSG$ gives the set of timestamps for which $v = \perp$ in TS, and where the values of all other timestamps are defined by the model M multiplied by a scaling constant $C_{TS} \in \mathbb{R}$.

As the error bound is formally defined as part of a model type, it is no longer included as part of the definition for a segment.

4.3 Static Grouping of Time Series

To minimize the network overhead when compressing correlated time series using MMGC, all time series in a group should be ingested by the same node in a cluster. However, determining how time series should be grouped before ingestion requires that the groups be based either on previously collected data or metadata such as the data set's dimensions. As time series can be long, determining exactly which time series are correlated is a computationally expensive task. For example, even a small data set of only 50,000 time series creates $\binom{50{,}000}{2} \approx 1.25 \times 10^9$ pairs of time series to compare for correlation. As an alternative, we propose a set of primitives that allow domain experts to describe how the time series in a data set are correlated based on their domain knowledge and an analysis of historical data if such exists and the computational resources needed to do are available. As such, finding correlated time series in a data set becomes an orthogonal problem for which existing methods have been proposed [19, 20, 32, 33]. To specify correlation, we propose different types of primitives that can be combined with either `AND` or `OR` semantics. The proposed primitives allow users to specify that time series are correlated based on the source of the time series and a data set's dimension hierarchy. Users can specify that all time series with a specific member, or all time series sharing a sequence of members, should be considered correlated. Users can also specify that time series are correlated if they are within a distance threshold of each other based on the dimension hierarchy.

The first type of primitives allows users to directly specify that specific time series are correlated based on the source of the time series such as a file or a socket, for example `4LR9a_Production.gz 4LR9b_EstimatedProduction.gz`. Defining correlation between time series with respect to the source of the time series is only feasible for data sets of trivial size. For this reason, our remaining primitives specify correlation between time series based on their dimensions. The second type of primitive allows users to specify a specific member that time series must share to be considered correlated. For example, `Location 3 Aalborg` specifies that time series are correlated if they have the member `Aalborg` as the third level of the `Location` dimension. Both the third and fourth type of primitives are based on the Lowest Common Ancestor (LCA) level of a dimension for two time series groups. The LCA level is the number of levels for which all time series in two groups share members in a dimension starting from $\top$. Figure 23 shows an example where the LCA level of two time series with $Tid = 2$ and $Tid = 3$ is the LCA level three, corresponding to the *Park* level, in this Location dimension. For the third type of primitives, users specify a level in a dimension and two time series group are considered correlated if their LCA level is equal to or higher for that dimension. As an example, specifying `Measure 2` means that two time series groups are correlated if all time series in the groups share members at least until level two of the `Measure` dimension. The last primitive simply requires users to specify a distance between 0.0 and 1.0 at which time series should be considered correlated. The distance is based on LCA levels and is computed for two time series groups with one dimension as $(height - level)/height$ where *height*

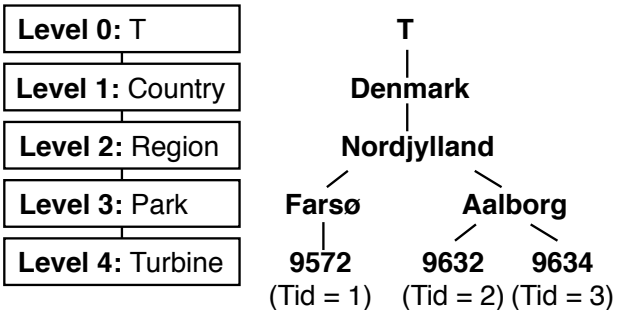

Figure 23: Location dimension were the LCA of the series with $Tid = 2$ and $Tid = 3$ is the level *Park* [41]

is the number of levels in the dimension and *level* is the LCA level. This distance definition ensures that groups of time series that share only a few members at the top of the hierarchy for that dimension are considered less correlated than those that share members at the lowest level. If the data set contain multiple dimensions, the distance is defined as the average distance across all dimensions. A distance of 0.0 requires that all members of two time series groups must match for them to be considered correlated, while a distance of 1.0 means that all time series are correlated. In addition to indicating how time series are correlated, users can also use these primitives to provide scaling factors for specific times series or for time series with specific members. The use of scaling factors allows correlated time series with different values to be compressed together. Based on hints users have provided as correlation clauses using these primitives, groups of correlated time series can be created before ingestion begins using Algorithm 2.

In Line 5, a group is created per time series. Then in Line 6–16 for each correlation *clause* provided by the user, groups are combined if all of the time series in the group are correlated according to the *clause*. In Line 13, the function *correlated* evaluates each *clause* and determines if two groups are correlated. Last, in Line 22 the set of groups created are returned.

4.4 Dynamic Grouping of Time Series

Algorithm 2 creates static groups that contain time series for which their metadata indicates that they are correlated. However, these time series are not guaranteed to always be correlated. As an example, co-located temperature sensors can be correlated, but if one of them is temporarily covered by a shadow while the others are located in the sun, the values they produce will differ. As such, to ensure high compression, a time series has to be split from its group if it temporarily becomes uncorrelated with the remaining time series in the group during ingestion. Should a time series split from a group become correlated with the time series in the group again, the split group must be re-merged so the time series are compressed together again. In addition to dynamic

Algorithm 2 Groups time series based on user-hints provided using the primitives [41]

```
1: Let 𝕋𝕊 be the list of time series.
2: Let 𝔻 be the dimensions for all time series.
3: Let Correlations be a list of correlation clauses.
4:
5: TSG ← createSingleTimeSeriesGroups(𝕋𝕊)
6: for clause ∈ Correlations do
7:     groupsModified ← true
8:     while groupsModified do
9:         groupsModified ← false
10:        for i ← 1 to |TSG| do
11:            for j ← i + 1 to |TSG| do
12:                (TSG_1, TSG_2) ← getPairAt(TSG, i, j)
13:                if correlated(clause, TSG_1, TSG_2, 𝔻) then
14:                    TSG[i] ← mergeGroups(TSG_1, TSG_2)
15:                    removeGroupAfterLoop(TSG_2, TSG)
16:                    groupsModified ← true
17:                end if
18:            end for
19:        end for
20:    end while
21: end for
22: return Groups
```

splitting and joining of time series groups, only a few modifications are required for the ingestion procedure described in Algorithm 1 to support MMGC. At each sampling interval, a group of data points must be buffered and passed to the current model type. In addition, each segment must store information about which time series from the group it represents data from, as segments must be produced for the remaining time series in a group despite a gap occurring in one of the group's time series.

The general idea of dynamically splitting and joining time series groups during ingestion is shown in Figure 24. In ModelarDB, the Segment Generator (SG) contains our method for fitting models to time series groups online. To minimize the overhead of checking when a split should be performed, we propose two heuristics: a compression ratio below a fraction of the average and that the error between ingested data points are above twice the user-defined error bound (2ϵ). If the compression ratio of the model emitted as part of a finalized segment is below a user-configurable fraction of the average, a time series group is divided into sub-group where each sub-group contains correlated time series. As a heuristic for splitting and joining, time series are considered correlated if their currently buffered data points are within twice the user-defined error bound (2ϵ) according to the error function of the included model-types. If only one

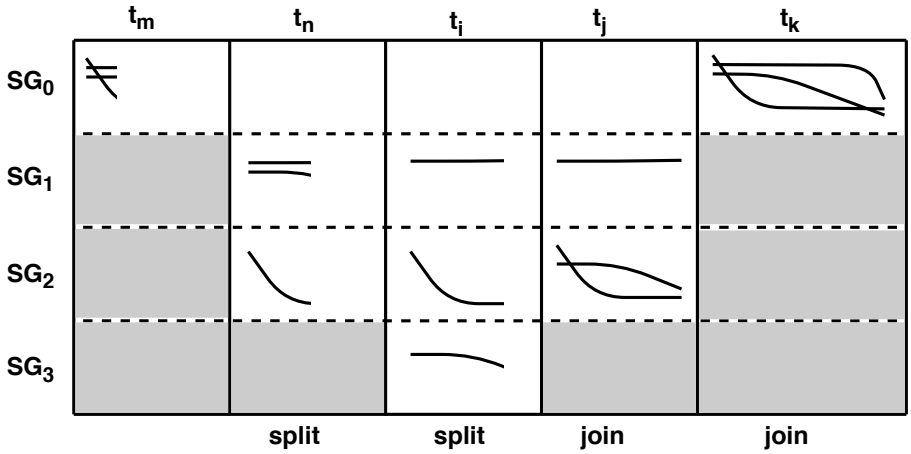

Figure 24: Dynamic splitting and joining of time series groups during ingestion [41]

sub-group is created, all time series in the group are still correlated. If not, each sub-group created is a split. Finalized segments are emitted after all model types have exceeded the user-defined error bound or length limit. For this reason, this check is naturally performed after a finalized segment is emitted and is based on the notion that the construction of a new finalized segment indicates a large change in the structure of at least one of the time series in the group. While ModelarDB deletes the data points that have been emitted as part of a finalized segment, the entire time series is shown in Figure 24 to illustrate how they change over time. At t_m, SG_0 ingests data points from all three time series in the group, but at t_n a time series is no longer correlated with the rest and the group is split and now ingested by SG_1 and SG_2. SG_0 is not deleted as it ensures SG_1 and SG_2 keep a synchronized sampling interval to simplify joining them if the time series in the group become correlated again. Then, at t_i none of the time series are correlated anymore and each are now ingested by a separate generator.

A split time series group are joined if the time series in the group become correlated again. However, to minimize the overhead of checking if the time series are correlated again, it is only performed for each n finalized segments that have been emitted by a split Segment Generator, where n starts at one and is doubled after each check. This is again based on the notion that emitting a finalized segment indicates a large change in the structure of the time series, and that multiple failed checks to join a split time series group indicates that the current splits are stable. The joining of time series split from a group can be performed in any order and at t_j the split Segment Generators SG_3 and SG_2 are joined. Finally, all three time series in the group are joined again at t_k and ingestion is once again performed by SG_0.

4.5 Model Extensions

While the extensions added to our ingestion method to support MMGC are limited, most existing model types do not support representing time series groups and must be extended. One straightforward idea would be to store multiple models per segment, with each model representing the values of a single times series in the group. However, this would only reduce the storage required for the metadata in the segment and not the size of the actual values compared to MMC. In addition, all models stored in a segment must represent the same number of data points as the shared metadata include start time t_s and end time t_e. As such, to gain the full benefit of MMGC, model types that can represent multiple time series using one model must be used. We extended the model types used by ModelarDB: the constant PMC-Mean [52], the linear Swing [21], and the delta compression method proposed for the Gorilla TSMS [66]. The constant model type was changed from PMC-MR [52] to PMC-Mean [52] as it reduces the average error without a large increase in the amount of storage required. The model types extensions can be seen in Figure 25.

Both PMC-Mean [52] and Swing [21] fit models to time series online by maintaining an upper and lower bound for a function that can represent all ingested values, and creates a new segment when a data point is received that is outside the current upper or lower bound. For PMC-Mean [52], no modifications are required to support representing multiple data points for each sampling interval as it simply represents the

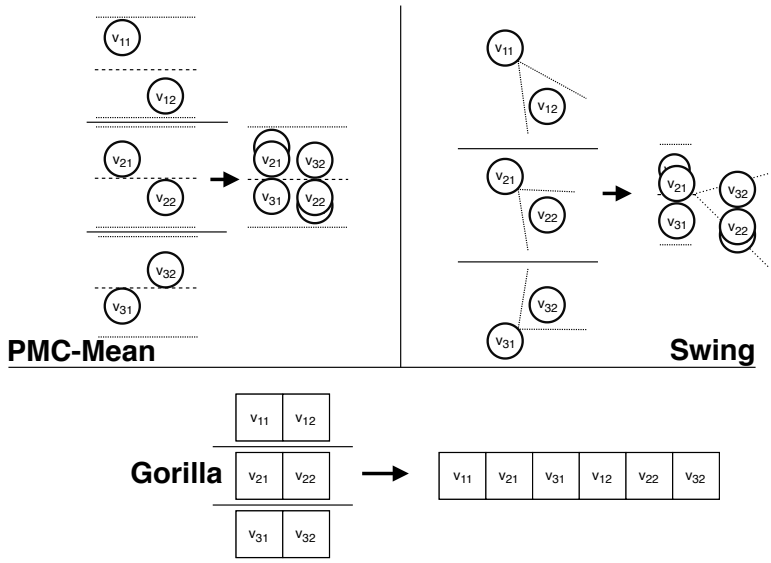

Figure 25: Model types modified to support MGC. For PMC-Mean and Swing a data point's position on the x-axis indicates time while its position on the y-axis indicates its value. Full lines separate the unmodified models, dashed lines are the average maintained by PMC-Mean, and dotted lines are error bounds. [41]

set of all values received V within the error bound ϵ of $\min(V)$ and $\max(V)$ as the single value $avg(V)$. Swing [21] represents each sub-sequence of a time series as a linear function that intersects with an initial data point and then maintains an upper and lower bound as additional data points are received. For the initial data point with timestamp t, we reduce all data points in the group with that timestamp t to a single data point using PMC-Mean. The data points from the following sampling intervals are then added one at a time until Swing cannot represent all data points ingested from a group at a particular sampling interval. For Gorilla [66], the data points are ordered by time so all values received at each sampling interval are stored adjacent. Given that the time series in the group are correlated, only a small delta exists between the values which leads to a high compression ratio.

4.6 Query Processing

The last set of extensions required to support MMGC are for query processing. To allow users to execute queries on specific time series using the Data Point View and the Segment View, the queries must be rewritten so the groups containing segments for the time series requested are retrieved. In addition, the query results should only include values from the requested time series, despite each model representing values for an entire group. For this reason, we propose methods using the Segment View to execute simple model-based aggregates and model-based aggregates in the time dimension. The time dimension is implicitly part of each segment so no separate time dimension is required, as is the case for a traditional data warehouse. Aggregation in the remaining dimensions can be performed as a simple `GROUP BY` query on the denormalized dimensions.

Figure 26 shows our approach for executing simple model-based aggregates on time series that are part of a group. These aggregates use the UDAFs defined on the Segment View to compute aggregates directly on the models. In ModelarDB, these are given the suffix `_S` for example `MIN_S` and `MAX_S`. Firstly, all the time series ids (`Tids`) in the query are rewritten to time series groups ids (`Gids`) so the relevant groups can be located. Segments are then read from disk and the aggregate computed for each time series requested by the user using model-based query processing. If a scaling constant has been applied to any of the time series, and the aggregate performed is not `COUNT`, the result is divided by the scaling constant so each result matches the original time series.

Figure 27 shows our approach for executing model-based aggregates in the time dimension, which exploits that time is a part of each segment. Aggregates in the time dimension can be performed on segments using a set of UDAFs defined on the Segment View. In ModelarDB, they are given the prefix `CUBE` with the aggregation time interval added as a suffix, for example `CUBE_MIN_HOUR` and `CUBE_MAX_HOUR`. The method performs the same operations as described for simple model-based aggregates, but instead of a single result per time series, as in Figure 26, the `StartTime` and `EndTime` of each segment is changed to produce a result per time interval. In

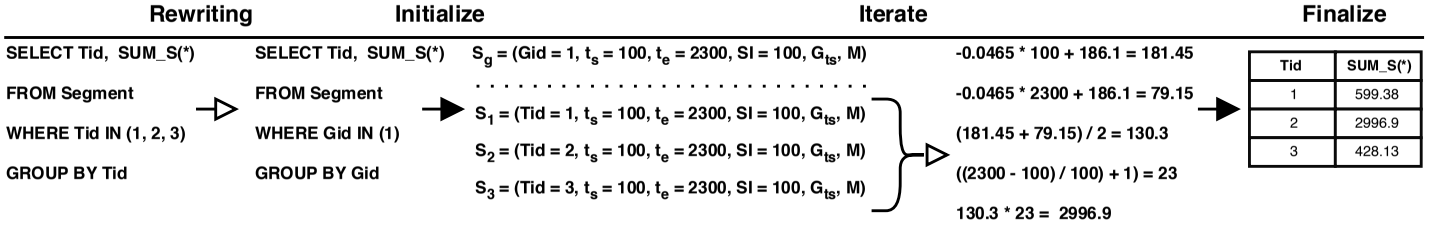

Figure 26: Method for performing simple model-based aggregation for a group of three time series represented by the linear function $-\mathbf{0.0465t} + \mathbf{186.1}$ [41]

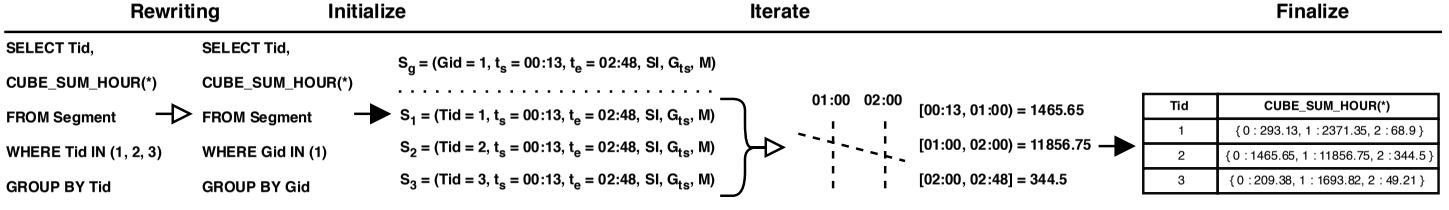

Figure 27: Method for performing model-based aggregation in the time dimension for a group of three time series represented by a linear function [41]

Figure 27, the aggregate is first performed for the time interval $[00:13, 01:00)$, then for $[01:00, 02:00)$ and finally for $[02:00, 02:48]$. `EndTime` is only inclusive for the last time interval as the result would otherwise include duplicate values due to overlapping timestamp.

4.7 Experimental Evaluation

Evaluation Environment

The proposed methods for MMGC and multi-dimensional query processing on models were implemented as extensions to ModelarDB. In addition, the implementation of ModelarDB were improved with multiple smaller optimizations over time to reduce its CPU and memory requirements. As such, we present results from three versions of ModelarDB: our original version ModelarDB$_{v1}$ from [43], ModelarDB$_{v2}-$G which is the current version from Paper C [41] with a more mature implementation but without any correlation specified so MMGC is disabled, and ModelarDB$_{v2}+$G which is the current version from Paper C [41] but each data set is ingested with user-hints specifying correlation using our primitives so MMGC is enabled. Results obtained with MMGC enabled are shown with a striped bar. The evaluation environment are reused from Section 3.5. Both the local cluster and Microsoft Azure are used for the evaluation. The two real-life data sets EP and EH are reused. However, we do not use ER as it is synthetically extended from REDD [49] by creating duplicate time series at different offsets, so the compression achieved using correlation and scaling constants would be artificially high. The queries are reused, and a set of multi-dimensional queries aggregating measurements of energy production for each month is added.

Ingestion Rate

As the data set ER is removed from this evaluation, the ingestion rate is evaluated using a subset of EP containing measurements of energy production. We specify correlation for ModelarDB$_{v2}+$G such that multiple measurements of energy production from the same entity are compressed together. The ingestion rate for all formats are evaluated using `spark-shell` running on a single node. We use the InfluxDB-Java [39] library to connect to InfluxDB instead of InfluxDB-Python, as used in Section 3.5. The Java client library provides a higher ingestion rate, and the use of a Java library allows all experiments to be executed inside `spark-shell`. For all formats, we evaluate the ingestion rate when Bulk Loading (B) without any queries being executed while loading. For ModelarDB$_{v2}$ we also evaluate the ingestion rate while performing Online Analytics (O) with simple aggregate queries continuously executed against the TSMS using the Segment View while loading the data. Finally, the consistency of ModelarDB$_{v2}$'s ingestion rate is evaluated by ingesting a time series continuously duplicated by a separate C program over multiple days.

The results can be seen in Figure 28. ModelarDB$_{v2}+$G outperforms all other storage solutions when bulk loading on a single node by 1.14–12.5 times, due to

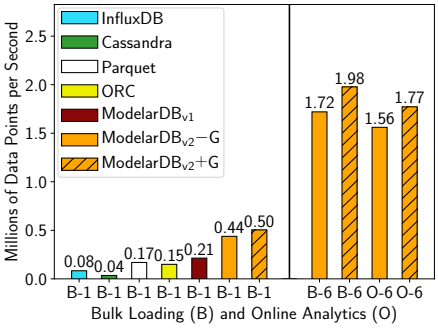

Figure 28: Ingestion rate, EP (Subset) [41]

its use of MMGC and more mature implementation than ModelarDB$_{v1}$. When using all six worker nodes in the cluster, ModelarDB$_{v2}$−G achieves a 3.91 times and a 3.55 times increases when bulk loading and performing online analytics, respectively. ModelarDB$_{v2}$+G achieves a 3.96 times increase when bulk loading and a 3.54 times increase when performing online analytics. When evaluating the consistency of ModelarDB$_{v2}$ ingestion rate, we found that the average time required to ingest the repeated time series for a window of a day only increases by 0.6% from the first to the last day.

Effect of Error Bound

The compression ratio of all storage solutions are compared using EP and EH, and the effect of changing the error bound is evaluated for ModelarDB. To enable MMGC for ModelarDB$_{v2}$+G, correlation between time series are specified for EP so measurements of energy production from the same entity are grouped. For EH, measurements of the same type from each wind turbine in a park are grouped. We expect that enabling MMGC will significantly reduce the amount of storage required for EP as the time series in this data set are highly correlated, while MMGC should only provide a limited benefit for EH as the time series in this data set are much less correlated.

The results for EP can be seen in Figure 29 with all versions of ModelarDB providing better compression than the remaining formats. To our surprise ModelarDB$_{v1}$ and ModelarDB$_{v2}$−G have similar storage requirements despite the more restrictive PMC-Mean constant model type being used for ModelarDB$_{v2}$. By comparing ModelarDB$_{v2}$−G and ModelarDB$_{v2}$+G, we can see that enabling MMGC provides a 1.44–1.56 times reduction in the amount of storage required. This is expected due to the high degree of correlation between the different measurements of energy production in this data set. The results for EH are shown in Figure 30 and the changes in ModelarDB$_{v2}$ provide a small decrease in compression ratio compared to ModelarDB$_{v1}$ as expected. MMGC only provides a benefit with a high error bound as ModelarDB$_{v2}$+G requires 1.06 and 1.28 times less storage then ModelarDB$_{v2}$−G with a 5% and 10% error bound,

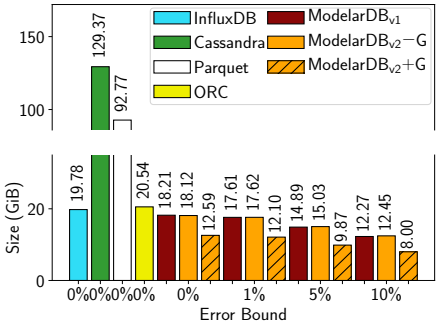

Figure 29: The amount of storage required, EP [41]

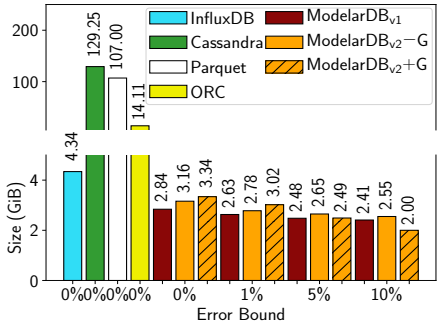

Figure 30: The amount of storage required, EH [41]

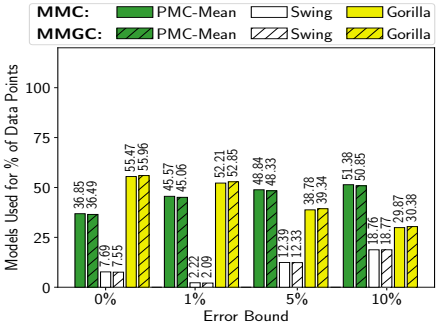

Figure 31: Model types used, EP [41]

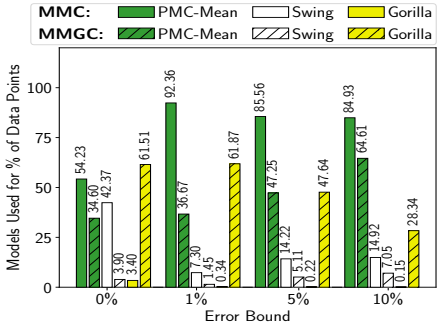

Figure 32: Model types used, EH [41]

respectively. Again, this is expected as the time series in this data set are much less correlated and even small differences between the time series creates new segments when a low error bound is used. The impact of MMGC on the model types used per data point can be seen in Figure 31 for EP and in Figure 32 for EH. For both data sets, enabling MMGC in general increases the number of data points represented by Gorilla and reduces the use of the two lossy models. This is expected as all time series in a group must exhibit the same structure for a sub-sequence to be efficiently compressed using one of the two lossy model types.

Scale-out

ModelarDB$_{v2}$'s ability to perform query processing at scale is evaluated by executing the set of large scale aggregate queries on EP for all storage solutions. The most appropriate query interface is used for each. For InfluxDB, we use its command line interface (CLI). We use an Apache Spark DataFrame (S) for Apache Cassandra, Apache Parquet and Apache ORC, and use the Data Point View (DPV) and the Segment View (SV) for ModelarDB. To evaluate the scalability of ModelarDB$_{v2}$+G, we also execute the queries on EP using between 1 and 32 Standard_D8_v3 nodes on Microsoft Azure.

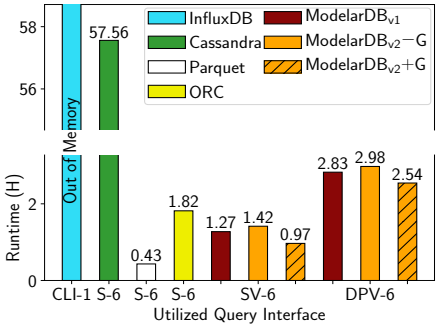

Figure 33: Large aggregate queries, EP [41]

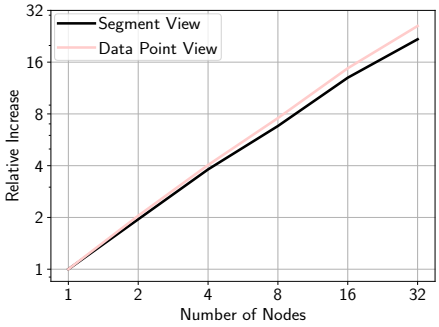

Figure 34: How ModelarDB$_{v2}+$G scales [41]

The results of executing the large scale aggregate queries on our local cluster can be seen in Figure 33. For this data set, InfluxDB is unable to execute the queries as it terminates with out-of-memory errors. Of the remaining formats, only Apache Parquet are faster than the different versions of ModelarDB, and it only provides a 2.26 times reduction in query processing time compared to ModelarDB$_{v2}+$G despite its columnar format being designed specifically for simple aggregate queries on a single column. When comparing the different versions of ModelarDB, ModelarDB$_{v2}+$G achieves a 1.31 and 1.46 times reduction in query processing time compared to ModelarDB$_{v1}$ and ModelarDB$_{v2}-$G, respectively, when using the Segment View. This is due to ModelarDB$_{v2}+$G having to read less data from disk as MMGC is used. The results of executing the large scale aggregate queries on Microsoft Azure can be seen in Figure 34 and shows that ModelarDB$_{v2}+$G provides linear scalability.

Further Query Processing Performance

The remaining sets of queries from Section 3.5 and the new multi-dimensional queries are evaluated on all possible storage solutions. The multi-dimensional queries cannot be executed using InfluxDB as it does neither support aggregation by month [37, 40] nor support any `DATEPART`-like functionality [38]. ModelarDB$_{v1}$ is also not included in our experiment with multi-dimensional queries as it does not support dimensions.

In general, ModelarDB$_{v1}$ and ModelarDB$_{v2}$ provide similar performance for the remaining sets of queries from Section 3.5. For small scale aggregate queries on EP that consists of many short time series, ModelarDB$_{v2}-$G is 17.63 seconds, or 1.62 times, slower than ModelarDB$_{v1}$, indicating that the additional computation required to support MMGC adds a minor overhead. However, for small scale aggregate queries on EH that consists of fewer but much longer time series ModelarDB$_{v2}-$G is 1.46 times faster than ModelarDB$_{v1}$, indicating that the more mature implementation of ModelarDB$_{v2}$ provides improved performance for longer running aggregate queries. The introduction of MMGC have the largest impact on time point-range queries as ModelarDB$_{v2}$ is 1.01–3.92 times slower than ModelarDB$_{v1}$ due to less effective predi-

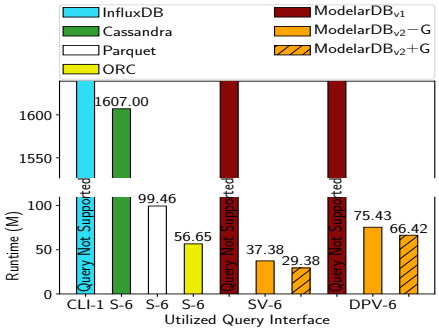

Figure 35: Multi-dimensional queries that `GROUP BY` at the level used for the MMGC groups, EP [41]

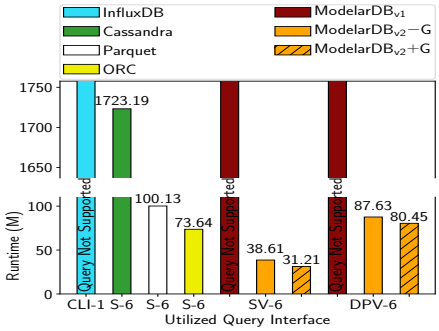

Figure 36: Multi-dimensional queries that `GROUP BY` one level beneath the MMGC groups, EP [41]

cate push-down for these queries and having to read groups. This is considered as an acceptable trade-off as such queries are not the intended use-case for ModelarDB. The results of executing the multi-dimensional queries on EP are shown in Figure 35 and Figure 36. ModelarDB$_{v2}$ provides a 1.52–55.21 times reduction in query processing time using the Segment View, due to the support for executing aggregates in the time dimension on models introduced in ModelarDB$_{v2}$. In addition, ModelarDB$_{v2}+$G is 1.24–1.27 times faster than ModelarDB$_{v2}-$G when using the Segment View due to the additional compression provided by MMGC for EP. As ModelarDB$_{v2}$ can execute queries on the individual time series in each group, the query performance is not reduced when aggregating below the level at which the grouping is performed. This would not be the case for simple aggregates which cannot be decomposed. For EH, ModelarDB$_{v2}-$G is 3.01–253.78 times faster than the other storage solutions, as shown in Figure 37 and Figure 38. However, enabling MMGC for EH increases the query processing time despite providing 1.28 times better compression. This is caused by the different group sizes in EH creating a different load on each node.

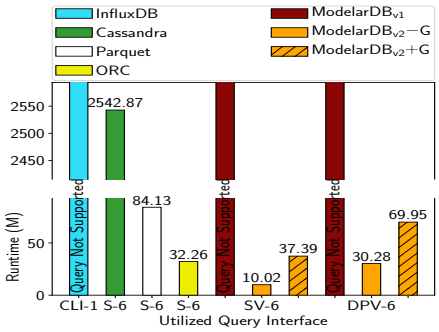

Figure 37: Multi-dimensional queries that `GROUP BY` at the level used for the MMGC groups, EH [41]

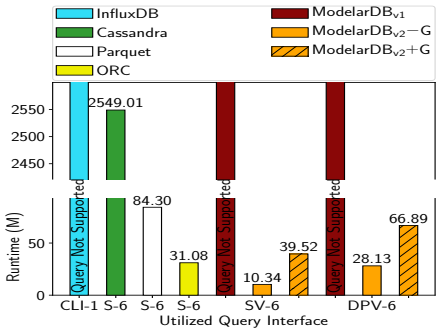

Figure 38: Multi-dimensional queries that `GROUP BY` one level beneath the MMGC groups, EH [41]

Conclusion

The evaluation demonstrates the benefit of ModelarDB$_{v2}$'s use of MMGC with static partitioning and its more mature implementation. The evaluation shows that when compared to ModelarDB$_{v1}$, ModelarDB$_{v2}$ generally provides increased ingestion speed, a reduced storage requirement, and faster query processing, while still scaling linearly on up to 32 nodes. In addition, the evaluation also demonstrates the limitations of MMGC as it only provides limited benefits for data sets containing time series with little correlation and can restrict the ability of ModelarDB$_{v2}$ to evenly distribute the time series across the cluster. However, MMGC can easily be disabled for such data sets, with ModelarDB$_{v2}$ then falling back to MMC for compressing the time series.

5 The ModelarDB System

5.1 Motivation and Problem Statement

ModelarDB is developed as a remedy to the limitations of storage solutions used in industry for time series management, as shown in Table 1, and because none of the TSMSs surveyed in Paper A [42] provides the paramount properties required to efficiently manage high quality time series. We designed ModelarDB to provide the benefits of model-based time series management while allowing users to use it as a TSMS with a simple SQL interface. The current version of ModelarDB uses the methods for model-based management of individual time series proposed in Paper B [43] and was extended with the methods for model-based management of correlated time series presented in the Paper C [41]. Paper D [44] addresses the problem of demonstrating the functionality provided by ModelarDB from a user's perspective. To provide a complete overview of ModelarDB, this section summarizes content from both Paper B [43], Paper C [41] and Paper D [44].

5.2 Architecture

ModelarDB is designed to be both modular and highly configurable to support different deployment environments and different data sets. To be modular, ModelarDB is implemented as the Java library ModelarDB Core, which is interfaced with a query processing system and a data storage system to create the full TSMS ModelarDB. ModelarDB Core contains all the functionality that are agnostic regarding the query processing systems and the data storage systems it currently can be interfaced with. For example, it contains an implementation of our static time series grouping and partitioning algorithm, our online model-based compression algorithm, and the model types provided as part of the system. Users can also implement their own model types without any changes to ModelarDB. The current implementation of ModelarDB interfaces with the stock version of Apache Spark for query processing and either the stock version of Apache Cassandra or a JDBC compatible RDBMS for storage.

However, support for other query processing system such as Apache Flink can be implemented as a new engine class, while support for other storage systems such as Apache HBase, MongoDB, or Apache Kudu can be implemented as a new class deriving from ModelarDB Core's provided storage interface. ModelarDB is purposely implemented without any modifications to its dependencies and only utilize their public interfaces to make it simple to deploy the system and upgrade the dependencies in the future. In addition, as it uses the stock versions of Apache Spark and Apache Cassandra, ModelarDB can be deployed on an existing cluster running Apache Spark and Apache Cassandra by simply deploying a JAR file using Apache Spark's included `spark-submit` script. As such, ModelarDB is simple to deploy and maintain for administrators. Due to the use of Apache Spark, ModelarDB uses a master/worker architecture. The master performs the static grouping of the time series using a *Partitioner* component. It then partitions the groups evenly across the cluster to minimize situations where one worker node is overwhelmed while the remaining are underutilized. ModelarDB uses a partitioning method based on [51] that minimizes $\max_{SG_1 \in SSG}($ $data\_points\_per\_minute(SG_1)) - \min_{SG_2 \in SSG}(data\_points\_per\_minute(SG_2))$ where SSG are the groups created by the Partitioner [41]. After each group has been assigned to a worker node, each worker node ingests the time series directly from their source so inter-cluster network traffic does not limit the scalability of the system.

The architecture of ModelarDB's worker nodes can be seen in Figure 39. For each component, the main provider of this component is written in parentheses. For example, the predefined models are implemented as part of ModelarDB Core while distributed query processing and caching are primarily implemented using functionality from Apache Spark. A worker consists of three sets of components: *Data Ingestion*, *Query Processing*, and *Segment Storage*. The Data Ingestion components receive data points from the time series groups created by the master and assigned to this worker. This set of components also fits models to these time series using a Segment Generator for each group and the model types the user has configured the system to use. The stream of segments produced are cached in memory by the Query Processing components so users can execute queries on them using both the Data Point View and the Segment View. As such, the in-memory cache stores both recently produced and recently queried segments read from the Segment Group Store. As segments from the Segment Generator are evicted from the in-memory cache they are sent to the Segment Storage components and persisted to disk using the Segment Group Store ModelarDB is configured to use. The full source code is available at `https://github.com/skejserjensen/ModelarDB` under the Apache License Version 2.0.

5.3 Model-Based Query Processing

Users provide queries to ModelarDB using its Data Point View and Segment View, with UDAFs and UDFs implemented on the Segment View to support model-based processing of aggregate queries. Queries are expressed using SQL and send to the Query Processing component which is implemented using Apache Spark SQL. The

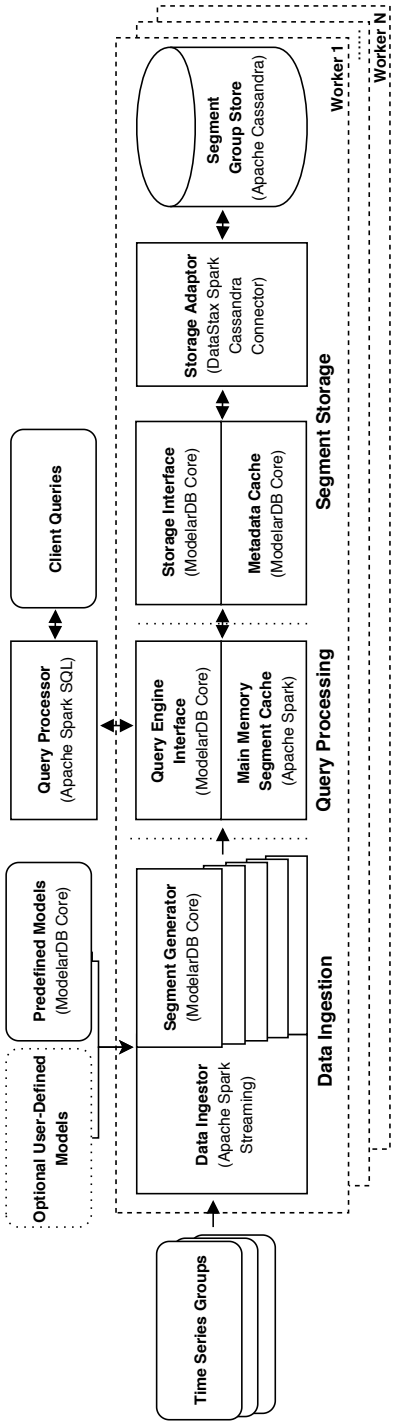

Figure 39: A general architecture for a model-based TSMS and as implemented by ModelarDB with data ingestion, query processing, and segment storage [44]

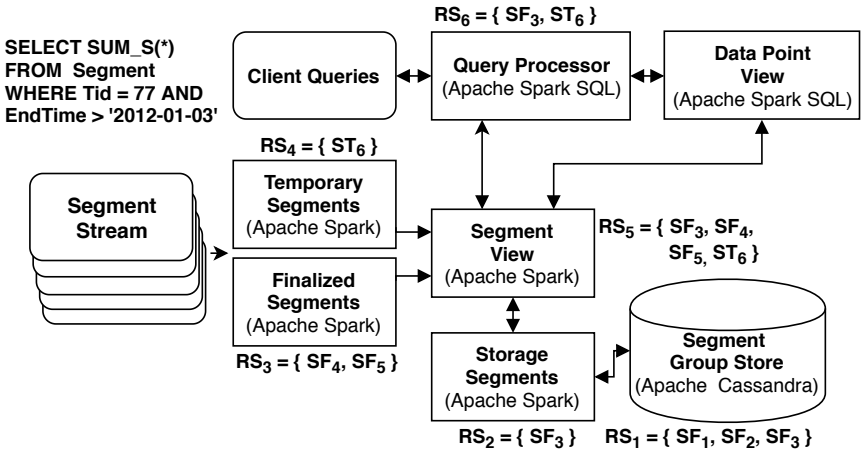

Figure 40: Query processing in ModelarDB using the Data Point View and the Segment View. The acronyms are: result set (RS), temporary segment (SF), and finalized segment (SF). (Updated) [43]

full query processing pipeline of ModelarDB is shown in Figure 40.

In this example, the query uses the Segment View so the query is parsed, the `Tid` rewritten to the corresponding `Gid`, and the requested columns and the predicates of the `WHERE` clause are provided to the Segment View. It is important to be aware that Apache Spark SQL does not provide information about which aggregates, if any, are being performed through its stable APIs. As such, all the requested segments or data points must be provided instead of pre-aggregated results. The Segment View uses the Storage Interface to rewrite the query predicates and push them to the Segment Store, which must return at least the requested finalized segments, shown as RS_2, from the stored finalized segments, shown as RS_1. ModelarDB allows a Segment Group Store to return more than the requested finalized segments to allow use of storage solutions without support for predicate push-down for all types of predicate. The finalized segments retrieved from the Segment Group Store are unioned with the temporary segments and finalized segments currently stored in the cache, shown as RS_3 and RS_4, respectively. The unioned segments representing time series groups, shown as RS_5, are then expanded to segments representing individual time series. The Segment View then filters the expanded segments according to the query's predicates so extra segments are removed before the expanded segments are returned to the distributed query processor, shown as RS_6. The distributed query processor then computes the final results using the `SUM_S` UDAF. Queries executed on the Data Point View use the Segment View read the required segments from the Segment Group Store as described above, before reconstructing approximations of the ingested data points using the model-based representations stored as part of each segment. As such, a new Segment Group Store needs only support predicate push-down from the Segment View, making it simpler to extend ModelarDB with a new Segment Group Store.

In addition to predicate push-down, ModelarDB also supports efficient execution of projections by combining static and dynamic code-generation. Projections are performed by each view before returning a result set. The Segment View and the Data Point View contain a known set of columns. Optimized code for projections that only contain these columns are generated at compile-time. However, as dimensions are defined by users at run-time, optimized code for performing projections involving these columns must be dynamically generated. Alternatively, the rows can be dynamically built based on the requested columns. ModelarDB dynamically builds each row for small data sets, using the number of Spark Partitions as an heuristic that is efficient to compute, and uses dynamic code-generation for larger result sets. ModelarDB performs dynamic code-generation and compilation using the `scala.tools.reflect.ToolBox` APIs provided by the Scala programming language. This approach is used as dynamically compiling Scala code at run-time adds a significant startup cost for each query. This overhead makes it faster to dynamically build the rows for projections on small result sets, with similar approaches also utilized by other researchers to minimize the overhead of dynamic code-generation [48].

5.4 Model-Based Data Storage

To store time series groups as sub-sequences represented by different model types, we propose the general schema shown in Figure 41. The schema consists of three tables; `Time Series`, `Model`, and `Segment`. `Time Series` always stores the time series identifier `Tid`, the `Gid` which is the identifier of the group to which ModelarDB has assigned the time series based on user-hints, the `Scaling` constant given by the user or the default value of one, and the sampling interval of the time series which is stored as `SI`. Any dimensions added by the user are denormalized and stored in the `Time Series` table. The `Model` table stores the identifier of each model type, `Mid`, and the Java `Classpath` to the concrete implementation of this model type. The Java Classpath can include classes that are not part of ModelarDB, as users can

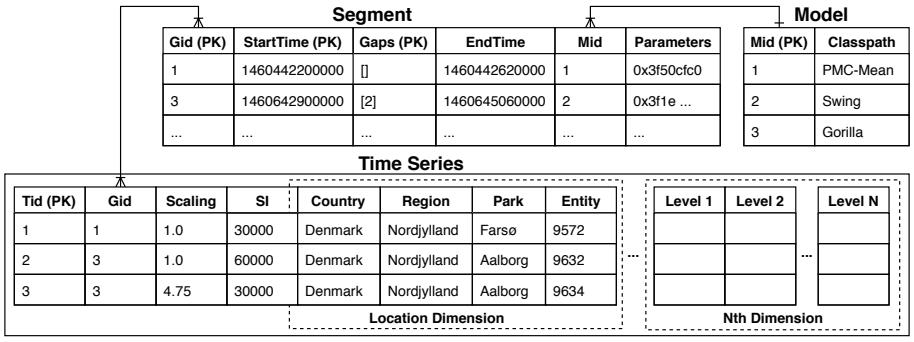

Figure 41: General schema for storing time series groups with dimensions as models [41]

implement their own model types to be loaded dynamically. The `Segment` table stores the segments representing dynamically sized sub-sequences of time series groups using different model types with the required parameters to reconstruct each time series in the group from the model-based representation. As gaps prevent all time series in a group from being part of all segments for that group, the column `Gaps` store the `Tid` of the time series in that group that are not part of the segment. As dynamic splitting and joining of time series groups can create multiple segments with the same `Gid` and `StartTime`, the column `Gaps` are included as part of the primary key.

This general schema is implemented in Apache Cassandra with a few modifications to optimize the schema for compression and due to Apache Cassandra functioning more like a distributed key-value store than a RDBMS. Firstly, `Gaps` in Apache Cassandra are stored as an integer with each bit set if the time series matching that index in the group is not represented by that segment due to a gap. A 64-bit integer is used in the current implementation. This creates an upper limit on the group size of 64 time series when Apache Cassandra is used. Secondly, to more efficiently support predicate push-down, `EndTime` is used instead of `StartTime` for the second part of the key, making the final primary key `Gid, EndTime, Gaps`. This means that the time series are distributed by `Gid` and ordered by `EndTime` followed by `Gaps`. Finally, `StartTime` is replaced with the column `Size` that stores the length of the segment to improve the compression ratio as it requires fewer bits than storing the full timestamp and `StartTime` can be recalculated as $\mathit{StartTime} = \mathit{EndTime} - (\mathit{Size} \times \mathit{SI})$ [43].

5.5 User-Defined Model Types

For domain experts to define a new model type, they must implement two classes deriving from interfaces provided by ModelarDB Core: a `Model` interface and a `Segment` interface. A `Model` must implement functionality to construct a model-based representation from a time series, while a `Segment` must implement functionality to query and reconstruct a time series from a model-based representation. By using two types instead of one, a `Model` can store additional information to more efficiently fit a model to data points without adding any overhead to each of the `Segment` objects created by a `Model`. As only one `Model` object is created for each model type enabled by the user for each Segment Generator, and a `Segment` is created for each sub-sequence represented by that specific model type, the impact of adding additional variables to a `Segment` is much larger than adding additional variables to a `Model`. An overview of the methods a user is required to implement for each type is shown in Table 3.

5.6 Configuration and Static Partitioning

For configuration, ModelarDB uses a commonly used configuration file format. Each setting is given as a name and value pair separated by a space, while each part of the name is separated by a punctuation. For example, the query processing system of ModelarDB can be set using the setting `modelardb.engine`, the Segment Group

Table 3: The interface required for user-defined models and segments, a method marked as ● is a required while a method marked as ◯ is optional (Updated) [43]

Model		
`new(Mid, Error, Limit)`	●	Return a model type with the user-defined error bound and length limit.
`append([Data_Point])`	●	Append a new group of data points if they and all data points the model already represents do not exceed the user-defined error bound.
`initialize([[Data_Point]])`	●	Clear the existing data points from the model and append the groups of data points from the list until one exceeds the error bound or length limit.
`parameters(Start_Time, End_Time, SI, [[Data_Point]])`	●	Get the parameters from the models state and the list of data points.
`get(Tid, Start_Time, End_Time, SI, Parameters, Offsets)`	●	Create a segment represented by the model from serialized parameters.
`length()`	●	Return the number of data points the model currently represents.
`size(Start_Time, End_Time, Resolution, [[Data_Points]])`	●	Return the size in bytes currently required for the models parameters.

Segment		
`get(Timestamp, Index)`	●	Return the value from the underlying model that matches the timestamp and index, both are provided to simplify implementation of this interface.
`min()`	◯	Compute the minimum value of data points represented by the segment.
`max()`	◯	Compute the maximum value of data points represented by the segment.
`sum()`	◯	Compute the sum of the values of data points represented by the segment

Store can be set using `modelardb.storage`, the time series to ingest are set using one or more instances of the setting `modelardb.source` which supports wildcards, the file to read dimensions from are specified using `modelardb.dimensions`, the model types to use are set by one or more instances of the setting `modelardb.model`, and correlations are specified using `modelardb.correlation` with examples shown in Listing 2.

```
1 modelardb.correlation 44L80R9a_AirTemperature 44L80R9b_AirTemperature
2 modelardb.correlation Measure 1 Temperature
3 modelardb.correlation Location 0, Measure 3
4 modelardb.correlation 0.25
5 modelardb.correlation Location 0, Measure 3 * Measure 1 Temperature 0.5
```

Listing 2: User-hints specifying correlated time series [44]

Users can combine our correlation primitives with `AND` semantics by writing multiple primitives separated by a comma as one `modelardb.correlation` setting, or with `OR` semantics by writing multiple different `modelardb.correlation` settings. Two groups of time series are only grouped by ModelarDB if all time series in each group are considered correlated according to a `modelardb.correlation` setting. In Line 1, the user states that the two time series `44L80R9a_AirTemperature` and `44L80R9b_AirTemperature` are correlated and should be grouped together. As this quickly becomes infeasible to manage for even small data sets, Line 2 shows an example of defining correlation based on a specific member. In this case, time series with the member `Temperature` at the first level of the `Measure` dimension are considered correlated and will be grouped. Line 3 shows multiple primitives being combined with `AND` semantics. The line states that time series that share members on all levels in the `Location` dimension and at least the first three members of the `Measure` dimension should be considered correlated. In Line 4, correlation is defined as the distance 0.25, and as such, ModelarDB will compute the distance between each pair of groups and combine the groups that share enough members in each dimension starting from $\top$. Lastly, in Line 5, multiple primitives are combined to not only group time series that share members on all level in the `Location` dimension and at least the first three members of the `Measure` dimension, but a scaling constant of 0.5 will also be applied to all time series with the member `Temperature` at the first level of the `Measure` dimension. This allows correlated time series with different values to be compressed together. The scaling performed by ModelarDB is hidden from the user as they only see the original data points approximated within the user-defined error bound.

5.7 Query Interface

From a user's perspective, ModelarDB query interface operates like a RDBMS as it accepts queries expressed using standard SQL extended with UDAFs, despite using a model-based physical storage layer. The inclusion of the Segment View makes this a leaky abstraction, as without it users querying the system would not know that time

series are stored as models. However, as started in Section 3.4 some query processing systems have very limited public APIs. As such, splitting the query interface into two views makes ModelarDB Core simpler to interface with a new query processing engine and as a result simpler to port to different domains. ModelarDB implements two sets of UDAFs on the Segment View for model-based query processing. The first set allows for model-based aggregates to be computed on segments as rows and each is suffixed by `_S`, for example `MIN_S`, or prefixed by `CUBE` and suffixed with the time interval it aggregates by, for example `CUBE_COUNT_MONTH`. The other set of UDAFs, which are suffixed by `_SS`, are implemented to execute model-based queries on segments as structs, as returned by the UDFs `START`, `END`, and `INTERVAL`. The two sets of UDAFs are needed as Apache Spark SQL requires UDFs to only return one value and does not support overloaded UDAFs. Multiple examples of the queries supported by ModelarDB are shown in Listing 3.

```
1  SELECT AVG(Value) FROM DataPoint WHERE Tid = 3
2  SELECT AVG_S(#) FROM Segment WHERE Tid = 3
3
4  SELECT MIN_SS( START(#, '2019-02-06 18:22') )
5  FROM Segment WHERE EndTime >= '2019-02-06 18:22'
6
7  SELECT Tid, CUBE_MAX_MONTH(#) FROM Segment
8  WHERE Category = 'Temperature' GROUP BY Tid
9
10 SELECT * FROM DataPoint WHERE Tid = 3 AND TS < '2012-04-22 12:25'
```

Listing 3: Examples of queries using the Data Point View and Segment View (Updated) [43, 44]

The query in Line 1 computes a simple `AVG` on one time series using the Data Point View, while the query on Line 2 computes the same average using model-based aggregation through the Segment View. The `#` operator is a specialized version of the `*` operator implemented by ModelarDB that provides the arguments necessary for a UDAF. The query in Line 4–5 computes the minimum value in the data set with a timestamp at or after `2019-02-06 18:22`. As a `WHERE` clause on the Segment View can only filter segments by their start and end time, any values with a timestamp before `2019-02-06 18:22` must be dropped by the UDF `START`. The query on Line 7–8 computes the maximum temperature for each month for each time series using the Segment View. Finally, the query on Line 10 retrieves the data points of the time series with `Tid = 3` with a timestamp before `2012-04-22 12:25`. While the Data Point View provides the illusion that the data points are stored by the TSMS, they are reconstructed from highly compressed models managed by ModelarDB.

6 Summary of Contributions

In conclusion, this thesis performs a comprehensive survey of existing TSMSs and proposes the model-based TSMS ModelarDB designed for regular high quality time series, possibly with gaps, that are correlated and have additional metadata describing them. As part of ModelarDB, the thesis proposes multiple techniques and optimizations

for a model-based TSMS. The four papers included in the thesis provide the following contributions:

- Paper A [42] provides a comprehensive survey of existing TSMSs proposed by academia and through industrial research. A set of classification criteria for TSMSs are proposed based on paramount properties required for a TSMS to efficiently manage high quality time series. The surveyed TSMSs are grouped according to the proposed classification criteria and the user-facing functionality provided by each system. An analysis of each group of systems show that TSMSs integrating both data storage and query processing in one application primarily are research prototypes, that TSMSs with data storage and query processing separated into multiple applications often reuse existing systems for these and are mature system deployed to solve real-world tasks, and that TSMS designed as extensions to RDBMSs are implemented to evaluate model-based query processing for time series. Based on the analyzed TSMSs and future research directions proposed by experts in the field, the survey proposes that a distributed model-based TSMS be developed with support for multi-dimensional analytics.

- Paper B [43] proposes the distributed and modular model-based TSMS ModelarDB, which is designed for individual time series. As part of the TSMS, the paper proposes a general purpose architecture for a model-based TSMS and multiple methods for model-based management of time series. A model-agnostic algorithm for compressing each dynamically sized sub-sequences of a time series with the most appropriate model type is also presented. An API for extending the TSMS with additional model types without changing the system is proposed. Methods for executing queries directly on the model-based representation instead of on reconstructed data points, for performing efficient predicate push-down, and for efficiently executing projections using static code-generation, are also proposed. Finally, a general schema for storing time series as models is presented. Specific optimizations for using the schema with a distributed key-value store are also proposed. An experimental evaluation shows that ModelarDB hits a sweet spot with a fast ingestion rate, state-of-the-art compression, is scalable, and provides fast query processing speeds for large scale aggregate queries, while also being competitive with other storage solutions for small scale aggregate queries and time point-range queries.

- Paper C [41] proposes multiple extensions to ModelarDB for exploiting that time series often are correlated, to increase compression and reduced query processing time. Primitives allowing domain experts to efficiently describe how time series are correlated and a method for creating groups of correlated time series from these user-hints are proposed. Creating the groups using only metadata removes the run-time overhead required to compute which time series are correlated from data points in a distributed setting. This increases the system's scalability. To

support management of time series groups, extensions are proposed for ModelarDB's ingestion method, the included model types, and its model-based query processing method. Methods for model-based query processing are proposed to support multi-dimensional analytics directly on models using metadata stored as a data cube. An evaluation demonstrates that the proposed methods provide faster ingestion rates, an increased compression ratio for correlated time series, and improves the query processing speed for large scale aggregate queries. The system is at the same time both scalable and provides competitive performance for small scale aggregate queries and time point-range queries on data sets with many short time series.

- Paper D [44] describes a demonstration scenario for the current version of ModelarDB TSMS that focus on the functionality it provides from a user's perspective. The paper provides a high-level overview of the system's architecture, its model-based ingestion method, its method for statically grouping correlated time series using our proposed primitives for describing how time series are correlated, and ModelarDB's model-based query processing method and the queries supported by its SQL interface. A graphical user interface is presented which illustrates both the functionality provided by ModelarDB and the underlying implementation.

7 Future Work

Multiple directions for extending ModelarDB exist as future work.

As ModelarDB is designed specifically for high quality time series, it only supports management of regular time series with gaps. As many time series do not follow a regular sampling interval, it would make the system usable in additional domains if it were extended to support time series with irregular and possibly dynamic sampling intervals. However, it is unclear if existing methods for compressing timestamps efficiently can represent timestamps from both regular and irregular time series, without increasing the amount of storage required for regular time series, or if a new compression method optimized for model-based management of time series have to be developed.

While ModelarDB automatically represents each dynamically sized sub-sequence of a time series group with the model type providing the highest compression ratio from a user-configurable set of model types, the system does not provide any information if the selected set of model types are not appropriate for a given data set. As such, the user must manually determine if it would be beneficial to implement a new model type optimized for their domain, and if the ingestion speed can be improved without reducing the compression ratio by disabling model types unsuited for their specific data set. To remedy these problems, ModelarDB should be extended with a model advisor that can determine what set of model types are sufficient to efficiently represent a set of time series. To gain the necessary information, the advisor should analyze historical data from the same sources to determine what structures the time series they produce

contain, or analyze the data compressed by ModelarDB to determine what structures are not efficiently represented by the currently selected set of model types.

ModelarDB is a distributed TSMS designed to ingest data points from remote sources with wired connectivity and electricity. However, as the ingestion rate and the number of sources increases, the network can become a bottleneck. To remedy this problem, the data ingestion components of ModelarDB should be moved to the edge nodes, for example be placed within wind turbines, so only highly compressed models are transferred over the network instead of the raw data points currently being transmitted. In addition, by supporting query processing directly on the edge nodes, both the query latency and the urgency at which the data have to be transmitted from the wind turbine would be reduced.

ModelarDB already utilizes the properties of each model type to efficiently execute aggregate queries directly on the model-based representations instead of on reconstructed data points. However, a model-based index should be built to more efficiently answer queries for specific ranges of values and allow high level analytics, such as similarity search, to be performed directly on the model-based representation.

While ModelarDB only requires two parameters, the sources of the time series to ingest and their error bounds, multiple additional parameters such as cache sizes and the maximum length of segments represented by model types using lossless compression can be configured. To simplify use of ModelarDB, these parameters should automatically be inferred by the system or dynamically tuned at run-time.

References

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data," in *Proceedings of the 8th ACM European Conference on Computer Systems (CIDR)*. ACM, 2013, pp. 29–42.

[2] M. P. Andersen and D. E. Culler, "BTrDB: Optimizing Storage System Design for Time-series Processing," in *Proceedings of the 14th Conference on File and Storage Technologies (FAST)*. USENIX, 2016, pp. 39–52.

[3] M. P. Andersen, S. Kumar, C. Brooks, A. von Meier, and D. E. Culler, "DISTIL: Design and Implementation of a Scalable Synchrophasor Data Processing System," in *Proceedings of the 6th International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2015, pp. 271–277.

[4] "Apache Cassandra - Hardware Choices," http://cassandra.apache.org/doc/latest/operating/hardware.html, Viewed: 2019-08-14.

[5] "Apache Spark - Hardware Provisioning," https://spark.apache.org/docs/2.1.0/hardware-provisioning.html, Viewed: 2019-08-14.

[6] "Azure Databricks," https://azure.microsoft.com/en-us/pricing/details/databricks/, Viewed: 2019-08-14.

[7] A. Bader, O. Kopp, and F. Michael, "Survey and Comparison of Open Source Time Series Databases," in *Datenbanksysteme für Business, Technologie und Web (BTW) - Workshopband*. GI, 2017, pp. 249–268.

[8] G. Bakkalian, C. Koncilia, and R. Wrembel, "On Representing Interval Measures by Means of Functions," in *Proceedings of the 6th International Conference on Model and Data Engineering (MEDI)*. Springer, 2016, pp. 180–193.

[9] B. Bębel, M. Morzy, T. Morzy, Z. Królikowski, and R. Wrembel, "OLAP-Like Analysis of Time Point-Based Sequential Data," in *Advances in Conceptual Modeling - 2012 ER Workshops CMS, ECDM-NoCoDA, MoDIC, MORE-BI, RIGiM, SeCoGIS, WISM*. Springer, 2012, pp. 153–161.

[10] M. Buevich, A. Wright, R. Sargent, and A. Rowe, "Respawn: A Distributed Multi-Resolution Time-Series Datastore," in *Proceedings of the 34th Real-Time Systems Symposium (RTSS)*. IEEE, 2013, pp. 288–297.

[11] S. Cejka, R. Mosshammer, and A. Einfalt, "Java embedded storage for time series and meta data in Smart Grids," in *Proceedings of the 6th International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2015, pp. 434–439.

[12] B. Chardin, J.-M. Lacombe, and J.-M. Petit, "Chronos: a NoSQL system on flash memory for industrial process data," *Distributed and Parallel Databases*, vol. 34, no. 3, pp. 293–319, September 2016.

[13] B. Chardin, O. Pasteur, and J.-M. Petit, "An FTL-agnostic Layer to Improve Random Write on Flash Memory," in *Proceedings of the 16th International Conference on Database Systems for Advanced Applications (DASFAA)*. Springer, 2011, pp. 214–225.

[14] S. Chaudhuri, B. Ding, and S. Kandula, "Approximate Query Processing: No Silver Bullet," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2017, pp. 511–519.

[15] A. Cuzzocrea and D. Saccà, "Exploiting compression and approximation paradigms for effective and efficient online analytical processing over sensor network readings in data grid environments," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 14, pp. 2016–2035, September 2013.

[16] "DB-Engines Ranking," https://db-engines.com/en/ranking, Viewed: 2019-08-14.

[17] L. Deri, S. Mainardi, and F. Fusco, "tsdb: A Compressed Database for Time Series," in *Proceedings of the 4th International Workshop on Traffic Monitoring and Analysis (TMA)*. Springer, 2012, pp. 143–156.

[18] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm, "A time-series compression technique and its application to the smart grid," *The VLDB Journal (VLDBJ)*, vol. 24, no. 2, pp. 193–218, April 2015.

[19] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed Linear Algebra for Large-Scale Machine Learning," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 12, pp. 960–971, August 2016.

[20] ——, "Compressed linear algebra for large-scale machine learning," *The VLDB Journal (VLDBJ)*, vol. 27, no. 5, pp. 719–744, October 2018.

[21] H. Elmeleegy, A. K. Elmagarmid, E. Cecchet, W. G. Aref, and W. Zwaenepoel, "Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 145–156, August 2009.

[22] M. Faschang, S. Cejka, M. Stefan, A. Frischenschlager, A. Einfalt, K. Diwold, F. P. Andrén, T. Strasser, and F. Kupzog, "Provisioning, deployment, and operation of smart grid applications on substation level," *Computer Science - Research and Development*, vol. 32, no. 1–2, pp. 117–130, March 2017.

[23] U. Fischer, D. Kaulakienė, M. E. Khalefa, W. Lehner, T. B. Pedersen, L. Šikšnys, and C. Thomsen, "Real-Time Business Intelligence in the MIRABEL Smart Grid System," in *Revised Selected Papers from the 6th International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*. Springer, 2012, pp. 1–22.

[24] U. Fischer, F. Rosenthal, and W. Lehner, "F^{2}DB: The Flash-Forward Database System," in *Proceedings of the 28th International Conference on Data Engineering (ICDE)*. IEEE, 2012, pp. 1245–1248.

[25] U. Fischer, C. Schildt, C. Hartmann, and W. Lehner, "Forecasting the Data Cube: A Model Configuration Advisor for Multi-Dimensional Data Sets," in *Proceedings of the 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 853–864.

[26] S. Gandhi, S. Nath, S. Suri, and J. Liu, "GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2009, pp. 771–784.

[27] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," in *Proceedings of the 12th International Conference on Data Engineering (ICDE)*. IEEE, 1996, pp. 152–159.

[28] T. Guo, T. G. Papaioannou, and K. Aberer, "Model-View Sensor Data Management in the Cloud," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2013, pp. 282–290.

[29] ——, "Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud," *Big Data Research*, vol. 1, pp. 52–65, August 2014.

[30] T. Guo, T. G. Papaioannou, H. Zhuang, and K. Aberer, "Online Indexing and Distributed Querying Model-View Sensor Data in the Cloud," in *Proceedings of the 19th International Conference on Database Systems for Advanced Applications (DASFAA)*. Springer, 2014, pp. 28–46.

[31] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, "Bolt: Data management for connected homes," in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2014, pp. 243–256.

[32] N. T. T. Ho, T. B. Pedersen, M. Vu, H. L. Van, and C. A. N. Biscio, "Efficient Bottom-Up Discovery of Multi-Scale Time Series Correlations Using Mutual Information," in *Proceedings of the 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1734–1737.

[33] N. T. T. Ho, H. Vo, M. Vu, and T. B. Pedersen, "AMIC: An Adaptive Information Theoretic Method to Identify Multi-Scale Temporal Correlations in Big Time Series Data," *IEEE Transactions on Big Data*, 2019, Early Access.

[34] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale, "WearDrive: Fast and Energy-Efficient Storage for Wearables," in *Proceedings of the Annual Technical Conference (ATC)*. USENIX, 2015, pp. 613–625.

[35] S. Huang, Y. Chen, X. Chen, K. Liu, X. Xu, C. Wang, K. Brown, and I. Halilovic, "The Next Generation Operational Data Historian for IoT Based on Informix," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 169–176.

[36] N. Q. V. Hung, H. Jeung, and K. Aberer, "An Evaluation of Model-Based Approaches to Sensor Data Compression," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 25, no. 11, pp. 2434–2447, November 2013.

[37] "InfluxDB - Issue 3991," https://github.com/influxdata/influxdb/issues/3991, Viewed: 2019-08-14.

[38] "InfluxDB - Issue 6723," https://github.com/influxdata/influxdb/issues/6723, Viewed: 2019-08-14.

[39] "InfluxDB API Client Libraries," https://docs.influxdata.com/influxdb/v1.4/tools/api_client_libraries/, Viewed: 2019-08-14.

[40] "InfluxQL Reference - Durations," https://docs.influxdata.com/influxdb/v1.4/query_language/spec/#durations, Viewed: 2019-08-14.

[41] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB," Unpublished manuscript provided as Paper C.

[42] ——, "Time Series Management Systems: A Survey," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 29, no. 11, pp. 2581–2600, November 2017.

[43] ——, "ModelarDB: Modular Model-based Time Series Management with Spark and Cassandra," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 11, pp. 1688–1701, July 2018.

[44] ——, "Demonstration of ModelarDB: Model-Based Management of Dimensional Time Series," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2019, pp. 1933–1936.

[45] Y. Katsis, Y. Freund, and Y. Papakonstantinou, "Combining Databases and Signal Processing in Plato," in *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR)*, 2015.

[46] M. E. Khalefa, U. Fischer, T. B. Pedersen, and W. Lehner, "Model-based Integration of Past & Future in TimeTravel," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 12, pp. 1974–1977, August 2012.

[47] U. Khurana, S. Parthasarathy, and D. S. Turaga, "FAQ: A Framework for Fast Approximate Query Processing on Temporal Data," in *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications (BigMine)*. JMLR.org, 2014, pp. 29–45.

[48] A. Kohn, V. Leis, and T. Neumann, "Adaptive Execution of Compiled Queries," in *Proceedings of the 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 197–208.

[49] J. Z. Kolter and M. J. Johnson, "REDD: A Public Data Set for Energy Disaggregation Research," in *Proceedings of the Workshop on Data Mining Appliations in Sustainbility (SustKDD)*. ACM, 2011.

[50] C. Koncilia, T. Morzy, R. Wrembel, and J. Eder, "Interval OLAP: Analyzing Interval Data," in *Proceedings of the 16th International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*. Springer, 2014, pp. 233–244.

[51] R. E. Korf, "Multi-Way Number Partitioning," in *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2009, pp. 538–543.

[52] I. Lazaridis and S. Mehrotra, "Capturing Sensor-Generated Time Series with Quality Guarantees," in *Proceedings of the 19th International Conference on Data Engineering (ICDE)*. IEEE, 2003, pp. 429–440.

[53] J. Li, A. Badam, R. Chandra, S. Swanson, B. Worthington, and Q. Zhang, "On the Energy Overhead of Mobile Storage Systems," in *Proceedings of the 12th Conference on File and Storage Technologies (FAST)*. USENIX, 2014, pp. 105–118.

[54] A. MacDonald, "PhilDB: the time series database with built-in change logging," *PeerJ Computer Science*, vol. 2, art. e52, 2016.

[55] A. Marascu, P. Pompey, E. Bouillet, O. Verscheure, M. Wurst, M. Grund, and P. Cudre-Mauroux, "MiSTRAL: An Architecture for Low-Latency Analytics on Massive Time Series," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2013, pp. 15–21.

[56] A. Marascu, P. Pompey, E. Bouillet, M. Wurst, O. Verscheure, M. Grund, and P. Cudre-Mauroux, "TRISTAN: Real-Time Analytics on Massive Time Series Using Sparse Dictionary Compression," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 291–300.

[57] N. D. Mickulicz, R. Martins, P. Narasimhan, and R. Gandhi, "When Good-Enough is Enough: Complex Queries at Fixed Cost," in *Proceedings of the 1st International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 2015, pp. 89–98.

[58] B. Mozafari, J. R. S. Menon, Y. M. S. C. H. Bhanawat, and K. Bachhav, "SnappyData: A Unified Cluster for Streaming, Transactions, and Interactive Analytics," in *Proceedings of the 8th Conference on Innovative Data Systems Research (CIDR)*, 2017.

[59] C. Palmer, P. Lazik, M. Buevich, J. Gao, M. Berges, A. Rowe, R. L. Pereira, and C. Martin, "Demo Abstract: Mortar.io: A Concrete Building Automation System," in *Proceedings of the 1st Conference on Embedded Systems for Energy-Efficient Buildings (BuildSys)*, 2014, pp. 204–205.

[60] T. Palpanas, "Data Series Management: The Road to Big Sequence Analytics," *ACM SIGMOD Record*, vol. 44, no. 2, pp. 47–52, June 2015.

[61] ——, "The Parallel and Distributed Future of Data Series Mining," in *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2017, pp. 916–920.

[62] N. Pan, P. Wang, J. Wu, and W. Wang, "MTSC: An Effective Multiple Time Series Compressing Approach," in *Proceedings of the 29th International Conference on Database and Expert Systems Applications (DEXA)*. Springer, 2018, pp. 267–282.

[63] "Pandas," https://pandas.pydata.org/, Viewed: 2019-08-14.

[64] T. G. Papaioannou, M. Riahi, and K. Aberer, "Towards Online Multi-Model Approximation of Time Series," in *Proceedings of the 12th International Conference on Mobile Data Management (MDM)*, vol. 1. IEEE, 2011, pp. 33–38.

[65] ——, "Towards Online Multi-Model Approximation of Time Series," https://infoscience.epfl.ch/record/164651/files/tech_report.pdf, EPFL LSIR, Tech. Rep., 2011.

[66] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A Fast, Scalable, In-Memory Time Series Database," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1816–1827, August 2015.

[67] K. S. Perera, M. Hahmann, W. Lehner, T. B. Pedersen, and C. Thomsen, "Modeling Large Time Series for Efficient Approximate Query Processing," in *Database Systems for Advanced Applications - DASFAA International Workshops, SeCoP, BDMS, and Posters*. Springer, 2015, pp. 190–204.

[68] J. L. Pérez and D. Carrera, "Performance Characterization of the servIoTicy API: an IoT-as-a-Service Data Management Platform," in *Proceedings of the 1st International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 2015, pp. 62–71.

[69] J. Qi, R. Zhang, K. Ramamohanarao, H. Wang, Z. Wen, and D. Wu, "Indexable online time series segmentation with error bound guarantee," *World Wide Web (WWW)*, vol. 18, no. 2, pp. 359–401, March 2015.

[70] "R-Project," https://www.r-project.org/, Viewed: 2019-08-14.

[71] J. Ronkainen and A. Iivari, "Designing a Data Management Pipeline for Pervasive Sensor Communication Systems," *Procedia Computer Science*, vol. 56, pp. 183–188, 2015.

[72] S. Sathe, T. G. Papaioannou, H. Jeung, and K. Aberer, "A Survey of Model-based Sensor Data Acquisition and Management," in *Managing and Mining Sensor Data*. Springer, 2013, pp. 9–50.

[73] A. L. Serra, T. E. C. i Sebastia, and V. Marta, "A Model for a Multiresolution Time Series Database System," in *Proceedings of the 12th International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED)*. WSEAS, 2013, pp. 55–60.

[74] A. L. Serra, S. Vila-Marta, and T. E. Canal, "Formalism for a multiresolution time series database model," *Information Systems*, vol. 56, pp. 19–35, March 2016.

[75] I. Shafer, R. R. Sambasivan, A. Rowe, and G. R. Ganger, "Specialized Storage for Big Numeric Time Series," in *Proceedings of the 5th conference on Hot Topics in Storage and File Systems (HotStorage)*. USENIX, 2013, pp. 15–15.

[76] "sktime," https://github.com/alan-turing-institute/sktime, Viewed: 2019-08-14.

[77] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The Architecture of SciDB," in *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management (SSDBM)*. Springer, 2011, pp. 1–16.

[78] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, "SciDB: A Database Management System for Applications with Complex Analytics," *Computing in Science & Engineering*, vol. 15, no. 3, pp. 54–62, May-June 2013.

[79] M. Stonebraker and U. Cetintemel, ""One Size Fits All": An Idea Whose Time Has Come and Gone," in *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE, 2005, pp. 2–11.

[80] T. S. D. Team, "Overview of SciDB: Large Scale Array Storage, Processing and Analysis," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2010, pp. 963–968.

[81] Á. Villalba, J. L. Pérez, D. Carrera, C. Pedrinaci, and L. Panziera, "servIoTicy and iServe: a Scalable Platform for Mining the IoT," *Procedia Computer Science*, vol. 52, pp. 1022–1027, 2015.

[82] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh, "Experimental comparison of representation methods and distance measures for time series data," *Data Mining and Knowledge Discovery*, vol. 26, no. 2, pp. 275–309, March 2013.

[83] R. S. Weigel, D. M. Lindholm, A. Wilson, and J. Faden, "TSDS: high-performance merge, subset, and filter software for time series-like data," *Earth Science Informatics*, vol. 3, no. 1-2, pp. 29–40, June 2010.

[84] J. W. Williams, K. S. Aggour, J. Interrante, J. McHugh, and E. Pool, "Bridging High Velocity and High Volume Industrial Big Data Through Distributed In-Memory Storage & Analytics," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 932–941.

[85] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A Real-time Analytical Data Store," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 157–168.

[86] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Indexing for Interactive Exploration of Big Data Series," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 1555–1566.

[87] ——, "RINSE: Interactive Data Series Exploration with ADS+," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1912–1915, August 2015.

[88] ——, "ADS: the adaptive data series index," *The VLDB Journal (VLDBJ)*, vol. 25, no. 6, pp. 843–866, December 2016.

Part II

Papers

Paper A

Time Series Management Systems: A Survey

Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen

The paper has been published in the
IEEE Transactions on Knowledge and Data Engineering (TKDE), Volume 29,
Number 11, Pages 2581–2600, November, 2017. DOI: 10.1109/TKDE.2017.2740932

Abstract

The collection of time series data increases as more monitoring and automation are being deployed. These deployments range in scale from an Internet of things (IoT) device located in a household to enormous distributed Cyber-Physical Systems (CPSs) producing large volumes of data at high velocity. To store and analyze these vast amounts of data, specialized Time Series Management Systems (TSMSs) *have been developed to overcome the limitations of general purpose Database Management Systems (DBMSs) for times series management. In this paper, we present a thorough analysis and classification of TSMSs developed through academic or industrial research and documented through publications. Our classification is organized into categories based on the architectures observed during our analysis. In addition, we provide an overview of each system with a focus on the motivational use case that drove the development of the system, the functionality for storage and querying of time series a system implements, the components the system is composed of, and the capabilities of each system with regard to* Stream Processing *and* Approximate Query Processing (AQP). *Last, we provide a summary of research directions proposed by other researchers in the field and present our vision for a next generation TSMS.*

1 Introduction

The increase in deployment of sensors for monitoring large industrial systems and the ability to analyze the collected data efficiently provide the means for automation and remote management to be utilized at an unprecedented scale [1]. For example, the sensors on a Boeing 787 produce upwards of half a terabyte of data per flight [2]. While the use of sensor networks can range from an individual smart light bulb to hundreds of wind turbines distributed throughout a large area, the readings from any sensor network can be represented as a sequence of values over time, more precisely as a *time series*. Time series are finite or unbounded sequences of data points in increasing order by time. *Data series* generalize the concept of time series by removing the requirement that the ordering is based on time. As time series can be used to represent readings from sensors in general, the development of methods and systems for efficient transfer, storage, and analysis of time series is a necessity to enable the continued increase in the scale of sensor network and their deployment in additional domains [1, 3–5]. For a general introduction to storage and analysis of time series see [6, 7], a more in-depth introduction to sensor data management, data mining, and stream processing is provided by [8–10].

While general DBMSs, and in particular RDBMSs, have been successfully deployed in many situations, they are unsuitable to handle the velocity and volume of the time series produced by the large scale sensor networks deployed today [3–5, 11]. In addition, analysis of the collected time series often requires exporting the data to another application such as R or SPSS, as these provide additional capabilities and a simpler interface for time series analysis compared to an RDBMS, adding complexity to the analysis pipeline [12]. In correspondence with the increasing need for systems that efficiently store and analyze time series, *TSMSs*[1] have been proposed for multiple domains including monitoring of industrial machinery, analysis of time series collected from scientific experiments, embedded storage for Internet of things (IoT) devices, and more. For this paper we define a TSMS as any system developed or extended for storing and querying data in the form of time series. Research into TSMSs is not a recent phenomenon and the problems using RDBMSs for time series have been demonstrated in the past. In the 1990s Seshadri et al. developed the system SEQ and the SQL-like query language SEQUIN [13]. SEQ was built specifically to manage sequential data using a data model [14] and a query optimizer that utilize that the data is stored as a sequence and not a set of tuples [15]. SEQ was implemented as an extension to the object-relational DBMS PREDATOR with the resulting system supporting storage and querying of relational and sequential data together. While additional support for sequences was added to the SQL standard through for examples window queries, development of query languages and TSMSs continued throughout the early 2000s. Lerner et al. proposed the algebra and query language AQuery [16]

[1] TSMS is one among multiple names for these systems commonly used in the literature, another common name is Time Series Database.

for which data is represented as sequences that can be nested to represent structures similar to tables. Utilizing the AQuery data model and information provided as part of the query, such as sort order, novel methods for query optimization were implemented with improved query performance as a result. In contrast to AQuery which uses a non-relational data model, Sadri et al. proposed an extension to the SQL query language named SQL-TS for querying sequences in the form of time series stored in an RDBMS or from an incoming stream [17]. SQL-TS extended SQL with functionality for specifying which column uniquely identifies a sequence and which column the sequence should be ordered by. Patterns to search for in each sequence can then be expressed as predicates in a WHERE clause. They proposed a query optimizer named OPS based on an existing string search algorithm, making complex pattern matching both simple to express and efficient to execute [18, 19].

It is clear that decades of research into management of time series data have lead to the development of expressive query languages and efficient query processing engines. However, this past generation of TSMSs are unable to process the amount of time series data produced today, as support for parallel query processing is limited and the capability to utilize distributed computing is non-existing. In addition, as the TSMSs developed were designed for general-purpose use, only limited optimizations could be implemented for specific use cases [11]. As a result, as new use cases and technologies appear, such as data management for IoT devices and commodity distributed computing, a new generation of TSMSs have been developed. With this paper we provide an overview of the current research state-of-the-art in the area of TSMSs presented as a literature survey with a focus on the contributions of each system. The goal of this survey is to analyze the current state-of-the-art TSMSs, discuss the limitations of these systems, analyze research directions proposed by other researchers in the field, and as a result of our analyses present our vision for a next generation TSMS. To focus the survey, we primarily analyze systems designed for storing numerous time series persistently, allowing the system to support *aggregate analysis* of multiple data streams over time. In addition, while *scalability* has not been used as a criterion for excluding systems, we see it as an important characteristic for any TSMS and a focus of this survey. As a consequence of these decisions, systems such as Antelope [20], HybridStore [21], LittleD [22] and StreamOp [23] developed for deployment on sensor nodes, and more broadly research in the area of query processing inside sensor networks, are not included. For an introduction to the topic of query processing in sensor networks see [8]. Also, the survey only includes systems with papers published in the *current decade*, due to the switch in focus towards Big Data systems built using commodity hardware for large scale data storage and analytics. For TSMSs we observed a general trend towards *distributed storage and processing*, except for TSMSs developed for evaluation of research or for use in embedded systems. Furthermore, only systems implementing methods specifically for storage and querying of time series are included due to the effect design decisions for each component have on the other. The survey also provides an overview of open research questions in order to provide not just an overview of existing systems but also provide insight into the

next generation of TSMSs. Last, in addition to the systems proposed by academia or produced through industrial research, many open-source and proprietary TSMSs have been developed, with the systems OpenTSDB, KairosDB, Druid, InfluxDB, and IBM Informix being popular examples. For information about selecting a TSMS for a particular workload see the following papers [12, 24–29].

The search method for the survey is a follows. An unstructured search using Google Scholar was performed to provide an overview of what research is performed in the area of TSMSs, and to determine the relevant conferences, terms specific to the research area, and relevant researchers. Based on the papers found during the unstructured search, iterations of structured search were performed. Relevant publications found in each iteration were used as input for the next iteration. In each iteration additional papers were located by exhaustively going through the following sources for each paper:

- The references included in the paper.
- Newer publications citing the paper, the citations were found using Google Scholar.
- All conference proceedings or journal issues published in this decade for the conference or journal in which the paper was presented or published. For the data management outlets the most commonly used were ACM SIGMOD, IEEE Big Data and PVLDB, while papers were primarily presented at USENIX conferences when looking at system outlets.
- The publication history for each paper's author found using a combination of DBLP, Google Scholar and profile pages hosted by the author's employer.

The rest of the paper is organized as follows. Section 2 provides a summary of all the systems included in this survey in addition to a description of the criteria each system will be classified by. Section 3, 4, and 5, describe the systems and are organized based on if the data store is internal, external, or if the system extends an RDBMS. Section 6 provides an overview of open research questions proposed by leading researchers in the field in addition to how these ideas correspond to the surveyed systems, culminating in our vision for a next generation system for time series management and what research must be performed to realize it. Last, a summary and conclusion are presented in Section 7.

2 Classification Criteria

An overview of the TSMSs included in this survey is shown with regards to our classification criteria in Table A.1 and the operations each system supports in Table A.2. As some publications refrain from naming the proposed system some systems are marked as *unnamed* with only the references as identification. The TSMSs were organized into

Table A.1: Summary of the Surveyed Systems in Terms of Classification Criteria

	Year	Purpose	Motivationa Use Case	Distributed	Maturity	Scale Shown	Processing Engine	API	Approximation	Stream Processing	Storage Engine	Storage Layout
Internal Stores												
tsdb [30]	2012	Monitoring	Monitoring of large computer networks at high resolution.	Centralized	Mature System	6.3 GB 1 Node	Implemented (C)	Client Library (C)	No support for approximation of time series.	No stream processing.	BerkeleyDB	Fixed sized arrays stored as values in BerkeleyDB.
FAQ [31]	2014	Evaluation	Efficient approximate queries on time series of histograms.	Centralized	Proof-of-Concept Implementation	1.9 GB 1 Node	Implemented (Java)	Unknown	Model-based AQP	No stream processing.	KyotoCabinet	Sketches and histograms organized in a range tree.
WearDrive [32, 33]	2015	IoT	Power efficient storage of data produced by wearable sensors.	Distributed	Demonstration System	Unknown 2 Nodes	Implemented (C, Java, JNI)	Client Library (C, Java, JNI)	No support for approximation of time series.	Register callbacks waiting for sensor readings.	Implemented (C, Java, JNI)	In-memory log of KV-pairs indexed by a hash table.
RINSE [34–36]	2015	Data Analytics	Querying time series without constructing a full index first.	Centralized	Demonstration System	1 TB 1 Node	Implemented (C)	Drawn nearest neighbor search.	Model-based AQP	No stream processing.	Implemented (C)	ADS+ tree indexing an unspecified ASCII format.
Unnamed [37]	2015	Evaluation	Fast approximated queries for decision support systems.	Centralized	Proof-of-Concept Implementation	Unknown 1 Node	Implemented (R)	Extended SQL	Model-based AQP used for aggregate queries.	No stream processing.	Implemented (R)	Separated storage of both raw data and models.
Plato [38]	2015	Data Analytics	Simple analysis of spatiotemporal data with signal processing.	Centralized	Demonstration System	Unknown 1 Node	Implemented (Unknown)	Extended SQL	Model-based AQP	No stream processing.	Implemented (Unknown)	Tables with models stored as a built-in data type.
Chronos [39, 40]	2016	Monitoring	Monitor hydroelectric plants using PCs with flash memory.	Centralized	Demonstration System	11.5 GB 1 Node	Implemented (C++)	Client Library (C++)	No support for approximation of time series.	Out-of-order inserts.	Implemented (C++)	B-Tree index over quasi-sequential data points.
Pytsms [41, 42]	2016	Evaluation	Reference implementation of two formalisms for time series.	Centralized	Proof-of-Concept Implementation	Unknown 1 Node	Implemented (Python)	Client Library (Python)	User-defined aggregates at multiple resolutions.	No stream processing.	Implemented (Python)	Python objects pickled to a file or serialized to CSV.
PhilDB [43]	2016	Data Analytics	Storage and analysis of versioned mutable time series.	Centralized	Demonstration System	119.12 MB 1 Node	Implemented (Python), Pandas	Client Library (Python)	No support for approximation of time series.	No stream processing.	Implemented (Python), SQLite	Triples stored as binary files, updates as HDF5.

Table A.1: (Continued)

	Year	Purpose	Motivationa Use Case	Distributed	Maturity	Scale Shown	Processing Engine	API	Approximation	Stream Processing	Storage Engine	Storage Layout
External Stores												
TSDS [44]	2010	Data Analytics	Querying data in multiple formats from one end-point.	Centralized	Mature System	Unknown 1 Node	Implemented (Java)	REST API serving multiple formats.	AQP using sampling.	No stream processing.	Implemented (Java) for caching.	A binary file with 64-Bit values, metadata as NcML.
SciDB [45–47]	2013	Data Analytics	Storage and querying of data from scientific instruments.	Distributed	Mature System	Unknown 16 Nodes	Implemented (C++), ScaLAPACK	Array languages AFL and AQP.	AQP using sampling.	Arrays piped through external processes.	Implemented (C++), PostgreSQL	N-Dimensional arrays of tuples, stored in chunks.
Respawn [48, 49]	2013	Monitoring	Executing low latency queries across cloud and sensor nodes.	Distributed	Demonstration System	21 GB 10,000 Nodes	Bodytrack Datastore	REST API serving JSON.	Predefined aggregates at multiple resolutions.	No stream processing.	Bodytrack Datastore	Bodytrack Datastore's compressed binary format.
SensorGrid [50]	2013	Data Analytics	OLAP of sensor data through AQP and grid computing.	Distributed	Demonstration System	Unknown 5 Nodes	Implemented (Unknown)	Web Interface, SQL	User-defined aggregates at multiple resolutions.	Window queries.	RDBMS	Data points, aggregates in two-dimensional array.
Unnamed [51–53]	2014	Evaluation	Indexing mathematical models in a distributed KV-store.	Distributed	Proof-of-Concept Implementation	12 GB 9 Nodes	Apache Hadoop	Unknown	Model-based AQP	Real-time modelling of segmented time series.	Apache HBase	In-memory binary trees indexing models in HBase.
Tristan [54, 55]	2014	Data Analytics	Enable creation of analytical applications using sensor data.	Centralized	Demonstration System	Unknown 1 Node	HYRISE	Unknown	Model-based AQP	Real-time modelling of segmented time series.	HYRISE	Sparse representation after dictionary compression.
Druid [56]	2014	Data Analytics	Ingestion and exploration of complex events from log data.	Distributed	Mature System	100 GB 6 Nodes	Implemented (Java)	REST API serving JSON.	User-defined aggregates and model-based AQP.	No stream processing.	Implemented (Java), DFS	Immutable columns encoded based on their type.
Unnamed [57]	2014	Monitoring	Query both time series and relational data through SQL.	Distributed	Mature System	0.90 TB 1 Node	IBM Informix	SQL	Model-based AQP	Real-time modelling of segmented time series.	IBM Informix	Blobs of values, or timestamp deltas and values.
Unnamed [58]	2014	Monitoring	Remote monitoring of industrial installations in real-time.	Distributed	Mature System	5 TB 46 Nodes	GE Streaming Engine, Pivotal GemFire	OQL and Client Library (Java)	No support for approximation of time series.	Real-time data transfor-mations and analytics.	Pivotal Gemfire	Ordered KV-pairs storing segments as linked lists.

Table A.1: (Continued)

	Year	Purpose	Motivationa Use Case	Distributed	Maturity	Scale Shown	Processing Engine	API	Approximation	Stream Processing	Storage Engine	Storage Layout
Bolt [59]	2014	IoT	Simplify management of IoT data from connected homes.	Distributed	Demonstration System	37.89 MB 1 Nodes	Implemented (C#)	Client Library (C#)	AQP using sampling.	Data is written to and read from streams.	Implemented (C#), S3, Azure	In-memory index of continuous chunks on disk.
Storacle [60, 61]	2015	Monitoring	Enable local processing of data at the edge of sensor networks.	Distributed	Demonstration System	541 MB 1 Node	Implemented (Java), Cloud	Unknown	Predefined aggregates.	Online computation of simple aggregates.	Implemented (Java), Cloud	Unknown in-memory format and Protocol Buffers.
Gorilla [62]	2015	Monitoring	Reducing query latency for a real-time monitoring system.	Distributed	Mature System	1.3 TB 20 Nodes	Implemented (C++)	Client Library (Unknown)	No support for approximation of time series.	No Stream processing.	Implemented (C++), DFS, HBase.	Blocks of deltas prefixed by a single timestamp.
Unnamed [63]	2015	Data Analytics	Execute aggregate queries on resource constrained systems.	Distributed	Mature System	3 TB Unknown	Implemented (Java), MySQL	SQL	Model-based AQP used for aggregate queries.	Online computation of aggregate models.	MySQL	Binary tree storing aggregates as models or sums.
servIoTicy [64, 65]	2015	IoT	Integration of stream processing and data storage for IoT.	Distributed	Demonstration System	Unknown 16 Nodes	Couchbase, Apache Storm, Elasticsearch	REST API serving JSON.	No support for approximation of time series.	Apache Storm extended with versioning of bolts.	Couchbase	JSON documents with ids indexed by Elasticsearch.
BTrDB [66, 67]	2016	Monitoring	Analyzing data with ns timestamps at multiple resolutions.	Distributed	Mature System	2.757 TB 2 Nodes	Implemented (Go)	Client Library (Go, Python)	Predefined aggregates at multiple resolutions.	Data is written to and read from streams.	DFS, MongoDB	Versioned tree with aggregates in internal nodes.
RDBMS Extensions												
TimeTravel [68, 69]	2012	Data Analytics	Continues forecasting of power consumption in a smart grid.	Centralized	Demonstration System	Unknown 1 Node	PostgreSQL	Extended SQL	Model-based AQP	No stream processing.	PostgreSQL	Data points in arrays with layers of models on top.
F^2DB [70, 71]	2012	Data Analytics	Forecasting directly integrated as part of a data warehouse.	Centralized	Demonstration System	Unknown 1 Node	PostgreSQL	Extended SQL	Model-based AQP for forecast queries.	No stream processing.	PostgreSQL	Tables and a hierarchy of models built on top.
Unamed [72–74]	2016	Evaluation	OLAP analysis of time series as ordered sequences of events.	Centralized	Proof-of-Concept Implementation	23 MB 1 Node	Oracle RDBMS	Extended SQL	Model-based AQP for interpolated data.	No stream processing.	Oracle RDBMS	Tables for both raw data and model parameters.

three categories based on how the data processing and storage components are connected, due to the major impact this architectural decision has on the implementation of the system. In addition to the differences in architecture, the remaining classification criteria were selected based on what differentiated the surveyed systems from each other. For systems incorporating an existing DBMS only additional functionality described in the surveyed papers will be documented. The full set of classification criteria are:

Architecture: The overall architecture of the implementation is primarily determined by how the data store and data processing components are connected. For some systems both the data store and processing engine are *internal* and integrated together in the same application, either due to them being developed together or if an existing DBMS is embedded and accessed through the provided interface. Other systems use an existing *external* DBMS or DFS as a separate data store, requiring the TSMS to implement methods for transferring time series data to the external DBMS or DFS. Last, some implement methods for processing time series as *extensions to an existing RDBMS*, making the RDBMS's internal implementation accessible for the TSMS in addition to removing the need to transfer data to an external application for analysis. As Table A.1 is split into sections based on architecture, the architecture used by the systems in each section of the table is written in italics as the heading for that section of the table.

Year: The year the latest paper documenting the surveyed system was published. This is included to simplify comparison of systems published close to each other. The year of the latest publication is used as new publications indicate that functionality continues to be added to the system.

Purpose: The type of workload the TSMS was designed and optimized for. The first type of systems are those designed for collecting, analyzing and reacting to data from *IoT* devices, next TSMSs developed for *monitoring* large industrial installations for example data centers and wind turbines, then systems for extracting knowledge from time series data through *data analytics*, and last systems for which no other real-world use case exist than the *evaluation* of research into new formalisms, architectures, and methods for TSMSs.

Motivational Use Case: The intended use case for the system based on the problem that inspired its creation. As multiple systems designed for the same purpose can be designed with different trade-offs, the specific problem a system is designed to solve indicates which trade-offs were necessary for a particular TSMS.

Distributed: Indicates if the system is intended for a *centralized* deployment on a single machine or built to scale through *distributed* computing. It is included due to the effect this decision had on the architecture of the system and the constraints a centralized system adds in terms of scalability.

Maturity: The maturity of the system based on a three level scale: *proof-of-concept implementations* implement only the functionality necessary to evaluate a new technique, *demonstration systems* include functionality necessary for users to interact with the system and are mature enough for the system to be evaluated with a real-life use case, *mature systems* have implementations robust enough to be deployed to solve real-life use cases and supported through either an open-source community or commercial support.

Scale Shown: Scale shown is used as a concrete measure of scalability and defined as the *size of the largest data set* and *number of nodes* a TSMS can manage as documented in the corresponding publications. The sizes of the data sets are reported in bytes after the data has been loaded into the TSMS. As data points differ in size depending on the data types used and the amount of metadata included, data sets for which the size is documented as the number of data points are not included. While some systems are continuously being improved, only the results presented in the referenced publications are included for consistency.

Processing Engine: The data processing engine used by the system for querying, and if supported, analyzing the stored time series data. Included as this component provides the system's external interface for the user to access the functionality provided by the system. If an existing system is used we write the *name of the system*, otherwise if a new engine has been developed we write *implemented* with the implementation language added in parentheses.

API: The primary methods for interacting with the system. The methods can be *query languages*, *extensions of existing query languages*, *client libraries* developed using general purpose programming languages, a *web service*, or a *web interface*.

Approximation: Describes the primary method, if any, for approximating time series that each system supports. Using approximation as part of a TSMS provides multiple benefits. Representing time series as simple aggregates uses less storage and reduces query processing time for queries capable of utilizing the aggregate. AQP expands upon the capabilities of simple aggregates and provides user-defined precision guarantees through the use of mathematical models or sampling. In addition, representing time series as mathematical models, such as a polynomial regression model, provides functionality for data analysis such as interpolation or forecasting depending on the type of model utilized. AQP utilizing sampling reads only a subset of a time series and uses this subset to produce an approximate query result. A survey of mathematical models for representing time series can be found in the second chapter of [8], while additional information about sampling and integration of AQP into a DBMS can be found in [75]. For this survey we differentiate between approximation implemented as simple *aggregates* without guaranteed error bounds, and *AQP* utilizing either *mathematical models* or *sampling* to provide query results with precision guarantees.

Stream Processing: Describes the primary method each system supports, if any, for processing time series in the form of a stream from which data points are received one at a time. As each data point is processed independently of the full time series, each data point can be processed in-memory upon ingestion for applications with low latency requirements [76]. For queries with an unbound memory requirement or high processing time, AQP methods such as sampling or mathematical models can be used to lower the resource requirement as necessary [76]. Stream processing can be implemented using *user-defined functions*, as *functionality part of the TSMS*, or as a *query interface based on streams*. Examples of stream processing include online computation of aggregates, online constructing of mathematical models, removal of outliers in real-time, and realignment of data points in a limited buffer using windows. Operations performed after the data has been ingested by a TSMS, such as maintaining indexes or mathematical models constructed from data on disk, are not included. For a discussion of stream processing in contrast to traditional DBMSs see [76], while an in-depth introduction can be found in [10].

Storage Engine: The data storage component or system embedded into the application, used as external storage, or extended to enable storage of time series. If an existing system is used for data storage, we write the *name of the system*. For TSMSs were the storage component has been written from scratch, the column is marked as *implemented* with the implementation language added in parentheses.

Storage Layout The internal representation used by the system for storing time series, included due to the impact the internal representation has on the systems batch processing, stream processing and AQP capabilities.

In addition to the classification criteria shown in Table A.1, the TSMSs are summarized with regard to their supported query functionality in Table A.2. To describe the functionality of the surveyed TSMSs with a uniform set of well-known terms, we elected to primarily express it using SQL. Two columns for keywords not part of SQL have been added to mark systems for which new data points can only be appended to an existing time series and to show which capabilities for data analytics each system supports. The full list of functionality included is: the capability to *select* data points based on timestamp or value, *insert* data points into a time series at any location, *append* data points to the end of a time series, *update* data points already part of a time series, *delete* data points from a time series, compute *aggregates* from a time series through the use of aggregate functions, *join* multiple time series over timestamp or value, perform computations *over* a time series using either window functions or user-defined functions, and any support for data *analytics* part of the system. In Table A.2 we show a black circle if the TSMS is known to support that type of query through its primary API, and leave the cell empty if the TSMS is known to not support that type of query or if we were unable to determine if the functionality was supported or not. For the column *analytics* we list the supported functionality to differentiate the methods for data analytics supported by each system.

Table A.2: Summary of the Surveyed Systems in Terms of Functionality

	Select	Insert	Append	Update	Delete	Aggregates	Join	Over	Analytics
Internal Stores									
tsdb [30]	●		●	●					
FAQ [31]	●					●			Jaccard, Top-K, AQP, etc
WearDrive [32, 33]	●		●		●			●	
RINSE [34–36]	●	●	●	●	●				NN-Search, AQP
Unnamed [37]						●			AQP
Plato [38]	●	●	●	●	●	●	●		Interpolation, AQP
Chronos [39, 40]	●	●	●	●	●	●			
Pytsms [41, 42]	●		●			●			
PhilDB [43]	●	●	●	●					Pandas
External Stores									
TSDS [44]	●					●			AQP
SciDB [45–47]	●	●	●	●	●	●	●	●	Linear Algebra, AQP
Respawn [48, 49]	●		●			●			
SensorGrid [50]	●		●			●		●	
Unnamed [51–53]	●		●						AQP
Tristan [54, 55]	●		●			●			AQP
Druid [56]	●		●			●			AQP
Unnamed [57]	●	●	●				●		AQP
Unnamed [58]	●		●			●		●	
Bolt [59]	●		●		●				AQP
Storacle [60, 61]	●		●			●			
Gorilla [62]	●		●						
Unnamed [63]	●		●			●			Frequency est., AQP, etc
servIoTicy [64, 65]	●	●	●	●	●			●	
BTrDB [66, 67]	●	●	●	●	●	●			
RDBMS Extensions									
TimeTravel [68, 69]	●	●	●	●	●	●	●	●	Forecasting, AQP
F^2DB [70, 71]	●	●	●	●	●	●	●	●	Forecasting, AQP
Unamed [72–74]	●	●	●	●	●	●	●	●	Interpolation, AQP

In the following three sections, we detail the primary contributions provided by each system in relation to the motivational use case and the classification criteria. Each section documents one set of systems based on the architecture of the system as shown in Table A.1 and Table A.2. In addition to a description of the contributions of each system, illustrations, redrawn from the original publications, are utilized to document the TSMSs. Two types of figures are used depending on the contribution of each system: *architecture diagrams* and *method illustrations*. Architecture diagrams use the following constructs: system components are shown as *boxes with full lines*, *lines* between components show undefined connections, data is shown as *boxes with rounded corners*, *arrows* show data flow between components, logically related components are grouped by *boxes with dotted lines* or by *dotted lines* for architectural layers, data storage is shown as *cylinders*, cluster nodes are components surrounded by *boxes with dashed lines*, and *text labels* are used as descriptive annotations. Method illustrations follow a less rigid structure due to the lack of correlation between the illustrated methods but in general layout of data in memory is shown as *boxes with full lines*. *Squares with rounded corners* are used for nodes in tree data structures. Constructs only used for a single figure are documented as part of the figure.

3 Internal Data Stores

3.1 Overview

Implementing a new TSMS as a single executable allows for deep integration between the data storage and data processing components. As the storage component is not accessible to other applications, the data layout utilized can be extensively optimized for the data processing component. Communication between the two components is also simplified as no communication protocol suitable for transferring time series is needed and no interface constrains access to the data storage layer unless an existing embeddable DBMS is used. The absence of external dependencies reduces the complexity of deploying the TSMS due to the system being self-contained. As a downside a system with an internal data store cannot utilize existing infrastructure such as a distributed DBMS or a DFS that already are deployed. In addition, if a new data storage layer is developed for a TSMS, instead of an existing embeddable DBMS being reused, time will need to be spent learning how that particular data store must be configured for it to perform optimally unless the TSMS provides automatic configuration and maintenance.

3.2 Systems

tsdb presented by **Deri et al.** [30] is a centralized TSMS designed for monitoring the large quantity of requests to the .it DNS registry. For storage tsdb utilizes the embeddable key-value store BerkeleyDB. The use of BerkeleyDB provides a uniform mechanism for storage of both the time series and the metadata used by the database.

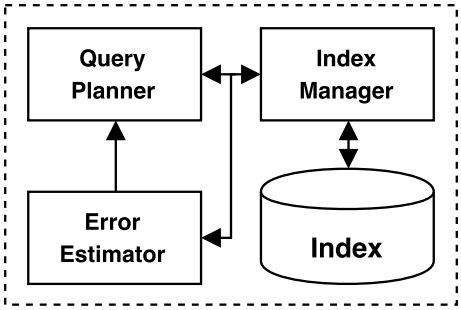

Figure A.1: The architecture of the FAQ system, redrawn from [31]

Data points ingested by tsdb are added to an array and when the array reaches a predefined size it is chunked and stored in BerkeleyDB. Each chunk is stored with a separate key computed by combining the starting timestamp for the array with a chunk id. tsdb requires all time series in the database to cover the same time interval, meaning each time series must have the same start timestamp and contain the same number of data points. This restriction simplifies computations that include multiple time series stored in the system, at the cost of tsdb not being applicable for domains where the need for regular time series adds complexity or is impossible to achieve. In terms of additional capabilities tsdb does not support stream processing, aggregation, or AQP.

The TSMS *FAQ* proposed by **Khurana et al.** [31] utilizes sketches organized in a range tree for efficient execution of approximated queries on time series with histograms as values. The system consists of multiple components as shown in Figure A.1. First, an index containing multiple types of sketches and histograms, each supporting a different type of query or providing a different trade-off between accuracy and query response time. An index manager and an error estimator are utilized by the query planner to select the most appropriate representation based on the provided query and the required error bound. However, utilization of the presented data structure, and thereby the TSMS in general, for use with stream processing of time series is relegated to future work. In addition, the external interface provided by the proof-of-concept implementation is not detailed in the paper.

WearDrive by **Huang et al.** [32] is a distributed in-memory TSMS for IoT, that demonstrates an increase in performance and battery life for a wearable device by transferring sensor data to a smartphone over a wireless connection. The system primarily uses volatile memory for storage as the flash storage used for wearable devices was identified as being a bottleneck in earlier work by Li et al. [33]. By extending the firmware used by a wearable device, WearDrive provides the same persistence guarantees as non-volatile flash storage. The system is split into two applications, as shown in Figure A.2, each built on top of a key-value store implemented for in-memory use. The store is organized as a sequential log of key-value pairs per application with each log file indexed by a hash map. WearCache is running on the wearable device and

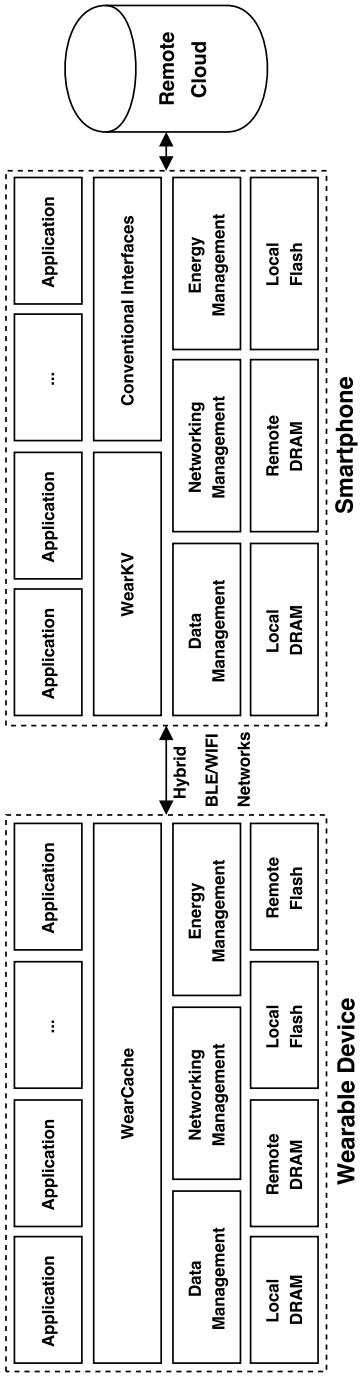

Figure A.2: The split architecture of WearDrive, remote resources are logical and actually located on the other device. The figure was redrawn from [32]

stores only a few sensor readings locally as the data is periodically pushed to remote memory or remote flash which is physically located on the smartphone, causing a drop in battery life for the smartphone. WearKV runs on the smartphone and stores the complete set of sensor readings collected from the wearable devices sensors in addition to the data sent from the smartphone itself to the wearable device. A simple form of stream processing is supported as applications can register to be notified when new values are produced by a specific sensor or for a callback to be executed with all the new sensor values produced over an interval. No support for AQP is provided by WearDrive. While the system is designed for a smaller scale, it follows a structure similar to other TSMSs with the wearable as resource-constrained device collecting data points. The data points are then transferred to the smartphone which serves as the system's backend and provides more capable hardware for storage and aggregate analysis.

RINSE, proposed by **Zoumpatianos et al.** [34], is a centralized system for data analytics supporting execution of efficient nearest neighbor queries on time series by utilizing the ADS+ adaptive index by Zoumpatianos et al. [35, 36]. The implementation is split into two components: a backend and a web frontend. The backend serve as the data storage and indexing component, storing time series in an unspecified ASCII on-disk format indexed using ADS+. The web frontend is served through NodeJS and provide the means to execute nearest neighbor queries by drawing a pattern to search for. In addition, the capabilities of the data storage component are available through a TCP socket. The use of ADS+ for its index provides the system with multiple capabilities. First, as ADS+ is an adaptive tree-based index only internal nodes are initialized while the leafs, containing the data points from the time series, are only initialized if that part of the time series is used in a query. This reduces the time needed to construct the index before queries can be executed compared to alternative indexing methods. Second, approximate queries can be executed directly on the index, providing both AQP and an index for exact queries using the same data structure.

A centralized TSMS that utilizes models for AQP was proposed by **Perera et al.** [37]. The system manages a set of models for each time series and the TSMS optimizer selects the most appropriate model for each query, falling back to the raw data points if necessary. To construct the models the system splits each time series into smaller segments and attempts to approximate each segment with a model. The use of segmentation allows different models optimized for the structure of each segment to be used. If a model with the necessary precision cannot be constructed for a segment then the segment is represented only by the data points, meaning only exact queries can be executed for that segment. The authors proposed an extension of the SQL query language with additional capabilities for AQP: support for specifying which model to use, the maximum error bound for the query, and a maximum query execution time. How a particular query should be executed is left to the query optimizer which given the provided model, maximum query execution time, the required error bound and statistics collected from past queries determines if a model can be used or if the query must be performed using the raw data. However, in the current R based implementation,

AQP using models is only supported for SQL aggregate queries. A later paper by the same authors, Perera et al. [77], reused the method of approximating time series using models to reduce the storage requirements of materialized OLAP cubes.

Another centralized TSMS that implements model-based AQP is *Plato* proposed by **Katsis et al.** [38]. The system combines an RDBMS with methods from signal processing to provide a TSMS with functionality for data analytics, removing the need for exporting the data to other tools such as R or SPSS. Plato consists of three layers, as shown in Figure A.3, in order to accommodate implementation of models, construction of model-based representations of time series, and querying of time series represented as models. At the lowest level, an external interface allows for additional models to be implemented by domain experts. By providing an interface for implementing new models, Plato becomes simple to extend for new domains. The implemented models, however, have to be constructed manually by a database administrator by fitting a model to a specific data set, as an automated model selection through a cost function is left as future work. AQP is supported by querying the models manually through one of two extensions of SQL. ModelQL is designed to appeal to data analysts familiar with R or SPSS, while the InfinityQL language is designed to be familiar to users experienced with SQL. Queries are evaluated directly on a model if the model implements the functionality being used in the query. If the functionality is not implemented, the model is instantiated at the resolution necessary for the particular query. As future work the authors propose that models are not used only as a method for representation of time series, but also returned as the query result in order to provide insight into the structure of the data.

The centralized open-source system *Chronos* by **Chardin et al.** [39] is designed as a TSMS for monitoring industrial systems located in a hydroelectric dam. Due to the location, all persistent storage is flash based to provide increased longevity, with a

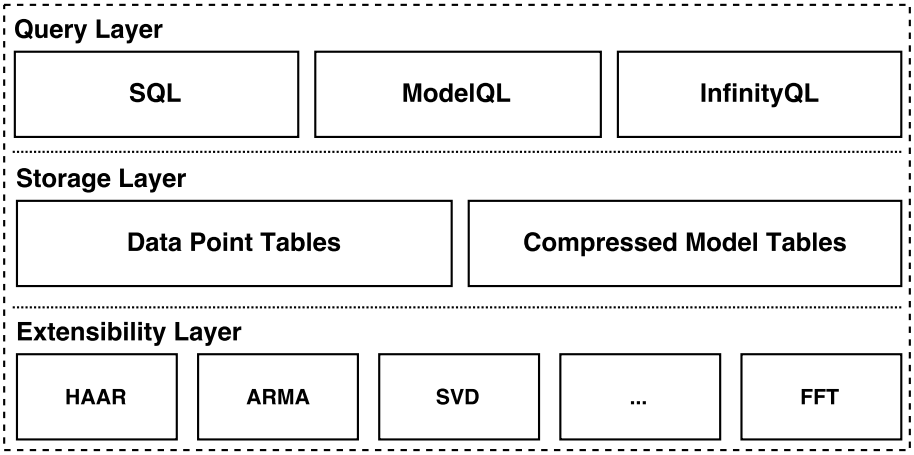

Figure A.3: The architectural layers of the Plato TSMS, redrawn from [38]

substantial drop in write performance compared to spinning disks. To accommodate the use of flash memory, an abstraction layer, proposed by Chardin et al. [40], is implemented on top of the file system so writes are kept close to the previous write on the flash memory ensuring near sequential writes. This is preferable as write duration increases the further from the previous write it is performed. The abstraction layer index data stored on this logical file system through a B-Tree implementing a new splitting algorithm modified to not degrade in performance when used with flash memory. The system provides a simple form of stream processing through efficient support for out-of-order inserts by buffering and reorganizing ingested data before making it persistent, thereby providing a trade off between efficient out-of-order inserts and the possibility of data loss due to hardware failure as the temporary buffers are stored in volatile memory. However, no support for stream processing using user-defined functions, functionality for construction of aggregates or creation of approximate representation of time series are provided.

Pytsms and its extension *RoundRobinson* serve as the reference implementation for two formalisms for TSMSs proposed by **Serra et al.** [41, 42]. However, as the implementation only serves as the proof-of-concept for the formalisms presented in the papers, no attempt was made to make them efficient nor make the implementation comparable in functionality to existing TSMSs. Pytsms implements a centralized TSMS for storage of time series. In addition to storage, Pytsms implements a set of operations that can be performed on time series. RoundRobinson provides a multiresolution abstraction on top of Pytsms. The concept of multiresolution time series is implemented as buffers and round robin discs. When a buffer is filled, either through a time series being bulk loaded or streamed from a sensor into Pytsms, an aggregate is computed by executing an aggregation function with the data points in the buffer as input. The aggregate is added to the disc which functions as a fixed size round robin buffer discarding old aggregates as new are added. As both buffers and discs can be configured, the system is capable of creating any aggregated view of a time series that is needed such as high resolution aggregates for the most recent part of a time series and then decrease the resolution in increments for historical data. An example of such a configuration is shown in Figure A.4. The example shows a schema for a time series represented at different resolutions where only a few aggregates are stored for the entire six hundred days the time series represents, while a high number of data points are stored for the discs containing more recent data.

PhilDB proposed by **MacDonald** [43] is an open-source centralized TSMS designed for data analytics in domains where updates to existing data points are a requirement. To preserve the existing readings and allow for old versions of a time series to be analyzed, PhilDB separates storage of time series into two parts: data points and updates. The data points are stored sequentially in binary files as triples containing a timestamp, a value, and last an integer used as a flag to indicate properties of the data point, for example if the data point is missing a value in an otherwise regular time series. Any updates performed to a time series is stored in an HDF5 file collocated with the binary file storing the data points. Each row in the HDF5 stores the same data

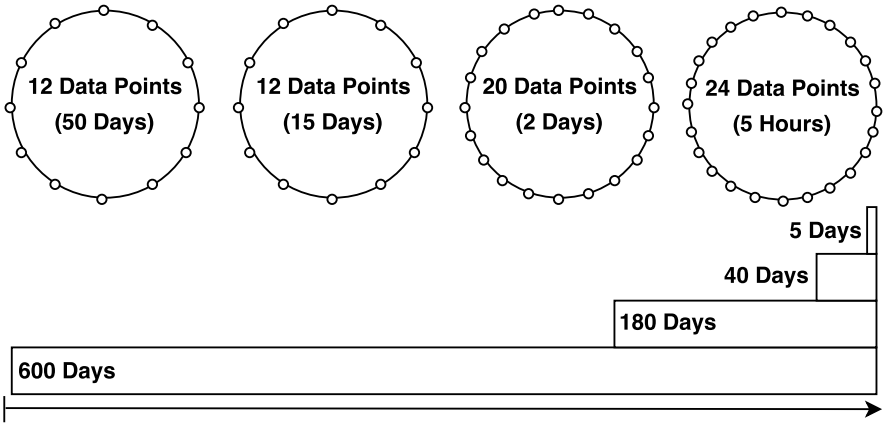

Figure A.4: A time series stored at multiple resolutions using discs of static size and with static granularity for each measurement, redrawn from [41]

as the triples in the binary file with an additional timestamp added to indicate when the update was performed. As the original data points are kept unchanged, queries can request a specific version of the time series based on a timestamp. Additionally, a time series is identified by a user-defined string and can be associated with additional attributes such as the data source for the time series, all stored using SQLite. Reading and writing the time series to disk is implemented as part of PhilDB, however, as its in-memory representation it uses Pandas. The use of Pandas provides a uniform data format for use by the DBMS and allows it to interface directly with the Python data science ecosystem.

3.3 Discussion

A few common themes can be seen among the TSMSs that use an internal data store. First, only WearDrive [32] is distributed and is specialized for use with wearable devices. Instead, researchers implementing distributed TSMSs use existing distributed DBMSs and DFSs running externally from the TSMS as will be described in Section 4. Similarly, only tsdb [30], Plato [38] and PhilDB [43] are intended for use as general TSMSs and have implementations complete enough for this task based on the description in their respective publications. The remaining systems serve only as realizations of new theoretical methods, FAQ [31] is used for evaluating a method for model-based AQP, RINSE [34] demonstrates the capabilities of the ADS+ index [35, 36], Chronos [39] illustrates the benefits of write patterns optimized for flash storage but does neither provide thread-safety nor protection against data corruption, the system by Perera et al. [37] is incomplete, and both Pytsms and RoundRobinson [41, 42] are simple implementations of a formalism. In summary, researchers focusing on development of general purpose TSMSs are at the moment focused on systems that

solve the problem at scale through distributed computing, with centralized systems being relegated to being test platforms for new theoretical methods. A similar situation can be seen in terms of AQP and stream processing as none of the three general purpose TSMS, tsdb [30], Plato [38] and PhilDB [43], support stream processing, and only Plato implements model-based AQP. In addition, Plato provides an interface for users to implement additional models, making it possible for domain experts to use models optimized for a specific domain. For the research systems, FAQ [31], RINSE [34] and the system Perera by et al. [37] all support AQP using models, while Pytsms and RoundRobinson [41, 42] only provide user-defined aggregates. None of the four research systems implemented functionality for stream processing.

4 External Data Stores

4.1 Overview

Implementing a new TSMS by developing a data processing component on top of an existing DBMS or DFS provides multiple benefits. Existing DBMS and DFS deployments can be reused and knowledge about how to configure the system for a particular workload can still be applied. The development time for the system is also reduced as only part of the TSMS must be implemented. In terms of disadvantages, the choice of DBMS will restrict how data is stored and force the processing component to respect that data model. Second, deployment of the system becomes more complicated as multiple separate systems must be deployed, understood, and configured for a particular workload.

4.2 Systems

TSDS developed by **Weigel et al.** [44] is an open-source centralized system designed for data analytics. TSDS supports ingesting multiple time series stored in different formats, caching them and then serving them from a central location by providing a HTTP interface for querying the time series. As part of TSDS a TSMS named TSDB is developed. TSDB is used primarily for caching and storing each separate time series in a sequential binary format on disk to reduce query response time. Multiple adapters have been implemented to allow for ingesting times series from sources such as ASCII files and RDBMSs. Transformations through filters and sampling can be performed by TSDS so only the time interval and resolution requested are retrieved from the system without depending on an external data analytics program such as R or SPSS. As the system is designed for ingesting static data sets no methods for stream processing is provided. Development of TSDS is no longer in progress, however, the TSDS2 project is developing a new version of the TSMS.

Proposed by the **SciDB Development Team** [47] and **Stonebraker et al.** [45, 46], *SciDB* is a distributed DBMS designed to simplify large scale data analytics for researchers. While not designed exclusively for use as a TSMS, SciDB includes

functionality for storing sequential data from sensors used to monitor experiments. SciDB stores data as N-dimensional arrays instead of tables like in an RDBMS, thereby defining an implicit ordering of the data stored. The arrays are versioned and updates to existing arrays produce a new version. A SciDB cluster uses an instance of PostgreSQL for storing the system catalog containing configuration parameters and array metadata, including for example the array version number. Two query languages are implemented: AQL a language similar to SQL that provides a high level declarative interface and AFL which is a lower level language inspired by APL, which allows a chain of operators to be defined directly. Matrix operations are performed outside SciDB using the ScaLAPACK linear algebra library executing alongside the DBMS. In addition, multiple client libraries have been developed, such as SciDB-R that supports R programs executed in SciDB. Support for AQP is provided through a function for sampling a subset of data from arrays, and stream processing is limited to an API inspired by Hadoop Streaming for streaming arrays through external processes akin to Unix pipes. Other DBMSs have been implemented using the open-source SciDB code base. As an example Li et al. [78] created FASTDB, a DBMS specialized for use in astronomy. FASTDB extends SciDB in several areas: an interactive frontend for data exploration, additional monitoring for cluster management, a parallel data loader, a method for dynamically determining the appropriate size of chunks when splitting the data into chunks for storage, and last enhancements of the query optimizer in SciDB was implemented. For additional information about DBMSs based on arrays see the survey by Rusu et al. [79].

The distributed TSMS *Respawn* presented by **Buevich et al.** [48] is designed for monitoring using large scale distributed sensor networks and provide low latency range queries on the produced data points. In Respawn, data storage is performed by the multi-resolution TSMS Bodytrack DataStore which is integrated into two different types of nodes: sensor network edge nodes and cloud nodes as shown in Figure A.5. Edge nodes are ARM based devices that are placed at the edge of a sensor network and ingest the data points produced by the sensor network and compute aggregates at multiple different resolutions, enabling reduced query response time for queries at resolutions lower than what the sensor is being sampled at. In addition to the distributed edge nodes,

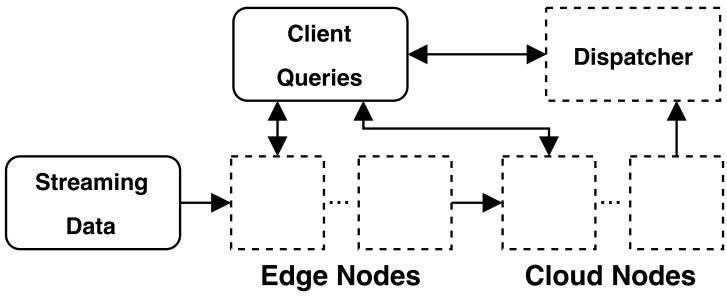

Figure A.5: Architecture of the Respawn TSMS with edge nodes collecting data at the source and migrating it to cloud nodes, redrawn from [48]

server grade cloud nodes are deployed to provide a cache of the data stored in the edge nodes, with segments of time series being migrated using two preemptive strategies. Periodic migration forces migration of low resolution aggregates at specific intervals, as the high resolution data usually is used only if an unusual reading is found through analysis of the low resolution time series. Proactive migration defines how segments of a time series should be migrated based on the standard deviation of the data points, as this can indicate irregular readings worth analyzing further. Queries are performed through HTTP requests and to determine if they should be sent to an edge node or to a cloud node a dispatcher is implemented to track what data has been migrated. After the location of the time series at the requested resolution has been provided, all communication is done directly between the client and the edge node or cloud node, without routing it through the dispatcher. Respawn was utilized as part of the Mortar.io platform for building automation presented by Palmer et al. [49]. Mortar.io provides a uniform abstraction on top of the automation infrastructure, such as sensors, providing a web and mobile interface in addition to simplifying the development of applications communicating with the distributed hardware components.

SensorGrid is a grid framework for storage and analysis of sensor data through data analytics proposed by **Cuzzocrea et al.** [50]. The proposed architecture consists of sensors, a set of stream sources that represent nodes for interacting with sensors and transition of the collected data points to stream servers that store the data. To reduce query response time for aggregate queries, the stream servers pre-compute a set of aggregates defined by the system with no mechanism provided for defining a user-defined aggregation function. The aggregates are computed using an application-specific time interval and the sensor that produced the data points, with the requirement that an aggregation hierarchy for sensors must be explicitly defined. Distributed processing is implemented as part of the stream servers providing them with three options when a query requests data not present at a node: redirect the query, use a local approximation of the data requested by the query at the cost of accuracy, or decompose the query into multiple sub queries and send them to other nodes in the grid. In addition to approximation, the system supports execution of queries over fixed windows, and even continuous queries over moving windows using SQL for stream processing. The SensorGrid architecture was realized as a distributed TSMS and used for hydrogeology risk analysis developed at IRPI-CNR, where the stream source and stream server nodes are integrated with the RDBMS Microsoft SQL Server 2000 and a web interface for visualizing the data.

Guo et al. [51–53] proposed a TSMS that uses mathematical models stored in a distributed key-value store to reduce query response time. The system consists of three parts, a key-value store for storing segments of the time series represented by models, two in-memory binary trees used as an index for the modelled segments, and last an AQP method based on MapReduce. To enable the index to support both point and range queries, the binary trees are ordered by intervals, with one tree constructed for time intervals and another for value intervals. Similarly, two tables are created in the key-value store as one stores the modelled segments with the start time and end

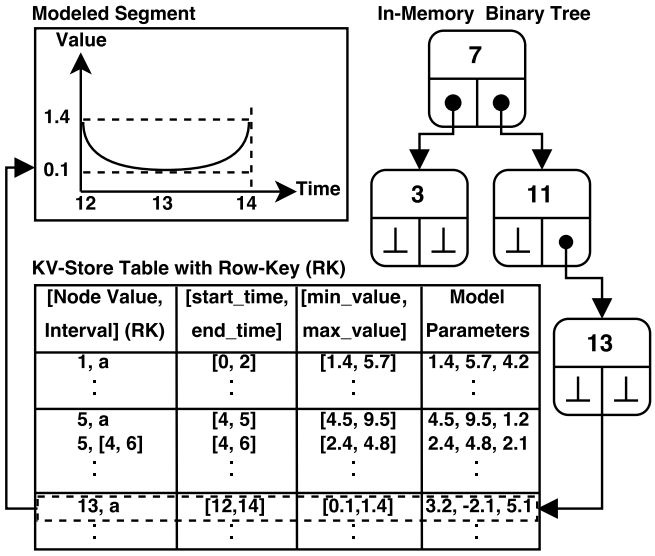

Figure A.6: Model-based AQP combining an in-memory tree and a distributed KV-store for the TSMS by Guo et al. [51–53], redrawn from [51]

time as part of each segment's key, while the other table stores each model with the minimum and maximum value that is part of the modelled segment as part of its key. Query processing is shown in Figure A.6 and is performed by first traversing the index to lookup segments relevant to the query by determining which nodes overlap with the time and values specified in the query. Second, MapReduce is used to extract the relevant segments from the key-value store and re-grid them. As each node in the tree might reference multiple segments, the mapping phase prunes the sequence of proposed segments to remove those that do not fit the provided query, and last the reducer phase re-grids the segments to provide values approximating the original data points. The described method does only rely on the current data point and a user-specified error bound allowing it to be implemented in terms of stream processing.

Tristan is a TSMS designed by **Marascu et al.** [54] for efficient analytics of time series through the use of AQP and online dictionary compression. The system is based on the MiSTRAL architecture also proposed by Marascu et al. [55]. The system's architecture is shown in Figure A.7 and is split into three different layers: a data acquisition and compression layer, a storage layer, and last a query execution layer. The data acquisition and compression layer ingests data into segments which are then stored in a temporary data store managed by the storage layer. When a segment reaches a pre-configured size, the segment is compressed as a sequence of smaller fixed size time series using a dictionary provided as a parameter to the system. The accuracy of the compressed time series can be configured based on the number of fixed sized time series

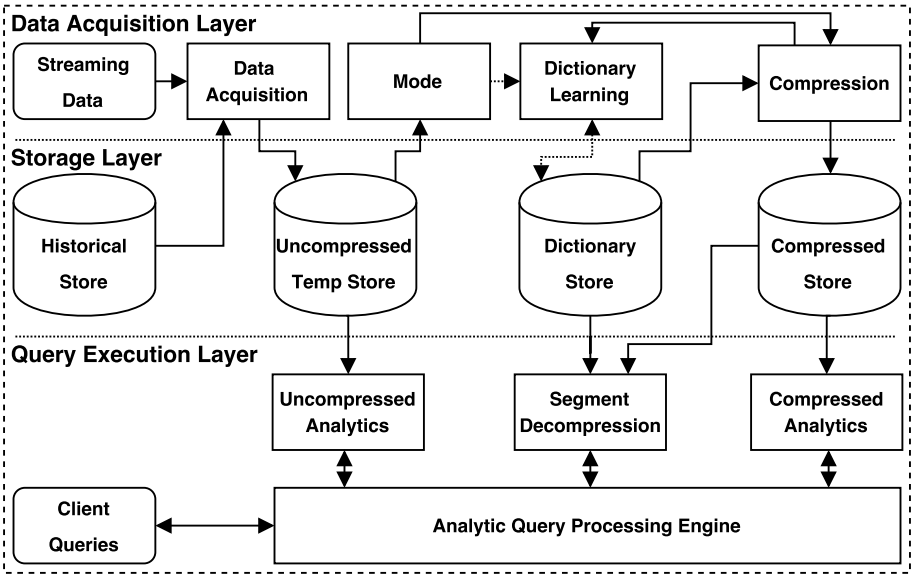

Figure A.7: The MiSTRAL architecture realized by the TSMS Tristan with data flow occurring only in offline mode as dotted arrows, redrawn from [55]

it is represented as. Creation of the dictionary is performed through offline training, however, when operational, the system will continue to adjust the dictionary based on predefined time intervals. The storage layer manages access to the uncompressed time series, the temporary store, the dictionaries used for compression and the compressed segments. The current implementation of the storage layer is based on a customized version of the open-source in-memory DBMS HYRISE. Last, the query execution layer receives queries from the user in an unspecified manner and format. Tristan is capable of using three different methods for answering the query: if the query is for data that is in the process of being compressed, the query is executed against the temporary data store, if the query references a compressed segment Tristan determines if the query can be answered directly from the compressed representation and only decompresses the time series segment if answering the query from the compressed form is not possible. The system supports use of AQP to reduce query response time due to its functionality for selecting the precision for compressing segments by adjusting how many smaller fixed size time series it is represented by. Tristan only supports stream processing through its capability to transform incoming data into an approximate model by buffering it in an uncompressed form in a temporary data store. No support for transforming the data through user-defined functions are possible.

Yang et al. implemented the open-source system *Druid* [56] as a distributed TSMS for efficiently ingesting time series in the form of events from log files, and then performing OLAP-based data analytics on the ingested data. Druid is based on a shared

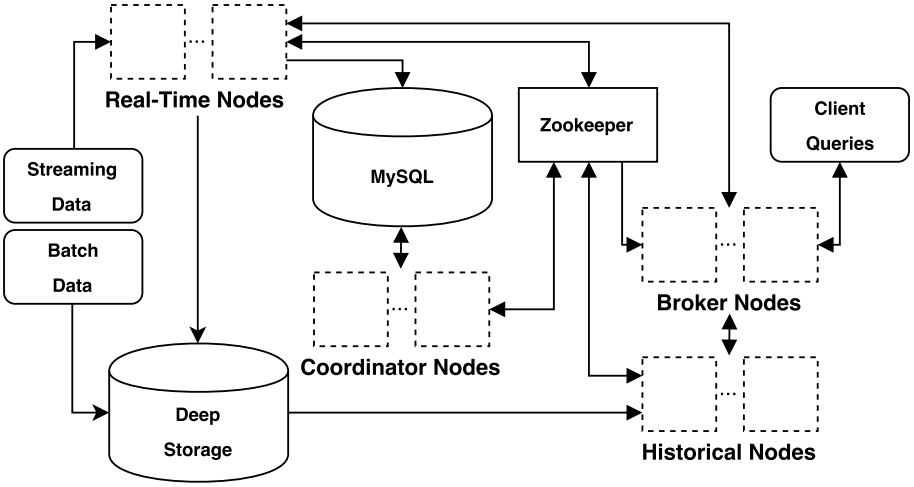

Figure A.8: The architecture of a Druid cluster annotated with information about data flow between nodes in the cluster, redrawn from [56]

nothing architecture coordinated through Zookeeper in-order to optimize the cluster for availability. Each task required by a Druid cluster is implemented as a specialized type of node as shown in Figure A.8. Real-Time nodes ingest events and store them in a local buffer to efficiently process queries on recent events in real-time. Periodically a background task collects all events at the node and converts them into immutable segments and uploads them to a DFS. Historical nodes execute queries for segments stored in the DFS and caches the segments used in the query to reduce query response time for subsequent queries. Coordinator nodes manage the cluster's configuration stored in a MySQL database, in addition to controlling which segments each historical node must serve. Last, the broker nodes accept queries formatted as JSON through HTTP POST requests and route them to the relevant real-time and historical nodes. When the query result is returned it is cached at the broker for subsequent queries. Druid does not provide any support for stream processing, and the authors suggest combining Druid with a stream processing engine. However, Druid does support aggregation of time series when ingested or when queried, and provides a set of aggregate functions while also supporting user-defined aggregation functions in addition to HyperLogLog for approximate aggregates.

Huang et al. [57] designed a distributed TSMS that uses IBM Informix for storage and provides a uniform SQL interface for querying both time series and relational data together. The system is classified as an Operational Data Historian (ODH) by the authors, marking monitoring as a clear priority. The TSMS consists of three components: A configuration component, a storage component, and a query component as shown in Figure A.9. The configuration component manages metadata for use by other components in the system. Concretely, it provides information about the data

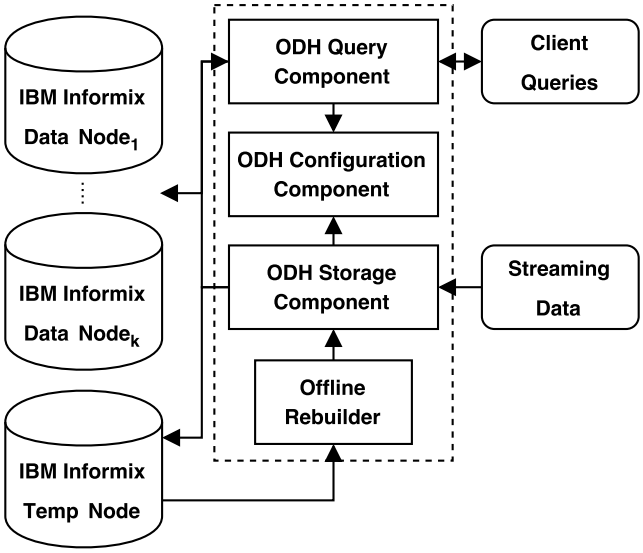

Figure A.9: Architecture of the TSMS proposed as an Operational Data Historian (ODH) by Huang et al. [57]. The figure was redrawn from [57]

sources the system is ingesting data from, and the IBM Informix instances available to the system. The storage component ingests data through a set of writer APIs into an appropriate data structure and compresses the data before it is stored. Different data structures are used depending on if the time series is regular or irregular. The three supported data formats are shown in Figure A.10. The first two formats store a single time series and contain four components, a starting timestamp, a device id, a counter and a blob of values. If the time series consists of data points separated by regular intervals only the value from the data points are stored without timestamps using the first format. If the time series is not regular then data points are stored as both the delta time compared to the starting timestamp and values using the second format. Last using the third format, a group of time series can be combined and stored in a single data structure, in which case the device id is changed to a group id and each element in the blob will then contain a reduced device id, a time delta and a value. To compress time series segments two existing compression algorithms are used, both supporting either lossless compression or lossy compression within an error bound. For stable values the data is compressed using a linear function, while quantization is used for fluctuating time series. The query component provides a virtual table as the query interface for the time series, allowing SQL to be used as a uniform query interface for both time series and relational data stored in the same TSMS.

Williams et al. [58] propose a distributed system for ingesting and analyzing time series produced by sensors monitoring industrial installations. The system is based on Pivotal's Gemfire in-memory data grid due to the authors arguing that a disk-based

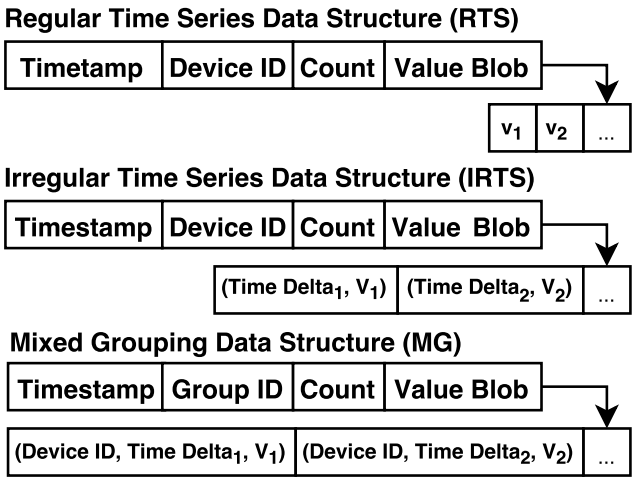

Figure A.10: The data structures for storage of time series used in the TSMS proposed by Huang et al. [57]. The figure was redrawn from [57]

system is an unsolvable bottleneck for data analytics. The data points produced by the sensors are ingested by a proprietary processing platform that cleans the data and performs real-time analytics before the data is inserted into the in-memory data grid. In the in-memory data grid, the cleaned data points are organized into a set of bins each containing a fixed time interval, sorted by time and identified by both the sensor id and the id of the machinery the sensor is installed on. The authors argue that each bin should only contain a few minutes of data as a compromise between duplicating the machine and sensor identifier, while also reducing the amount of data being retrieved during query execution to extract a subset of the data points in a bin. Inside a bin, the segment of data points produced by a specific sensor is stored as a doubly linked list, allowing memory usage to scale with the number of data points in the bin while achieving comparable read and write performance to a statically allocated circular array. Due to memory limitations, the in-memory data grid is used as a FIFO cache, storing all data points over a pre-specified time frame, with efficient transfer of the data to a long term disk based solution designated as future work. Additionally, neither AQP nor pre-commutation of aggregates when the data is ingested is supported by the system. For more information about in-memory big data storage and data analytics see the survey by Zhang et al. [80].

Bolt developed by **Gupta et al.** [59] is a distributed open-source TSMS designed for being embedded into applications to simplify the development of applications for IoT. Data is organized into tuples containing a timestamp and a collection of tag and value pairs, providing the opportunity to store not only readings from sensors but also annotate each reading with metadata. Bolt's implementation is based on encrypted streams in a configuration with one writer, the application that created the

stream, and multiple readers that can request readings from the stream. Sample-based AQP is provided as a function call with a parameter indicating how many elements should be skipped between each returned sample. The streams are chunked and stored sequentially in a log file on disk, while an index is kept in memory to provide efficient queries for tags or tags in a specific time frame. If the index reaches a pre-specified size, it is spilled to disk with only a summary kept in-memory, making querying historical data more expensive than recent data points. Sharing data between multiple instances of Bolt is achieved by having a metadata server provide information about the location of available streams, while exchange of the encrypted data is performed through either Amazon S3 or Microsoft Azure.

A similar TSMS *Storacle*, was developed for monitoring a smart grid and proposed by **Cejka et al.** [60]. The system is designed to have low system requirements allowing it to be used throughout a smart grid and connected to local sensors and smart meters. The TSMS provides the means for computing aggregates and in general process data from multiple sources locally before it is transmitted to the cloud over SSH. The cloud is defined as a remote storage with support for replication, offline data processing and remote querying. Storacle uses protocol buffers as its storage format and uses a three-tiered storage model consisting of in-memory storage, local storage and cloud storage. Storacle supports multiple parameters for configuring the amount of data that should be kept in-memory or stored locally for efficient access. However, the most recent data is kept when data is transferred to the next tier, ensuring that the most recent readings always are available. In addition to immutable storage of time series Storacle provides support for mutable storage of tags and meta-fields, both of which are lists of strings but tags can be used as part of a query while meta-fields cannot. Last, Storacle uses stream processing to process each ingested data point and produce aggregates from the data such as the number of data points, the average of their values, the min and max values, in addition to a histogram of observed values. Additional software built on top of Storacle is presented by the authors in the form of a CSV exporting tool and a tool that continuously retrieves the latest data points and computes aggregates for monitoring. In addition to the application presented in the original paper, Faschang et al. [61] proposed an integrated framework for active management of a smart grid in which Storacle was integrated with a message bus named GridLink.

Pelkonen et al. designed *Gorilla* [62] at Facebook as a distributed in-memory TSMS to serve as a caching layer for an existing system based on Apache HBase for monitoring the infrastructure at Facebook. Gorilla was designed to reduce query response time for data points collected in the last 26 hours, with the time frame determined by analyzing how the existing system was used. The system was designed as a cache instead of a replacement as the existing TSMS based on HBase contained petabytes of data. Data points ingested by Gorilla contain three components: a key, a timestamp and a value. The key is unique for each time series and used for distributing the data points, ensuring each time series can be mapped to a single host. Each time series is stored in statically sized blocks aligned with a two-hour window, with one block being written to at a time per time series and older blocks being immutable. Gorilla uses

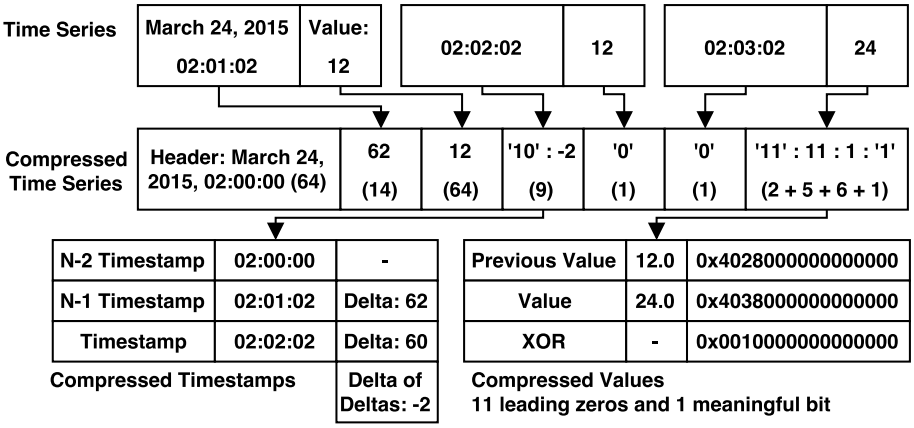

Figure A.11: Gorilla's compression with bit patterns written in single quotes and the size of values in bits written in parentheses, redrawn from [62]

a lossless compression method for both timestamps and values based on the assumption that data points are received at an almost regular interval as shown in Figure A.11. The method works by having each block be prefixed with a timestamp at full resolution, and the first data point stored with the timestamp encoded as the delta between the block's prefixed timestamp and the data point's timestamp, and the data point's value stored in full. Subsequent data points are compressed using two methods, one for timestamps and another for values. Timestamps are compressed by first computing the deltas between the current and previous timestamps and then storing a delta of these deltas using variable length encoding. Values are compressed by first XOR'ing the current and previous value, and then storing the difference with zero bits trimmed if possible. Both methods resort to storing a single zero bit in the instance where the computed delta or XOR result show no difference. To ensure availability Gorilla replicates all data to a different data center but provides no consistency guarantees. As each query is redirected to the nearest location the TSMS trades lower query response time for consistency. For fault tolerance Gorilla achieves persistence by writing the time series data to a DFS with GlusterFS being used at the time the paper was written. Data processing and analytics are performed by separate applications that retrieve blocks of compressed data from the in-memory TSMS through a client library, with applications for computing correlation between time series, real-time plotting and computation of aggregates being developed. Similarly, no capabilities for stream processing are implemented directly as part of Gorilla. An open-source implementation of the ideas presented in the paper was published by Facebook as the Beringei project.

Mickulicz et al. [63] propose a TSMS for data analytics using a novel approach for executing approximate aggregate queries on time series with the storage requirement defined by the time interval and not the size of the raw data points. By defining what aggregate queries will be executed, what granularity aggregates will be required in

terms of time, and with what error bound, a hierarchy of aggregates can be computed during data ingestion. The computed aggregates are stored in a binary search tree with the aggregates spanning the smallest time intervals at the leaves and an aggregate over the entire time series at the root. This structure enables efficient query response time for aggregate queries with differently sized time intervals, as aggregates over large time intervals can be answered using aggregates at the root of the tree, while queries over smaller time intervals use aggregates from the leaves. The presented approach is implemented as part of a distributed TSMS used for analyzing time series of events from mobile applications. The system consists of three components: a set of aggregation servers that computes multiple aggregates based on the ingested events, multiple MySQL RDBMSs for storing both the unmodified events and the aggregates computed, and last a querying interface routing queries to the relevant RDBMS. In terms of aggregates, the system uses a simple sum for counting events, HyperLogLog for approximately computing the number of distinct values, and the Count-Min Sketch for approximate frequency estimation.

servIoTicy is a TSMS implemented by **Pérez et al.** [64] for storing time series and metadata from IoT devices and is split into a frontend and a backend component. The frontend provides a REST API serving JSON for interacting with the system. To increase the number of devices that can communicate with the system, the REST API is accessible using multiple different communication protocols. The backend provides data storage using a combination of Couchbase and Elasticsearch for storage and query processing. Data is stored in Couchbase as one of two differently structured JSON documents. The first JSON format is used for storing metadata about the IoT devices the system is ingesting data from, while the second format is used to store data received from each IoT device. To reduce query processing time, the data stored in Couchbase is indexed by Elasticsearch. Stream processing using user-defined topologies is implemented through integration with Apache Storm. Apache Storm has also been extended with a mechanism for updating a function being executed in an Apache Storm Bolt. The system can then execute a specific version of the Bolt depending on the data points being processed. To change the topology it is still required that the system is terminated before a new topology can be deployed. In terms of AQP no support is currently provided by servIoTicy. Due to servIoTicy being part of the COMPOSE project [81], it was later integrated with the web service discovery system iServe, to augment the data stored in servIoTicy as documented by Villalba et al. [65].

BTrDB proposed by **Andersen et al.** [66, 67] is an open-source distributed TSMS designed for monitoring high precision power meters producing time series with timestamps at nanosecond resolution. A new system was developed as existing TSMSs were evaluated and determined to be unable to support the required ingestion rate and resolution of the timestamps. In addition to the high resolution no guarantees can be made about the ordering of data points or if each data point only will be received once. In BTrDB both problems are solved by storing each time series as a copy-on-write k-ary tree with each leaf node in the tree containing data points for a user-defined time interval and each internal node containing aggregates such as min, max, and mean of

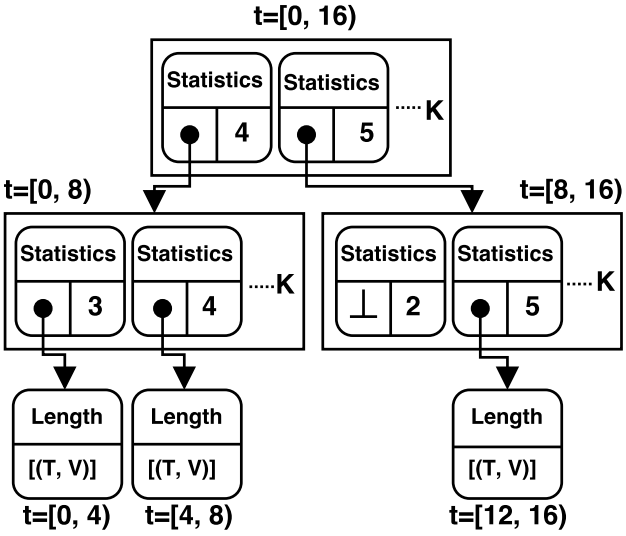

Figure A.12: The copy-on-write K-ary tree stores time series in the leafs while aggregates and version numbers are in non-leaf nodes, redrawn from [66]

the data points stored in children nodes. An example of the tree is shown in Figure A.12. This data structure provides multiple benefits. Data points received out of order can be inserted at the correct leaf node without any changes to the rest of the stored time series, support for efficient queries at multiple different resolutions is also enabled by the tree structure, as a time series can be extracted at multiple different resolutions by using the pre-computed aggregates, and last as the tree is a persistent data structure, older versions of a time series are available despite new data points being inserted and other being deleted. The system is split into three components. BTrDB provides the query interface, optimization of data ingestion through buffering, and manipulation of the k-ary trees. A DFS in the form of CEPH is used for persistent storage of the k-ary trees constructed for each time series, and last MongoDB is used to store the metadata necessary for BTrDB. To reduce the storage requirements for the trees, compression is applied. To compress a sequence, the compression algorithm computes the delta to the mean of the previous deltas in the sequence, and the computed deltas are then encoded with Huffman encoding. Apart from the API being structured around streams BTrDB does not provide any stream processing capabilities. However, the low query responds time for reads and writes provided by the TSMS were utilized to implement the DISTIL stream processing framework that integrates with BTrDB [67].

4.3 Discussion

The main domains covered by systems with external data stores are IoT, monitoring of industrial or IT systems, and management of scientific data. The developed systems are in general complete, in use, and support larger data sets by scaling out through distributed computing and caching of data in memory for efficient processing of either the most recently collected or queried data. TSDS [44] and Tristan [54] are exceptions as both of them operate as centralized systems limiting their scalability. An interesting observation is also that none of the systems implement an entirely new data storage layer, and instead to varying degrees reuse existing systems for example MySQL, HDFS and Couchbase. Only three systems, the system by Williams et al. [58], Gorilla [62], and servIoTicy [64], provide no capabilities for constructing approximate representation of time series while the remaining systems support either simple aggregates or AQP. The systems implementing AQP using models are either research systems or provide limited functionality for implementing additional user-defined models. The system by Guo et al. [51–53] is used to demonstrate a new indexing and query method and provides a generic schema for storing the models. Tristan employs AQP by representing time series using multiple smaller fixed sized time series but the TSMS provides no interface for implementing other methods for representing time series suitable for AQP. Druid [56] allows implementation of aggregation methods through a Javascript scripting API or through a Java extensions API. The TSMS developed by Huang et al. [57] provides no mechanisms for implementing new models for use by the system and only two types of models are implemented and used for compression. No interface is documented for extending the system by Mickulicz et al. [63] with new models, however, the authors note that the approximate representation used as part of the proposed tree structure can take different forms depending on the queries the tree the must support. In summary, only Druid [56] and to a certain degree the system by Mickulicz et al. [63] provide an interface for end users to implement alternative representations of time series so domain experts can use models optimized for a particular domain. Multiple aspects of stream processing are utilized by this category of systems. Stream processing using user-defined computation is provided by SensorGrid [50] through the use of SQL window functions, while servIoTicy [64] and the TSMS by Williams et al. [58] support user-defined functions. Tristan [54], Guo et al. [51–53], Huang et al. [57], and Mickulicz et al. [63] construct models from data points online. Last, SciDB [45–47], Bolt [59] and BTrDB [66, 67], logically structure some APIs as streams.

5 RDBMS Extensions

5.1 Overview

Existing RDBMSs have been extended with native functionality for efficient storage, querying and analysis of time series stored in the RDBMS. Implementing functionality for processing time series data directly into an RDBMS provides multiple benefits. It

simplifies analysis by providing storage and analysis through one system, removing the need for data to be exported and the analysis performed using an additional program such as R or SPSS. Existing functionality from the RDBMS can be reused for the implementation of the extensions. Last, extending a known system, such as PostgreSQL, allows knowledge of said system to be reused. However, as with all extensions of existing systems, decisions made about the RDBMS implementation can restrict what changes can be made, and functionality such as transactions add unnecessary overhead if the time series is immutable.

5.2 Systems

TimeTravel, by **Khalefa et al.** [68], extends PostgreSQL with the capability to do model-based AQP and continuously compute forecasts for time series stored in the centralized RDBMS. Using forecasting and an extended SQL query language, TimeTravel simplifies data analytics by providing a uniform interface for both exact query processing used for historical values and approximate query processing used for historical and future values. Use of AQP is necessary as the computed forecasts will have some estimation error due to the exact values being unknown. However, the use of AQP is also beneficial for reducing query time on historical data. The architecture of TimeTravel can be seen in Figure A.13, and consists of a hierarchical model index, offline components for building and compressing the model index, and online components for query processing and maintenance of the model index. The models range from a coarse grained model with a high error bound at the top to multiple

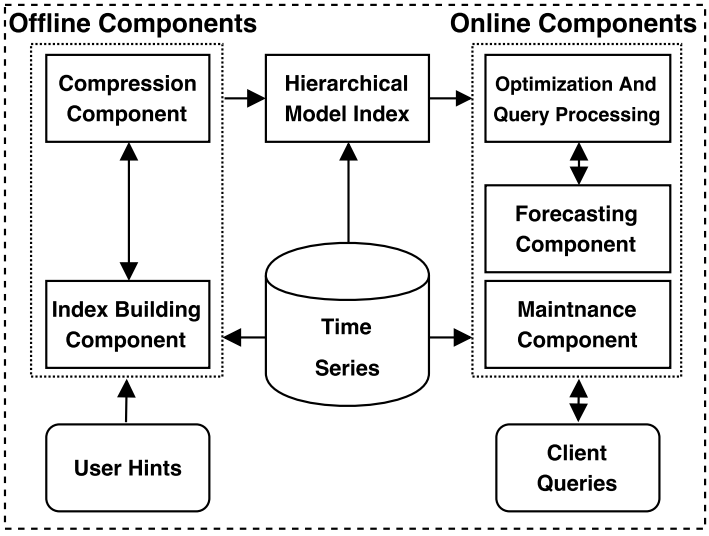

Figure A.13: Architecture of the TimeTravel TSMS with components for index construction and query processing. The figure was redrawn from [68]

finer grained models with lower error bounds at the bottom. This hierarchy allows the system to process queries more efficiently by using a model of the right granularity based on the required accuracy of the result. To build the model hierarchy, the system requires multiple parameters to be specified: First, hints about the time series seasonal behavior, second, error bound guarantees required for the application that will query the data, and last what method to use for forecasting. The system builds the hierarchical model index by creating a single model, and then based on that model's error, it splits the time series into more fine-grained models based on the required error bound and provided seasonality. In terms of stream processing, TimeTravel provides no extensions to PostgreSQL other than the module for maintaining the model index automatically. The methods created as part of TimeTravel have been incorporated into a Electricity Data Management System (EDMS) developed as part of the MIRABEL smart grid project [82] by Fisher et al. [69]. To facilitate both exact and approximated analysis of time series in the EDMS, a real-time data warehouse is implemented as part of the system. For approximate queries the AQP functionality implemented as part of TimeTravel were used while exact queries were answered from the data points stored in an array in PostgreSQL.

Another endeavor to extend PostgreSQL with the capability to forecast time series in a centralized RDBMS is *F^2DB* proposed by **Fischer et al.** [70]. The use case for F^2DB is data analytics using data warehouse based business intelligence, and the system extends the SQL query language with statements for manually creating models and performing forecast on said models. As an alternative to creating a model by specifying it as part of an SQL statement, F^2DB can compute which model provides the best fit for a given time series based on an evaluation criteria. The system provides a general architecture for implementing forecast models, allowing domain experts to add additional models to the system so they can be used through the SQL query language. The current implementation provides the means to create models based on ARIMA and exponential smoothing. All models are stored in a central index and maintained automatically by the system as shown in Figure A.14, but no extensions providing stream processing are provided. To facilitate efficient query execution, the model index provides an interface for describing the hierarchical relationship between models to the query language. Based on query statistics and model performance, an alternative configuration of models might be proposed by a model advisor [71]. The advisor selects a configuration of models to use by utilizing a set of heuristics named indicators. Each indicator is either focused on a single time series or the relationship between multiple time series in the dimensional hierarchy. The choice of one of multiple models is determined based on a trade-off between accuracy and performance cost. The produced model configuration can then be loaded into F^2DB.

Bakkalian et al. [72] present a proof-of-concept PL/SQL extension for the Oracle RDBMS. The implemented extension allows time series to be stored and queried as linear functions in an OLAP data model. The system splits storage of a time series into two tables: the raw data is organized in one table as events with each row representing an event with a timestamp, metadata, and a value. Another table stores the intervals

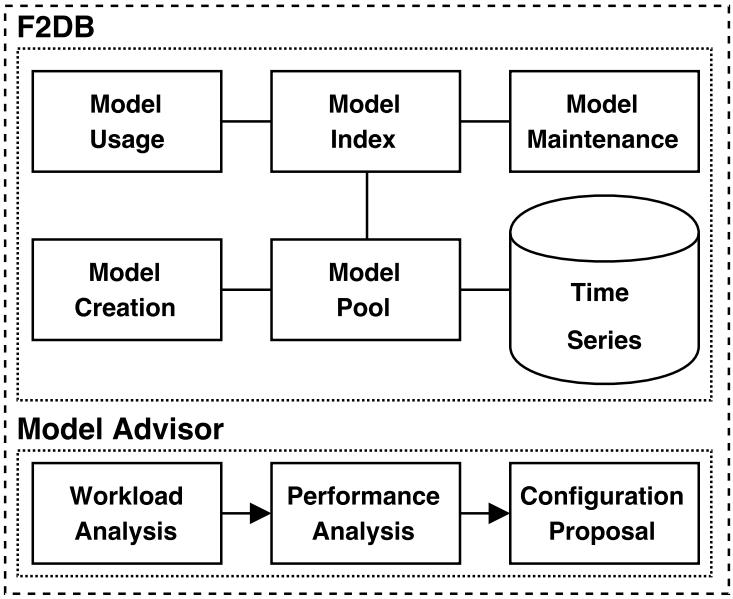

Figure A.14: Overview of the F^{2}DB TSMS. Models are constructed from time series stored in the base tables, indexed, and then used by the model usage component for query processing. The figure was redrawn from [70]

between each two consecutive events as a model, with the current implementation supporting linear functions only. Queries are executed directly against the models as they substitute the raw data representation. The use of linear functions allows the system to interpolate values not present in the raw data set to support AQP. The authors also discuss the theoretical use of models for forecasting time series as a benefit of a model-based approach to time series storage and querying. However, no information is provided about support for forecasting being implemented. In addition, no support for stream processing is described in the paper. The proposed system builds directly upon the following two publications in which some of the authors participated. Bebel et al. [73] proposed a model for OLAP analysis of sequential data that formalized the notion of sequences as facts and where sequences were formalized as an ordered collection of events. Building upon the model proposed in the previously mentioned paper, Koncilia et al. [74] proposed a new model enabling OLAP analysis of sequential data by formally introducing the notion of intervals as the time between two consecutive events, as well as defining a sequence to be an ordered set of intervals.

5.3 Discussion

TimeTravel [68] and F^{2}DB [70] provide similar capabilities for forecasting time series stored in an RDBMS. However, the intended use case for the systems differs.

TimeTravel is focused on approximating time series using models to provide AQP for historical, present, and future values. In addition, a hierarchy of models is created so the system can select a model based on the necessary accuracy to decrease query time, and allow users to indicate what seasonality the time series will exhibit. F^2DB is motivated by a need to perform forecasting in a data warehouse and use the multi-dimensional hierarchy as part of the modelling process. The systems also differ in-terms of their ability to perform AQP as TimeTravel supports execution of AQP queries on both historical data and as forecast queries, while F^2DB's focus is on forecast queries. The system by Bakkalian et al. [72] represents time series using models but does not focus on their use for forecasting. This system instead uses models to reduce the storage requirement and query response time for OLAP queries. In addition to extending PostgreSQL and Oracle RDBMS with new methods for storing and querying time series, researchers have also extended RDBMSs with functionality for analytics without any changes to data storage through for example MADLib [83] and SearchTS [84]. Despite all of surveyed TSMSs in this section extend an RDBMS with AQP functionality, none of the systems implement functionality for stream processing.

6 Future Research Directions

The increase in time series data produced has lead researchers to discuss new directions for research into development of TSMSs, with new research opportunities being created as a result of this effort.

6.1 Research Directions Proposed in Literature

Cuzzocrea et al. [85] detail how the combination of sensor networks with cloud infrastructure enables the development of new types of applications, detail problems with the current state-of-the-art, and present a set of open problems regarding the combination of sensor networks with cloud infrastructure. The first problem presented is that different sources of sensor data often are heterogeneous, increasing the difficulty of combining different data sources for analytics. Heterogeneous data sources also complicate the addition of structure to a data set as a prerequisite for an analytical task. Another problem described is the amounts of sensor data produced which add additional requirements to the scalability of TSMSs. The authors propose to utilize methods from both RDBMSs and NoSQL database systems, and develop new systems with a high focus on scalability, primarily through horizontal partitioning and by reducing the reliance on join operations.

Cuzzocrea [86] provides an overview of the state-of-the-art methods for managing temporal data in general. However, while the general topic of the paper is temporal data management, a discussion of existing research and future research direction in the area of sensor data and scientific data management is also included in the paper. In addition, the author provides a discussion of additional open research questions within

the area of temporal and sensor data management. For this summary only the proposed research directions relevant for management of time series data are included. The presented research directions are primarily centered around the problem of scalability. First the author argues that new storage methods are required, in conjunction with additional research into indexing mechanisms. In terms of data analysis, the author sees a hybrid system combining in-memory and disk based storage together with the next generation of system using the distributed data processing model popularized by Apache Hadoop as an import research direction. This architecture was later realized for general purpose data by SnappyData [87, 88]. In addition, the author proposes that further research into AQP method be performed, as such methods have already been proven successful for sensor data. The development of innovative analytical methods robust against imprecise values and outliers is presented as a pressing concern, and new methods for data visualization are presented as being necessary, as existing solutions for visualization of time series at different resolutions cannot be used at the scale required today.

Additional arguments for further research into the development of TSMSs were presented by **Shafer et al.** [12]. Traditional RDBMSs support functionality unnecessary for processing time series, such as the ability to delete and update arbitrary values, with the argument that supporting functionality that is not necessary for analyzing time series adds complexity and overhead to these systems. In addition, since time series analysis often follows a common set of access patterns, creating a system optimized specifically for these access patterns should lead to a performance increase over a general purpose DBMS. The authors present an overview of some existing TSMSs and DBMSs suitable for time series analysis, and argue that they all effectively function as column stores with additional functionality specific for processing time series added on top. Using a column store for storing and processing time series provides the benefit of run length compression and efficient retrieval of a specific dimension of a time series [89]. However, the authors dispute the use of column stores and describe a set of design principles for a time series data store as shown in Figure A.15. They propose that time series data store should separate time series into timestamps and values, the values be partitioned based on their origin to make appending them to a time ordered list trivial, and last the values should be archived in blocks ordered by time to allow for efficient access to a continuous part of the time series. A preliminary evaluation demonstrates that using these principles, a higher compression rate compared to existing RDBMS and TSMS, can be achieved.

Sharma et al. [1] present new research directions for time series analysis in the area of Cyber-Physical Systems (CPSs). Creating CPSs by combining sensor networks with software for detailed analysis and decision-making allows for a new wave of informed automation system based on sensor readings. However, the authors argue that before such systems can be realized at a large scale, multiple challenges must be resolved. First, the lack of information about how changes to a CPS change the sensor readings produced, prevents systems from automatically controlling the monitored system based on information from the sensors. As a possible solution the authors

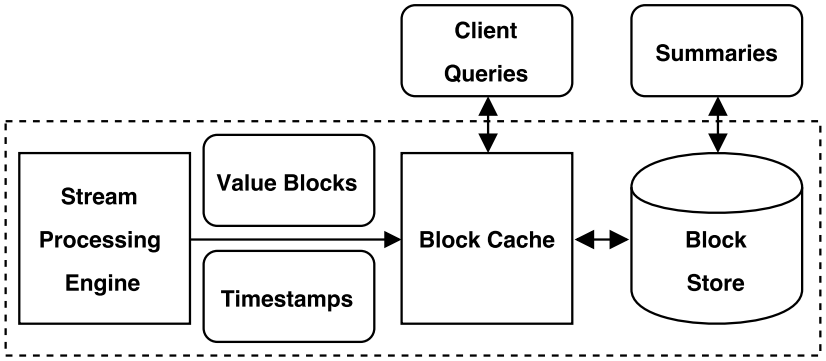

Figure A.15: Architecture proposed by Shafer et al. [12], redrawn from [12]

propose using graph-based methods to infer the relationship between changes to a system and its sensor readings. Changes in discrete state could also be used to predicate changes to sensor readings, such as predicting a change in the speed of a car if cruise control is turned on or off. Another problem presented is scaling systems large enough to analyze the amount of low quality data large networks of cheap sensors produce, due both to the amount of data produced and the necessity to clean it before it can be analyzed. For this problem, the authors propose that existing methods designed for detection of sequential patterns in symbolic data could be effective for time series data mining, if efficient methods for converting time series to a symbolic representation can be created.

Another proponent for the development of specialized systems for analysis of time series and data series is **Palpanas** [3–5] as he argues that the development of a new class of systems he names Data Series Management System (SMS) is a requirement for analysis of data series to continue to be feasible as the amounts of data from data series increase. However, for an SMS to be realized, methods enabling a system to intelligently select appropriate storage and query execution strategies based on the structure of the data is necessary. As time series from different domains exhibit different characteristics, the system should be able to take these characteristics into account, and utilize one or multiple different representations to minimize the necessary storage requirement while still allowing for adaptive indexes to be created and queries to be executed efficiently. Parallel and distributed query execution must be an integrated part of the system to take advantage of not only multi-core processor and SIMD instructions, but also the scalability provided by distributed computing. To determine an appropriate execution method for a query in an SMS a cost-based optimizer specialized for data series would also be a required component, and it should abstract away the parallel execution of the analysis and indexing methods implemented in the system. However, the author argues that no existing cost-based optimizer is suitable for use in a SMS. In addition to the proposed system architecture, an argument for the development of a standardized benchmark for comparison of SMSs is also presented.

6.2 Online Model-Based Sensor Warehouse

In addition to the proposed research direction into storage and analysis of time series, and based on the surveyed systems, we see the following areas as being of primary interest for the scalability and usability of TSMSs to be further increased. The end result of the research we propose is a distributed TSMS with a physical layer storing time series as mathematical models, serving as the DBMS for a distributed data warehouse designed specifically for analysis of sensor data collected as time series and updated in real-time.

Online Distributed Model-Based AQP: Among the systems in the survey, few utilized models to approximate time series and provide a means for AQP, opting instead to reduce query response time through pre-computed aggregates and distributed computing. However, we believe that for use cases where approximate results are acceptable, the use of models provide a more flexible means for AQP. This is due to the configurable error bound and storage requirements, the capability to infer missing values and remove outliers, and due to some queries being answerable directly from the models, dramatically decreasing query response time. Similarly, to the use of models for distributed batch processing, the use of models for distributed stream processing should be evaluated, with the possibility that new algorithms could be developed due to the reduction in bandwidth use. Research in this direction has been performed by Guo et al. [51–53]. However, the proposed system does primarily demonstrate a new indexing solution. In addition, the system only provides limited support for low latency stream processing and the system does not implement a declarative query language. Centralized TSMSs that use models as an integral part of the system and provide a declarative interface have been designed, for example Plato [38], TimeTravel [68] and F^2DB [70]. However, as these systems do not utilize distributed computing, they are limited in terms of scalability.

Automatic Model Management: Additional support for helping users determine how model-based AQP should be utilized is necessary. Most TSMSs using model-based AQP either provide no means for implementing and selecting models appropriate for a particular time series or relegate the task to domain experts [38], with only a few systems, for example F^2DB [70, 71], providing a mechanism for automatic selection of models. While development of new models and integration of such into a TSMS properly will need to be performed manually for some time, a TSMS supporting AQP using models should at minimum implement methods for automatically fitting models to time series based on an error function for both batch and streaming data. While researchers have proposed some methods for automatically fitting models to time series stored in a DBMS, only a few publications present methods for approximating times series by automatically selecting appropriate models in real-time [90, 91]. Inspired by similar proposals in the area of machine learning [92], we propose that methods and systems be developed that interactively guide non-expert user to select appropriate models for a particular time series without the need for an exhaustive search when data

is stored in a DBMS, and provide efficient selection of models for segmenting and approximating time series with a pre-specified error bound when ingested. Similarly, such a TSMS should as part of query processing provide insights into how an error bound should be interpreted and how a model might have effected the time series it represents.

Model-Based Sensor Data Warehousing: A distributed TSMS using models for stream and batch processing of time series has the capability to scale through the use of distributed processing by adjusting the acceptable error bound for a query, through selection of alternative models, and by implementing a query optimizer capable of taking the properties of the models used into account. However, to enable simple analytics on top of the model, an abstraction is needed. One possible abstraction, the data cube, has already been used successfully for analyzing time series data with the MIRABEL project as an example [69, 82], while the methods proposed by Perera et al. [77] and Bakkalian et al. [72] demonstrated that, for centralized systems, representing time series using models in a data cube can lead to reduced storage requirements and query response times. However, a system utilizing models for representing time series in a data cube has, to the best of our knowledge, never been successfully integrated with methods for continuously updating a data cube such as for example Stream Cubes [93] or systems such as Tidalrace [94], in a distributed system. Another benefit of using a data cube is that it is a well-known by data analysts, providing users of our envisioned TSMS with a familiar interface for analytics of multi-dimensional time series. Therefore, we see a data cube as a useful abstraction for analyzing multi-dimensional time series represented using models at the physical layer, if methods for continuously maintaining an OLAP cube can be unified with the methods for representing time series using models in a distributed TSMS.

7 Conclusion

The increasing amount of time series data that is being produced requires that new specialized TSMSs be developed. In this paper we have provided a systematic and thorough analysis and classification of TSMSs. In addition, we have discussed our vision for the next generation of systems for storage and analysis of time series, a vision that is based on the analysis in this survey and the directions for future work proposed by other researches in the field.

From our analysis we provide the following conclusions. TSMSs that use an *internal data store* and integrate it directly with the query processing component are predominately centralized systems, while distributed TSMSs are being developed using existing DFSs and distributed DBMSs. Research into systems with an internal data store is instead primarily focused on systems for embedded devices or which function as a proof-of-concept implementation of a new theoretical method. This point is further reinforced as only a few of the proposed TSMSs in the internal data store category can be considered mature enough for deployment in a business critical scenario. This

situation is contrasted by the set of systems using *external data stores*, as a larger portion of these systems are distributed and developed by or in collaboration with a corporate entity where the system then can be deployed in order to solve a data management problem. As none of the TSMSs surveyed use a distributed storage system developed and optimized exclusively for persistent storage of time series, it is an open question what benefit such a storage solution would provide. Last, systems built as *extensions to existing RDBMSs* are few and all extend a RDBMS with support for AQP through the use of mathematical models. Other extensions to RDBMSs have added new methods for query processing but no additional functionality for storage of time series. The described RDBMSs are in general complete in terms of development, two of the systems were integrated into larger applications for use in a smart grid and the last being a prototype. However none of the RDBMS systems are distributed, limiting their ability to scale.

Additional functionality built on top of the central storage and query processing components is scarce in all three categories of systems. At the time of writing, only a limited number of systems provide any capabilities for stream processing to support transformation of the data as it is being ingested. In contrast to implementing stream processing during data ingestion, some TSMSs provide mechanisms for piping the data through a set of transformations after it has been stored in the TSMS or structure APIs around streams. Compared to stream processing, more TSMSs implement methods for approximating time series as a means to reduce query response time, storage requirements, or provide advanced analytical capabilities. However, for the systems that implement methods for approximating time series, it is uncommon to have an interface for domain experts to extend the systems with additional user-defined methods or models optimized for a particular domain or data set.

The future research directions proposed by experts in the field are primarily focused on the need for a TSMS with a storage solution and query processor developed from the ground up for time series, instead of reusing existing components or data models from for example an RDBMS for storage of time series. Additionally, it is proposed that such systems should support in-memory, parallel and distributed processing of time series as a means to reduce query processing time enough for interactive data analytics and visualization. AQP is mentioned as another possibility for reducing query processing time. As future work we propose that a TSMS that inherently provides a data cube as the means for analyzing multi-dimensional time series be developed. The system should provide interactive query speeds through the use of distributed in-memory processing and incorporate AQP as a central part of the system from the start. The system must support stream processing to transform and clean the data as it is ingested, and provide the means for construction of user-defined models, in order to compress the time series and enable AQP.

In summary, we propose that a distributed TSMS providing the same analytical capabilities as a data warehouse be developed specifically for use with time series. The TSMS should provide functionality that allows the data to be updated in real-time, support stream processing using user-defined functions and allow queries to be executed

on both historical and incoming data at interactive query speed through the use of AQP.

8 Acknowledgments

This research was supported by the DiCyPS center funded by Innovation Fund Denmark [95].

References

[1] A. B. Sharma, F. Ivančić, A. Niculescu-Mizil, H. Chen, and G. Jiang, "Modeling and Analytics for Cyber-Physical Systems in the Age of Big Data," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 4, pp. 74–77, 2014.

[2] J. Ronkainen and A. Iivari, "Designing a Data Management Pipeline for Pervasive Sensor Communication Systems," *Procedia Computer Science*, vol. 56, pp. 183–188, 2015.

[3] T. Palpanas, "Data Series Management: The Road to Big Sequence Analytics," *ACM SIGMOD Record*, vol. 44, no. 2, pp. 47–52, June 2015.

[4] ——, "Big Sequence Management: A glimpse of the Past, the Present, and the Future," in *Proceedings of the 42nd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. Springer, 2016, pp. 63–80.

[5] ——, "Data Series Management: The Next Challenge," in *Proceedings of the 32nd International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2016, pp. 196–199.

[6] T.-c. Fu, "A review on time series data mining," *Engineering Applications of Artificial Intelligence*, vol. 24, no. 1, pp. 164–181, February 2011.

[7] P. Esling and C. Agon, "Time-Series Data Mining," *Computing ACM Surveys (CSUR)*, vol. 45, no. 1, 2012.

[8] C. C. Aggarwal, Ed., *Managing and Mining Sensor Data*. Springer, 2013.

[9] C. C. Aggarwal, *Data Mining: The Textbook*. Springer, 2015.

[10] M. Garofalakis, J. Gehrke, and R. Rastogi, Eds., *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2016.

[11] M. Stonebraker and U. Cetintemel, ""One Size Fits All": An Idea Whose Time Has Come and Gone," in *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE, 2005, pp. 2–11.

[12] I. Shafer, R. R. Sambasivan, A. Rowe, and G. R. Ganger, "Specialized Storage for Big Numeric Time Series," in *Proceedings of the 5th conference on Hot Topics in Storage and File Systems (HotStorage)*. USENIX, 2013, pp. 15–15.

[13] P. Seshadri, M. Livny, and R. Ramakrishnan, "The Design and Implementation of a Sequence Database System," in *Proceedings of 22th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 1996, pp. 99–110.

[14] ——, "SEQ: A model for sequence databases," in *Proceedings of the 11th International Conference on Data Engineering (ICDE)*. IEEE, 1995, pp. 232–239.

[15] ——, "Sequence Query Processing," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 1994, pp. 430–441.

[16] A. Lerner and D. Shasha, "AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments," in *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 2003, pp. 345–356.

[17] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi, "A Sequential Pattern Query Language for Supporting Instant Data Mining for e-Services," in *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 2001, pp. 653–656.

[18] ——, "Optimization of Sequence Queries in Database Systems," in *Proceedings of the 20th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. ACM, 2001, pp. 71–81.

[19] ——, "Expressing and Optimizing Sequence Queries in Database Systems," *ACM Transactions on Database Systems (TODS)*, vol. 29, no. 2, pp. 282–318, June 2004.

[20] N. Tsiftes and A. Dunkels, "A Database in Every Sensor," in *Proceedings of the 9th Conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 2011, pp. 316–332.

[21] B. Wang and J. S. Baras, "HybridStore: An Efficient Data Management System for Hybrid Flash-Based Sensor Devices," in *Proceedings of the 10th European Conference on Wireless Sensor Networks (EWSN)*. Springer, 2014, pp. 50–66.

[22] G. Douglas and R. Lawrence, "LittleD: A SQL Database for Sensor Nodes and Embedded Applications," in *Proceedings of the 29th Symposium on Applied Computing (SAC)*. ACM, 2014, pp. 827–832.

[23] A. Cuzzocrea, J. Cecilio, and P. Furtado, "StreamOp: An Innovative Middleware for Supporting Data Management and Query Functionalities over Sensor Network Streams Efficiently," in *Proceedings of the 17th International Conference on Network-Based Information Systems (NBiS)*. IEEE, 2014, pp. 310–317.

[24] C. Pungilă, T.-F. Fortiş, and O. Aritoni, "Benchmarking Database Systems for the Requirements of Sensor Readings," *IETE Technical Review*, vol. 26, no. 5, pp. 342–349, 2009.

[25] T. W. Wlodarczyk, "Overview of Time Series Storage and Processing in a Cloud Environment," in *Proceedings of the 4th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2012, pp. 625–628.

[26] T. Goldschmidt, A. Jansen, H. Koziolek, J. Doppelhamer, and H. P. Breivold, "Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes," in *Proceedings of the 7th International Conference on Cloud Computing (CLOUD)*. IEEE, 2014, pp. 602–609.

[27] D. Namiot, "Time Series Databases," in *Proceedings of the XVII International Conference Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL)*. CEUR-WS.org, 2015, pp. 132–137.

[28] A. K. Kalakanti, V. Sudhakaran, V. Raveendran, and N. Menon, "A Comprehensive Evaluation of NoSQL Datastores in the Context of Historians and Sensor Data Analysis," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 1797–1806.

[29] A. Bader, O. Kopp, and F. Michael, "Survey and Comparison of Open Source Time Series Databases," in *Datenbanksysteme für Business, Technologie und Web (BTW) - Workshopband*. GI, 2017, pp. 249–268.

[30] L. Deri, S. Mainardi, and F. Fusco, "tsdb: A Compressed Database for Time Series," in *Proceedings of the 4th International Workshop on Traffic Monitoring and Analysis (TMA)*. Springer, 2012, pp. 143–156.

[31] U. Khurana, S. Parthasarathy, and D. S. Turaga, "FAQ: A Framework for Fast Approximate Query Processing on Temporal Data," in *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications (BigMine)*. JMLR.org, 2014, pp. 29–45.

[32] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale, "WearDrive: Fast and Energy-Efficient Storage for Wearables," in *Proceedings of the Annual Technical Conference (ATC)*. USENIX, 2015, pp. 613–625.

[33] J. Li, A. Badam, R. Chandra, S. Swanson, B. Worthington, and Q. Zhang, "On the Energy Overhead of Mobile Storage Systems," in *Proceedings of the 12th Conference on File and Storage Technologies (FAST)*. USENIX, 2014, pp. 105–118.

[34] K. Zoumpatianos, S. Idreos, and T. Palpanas, "RINSE: Interactive Data Series Exploration with ADS+," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1912–1915, August 2015.

[35] ——, "Indexing for Interactive Exploration of Big Data Series," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 1555–1566.

[36] ——, "ADS: the adaptive data series index," *The VLDB Journal (VLDBJ)*, vol. 25, no. 6, pp. 843–866, December 2016.

[37] K. S. Perera, M. Hahmann, W. Lehner, T. B. Pedersen, and C. Thomsen, "Modeling Large Time Series for Efficient Approximate Query Processing," in *Database Systems for Advanced Applications - DASFAA International Workshops, SeCoP, BDMS, and Posters*. Springer, 2015, pp. 190–204.

[38] Y. Katsis, Y. Freund, and Y. Papakonstantinou, "Combining Databases and Signal Processing in Plato," in *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR)*, 2015.

[39] B. Chardin, J.-M. Lacombe, and J.-M. Petit, "Chronos: a NoSQL system on flash memory for industrial process data," *Distributed and Parallel Databases*, vol. 34, no. 3, pp. 293–319, September 2016.

[40] B. Chardin, O. Pasteur, and J.-M. Petit, "An FTL-agnostic Layer to Improve Random Write on Flash Memory," in *Proceedings of the 16th International Conference on Database Systems for Advanced Applications (DASFAA)*. Springer, 2011, pp. 214–225.

[41] A. L. Serra, S. Vila-Marta, and T. E. Canal, "Formalism for a multiresolution time series database model," *Information Systems*, vol. 56, pp. 19–35, March 2016.

[42] A. L. Serra, T. E. C. i Sebastia, and V. Marta, "A Model for a Multiresolution Time Series Database System," in *Proceedings of the 12th International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED)*. WSEAS, 2013, pp. 55–60.

[43] A. MacDonald, "PhilDB: the time series database with built-in change logging," *PeerJ Computer Science*, vol. 2, art. e52, 2016.

[44] R. S. Weigel, D. M. Lindholm, A. Wilson, and J. Faden, "TSDS: high-performance merge, subset, and filter software for time series-like data," *Earth Science Informatics*, vol. 3, no. 1-2, pp. 29–40, June 2010.

[45] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, "SciDB: A Database Management System for Applications with Complex Analytics," *Computing in Science & Engineering*, vol. 15, no. 3, pp. 54–62, May-June 2013.

[46] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The Architecture of SciDB," in *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management (SSDBM)*. Springer, 2011, pp. 1–16.

[47] T. S. D. Team, "Overview of SciDB: Large Scale Array Storage, Processing and Analysis," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2010, pp. 963–968.

[48] M. Buevich, A. Wright, R. Sargent, and A. Rowe, "Respawn: A Distributed Multi-Resolution Time-Series Datastore," in *Proceedings of the 34th Real-Time Systems Symposium (RTSS)*. IEEE, 2013, pp. 288–297.

[49] C. Palmer, P. Lazik, M. Buevich, J. Gao, M. Berges, A. Rowe, R. L. Pereira, and C. Martin, "Demo Abstract: Mortar.io: A Concrete Building Automation System," in *Proceedings of the 1st Conference on Embedded Systems for Energy-Efficient Buildings (BuildSys)*, 2014, pp. 204–205.

[50] A. Cuzzocrea and D. Saccà, "Exploiting compression and approximation paradigms for effective and efficient online analytical processing over sensor network readings in data grid environments," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 14, pp. 2016–2035, September 2013.

[51] T. Guo, T. G. Papaioannou, and K. Aberer, "Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud," *Big Data Research*, vol. 1, pp. 52–65, August 2014.

[52] T. Guo, T. G. Papaioannou, H. Zhuang, and K. Aberer, "Online Indexing and Distributed Querying Model-View Sensor Data in the Cloud," in *Proceedings of the 19th International Conference on Database Systems for Advanced Applications (DASFAA)*. Springer, 2014, pp. 28–46.

[53] T. Guo, T. G. Papaioannou, and K. Aberer, "Model-View Sensor Data Management in the Cloud," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2013, pp. 282–290.

[54] A. Marascu, P. Pompey, E. Bouillet, M. Wurst, O. Verscheure, M. Grund, and P. Cudre-Mauroux, "TRISTAN: Real-Time Analytics on Massive Time Series Using Sparse Dictionary Compression," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 291–300.

[55] A. Marascu, P. Pompey, E. Bouillet, O. Verscheure, M. Wurst, M. Grund, and P. Cudre-Mauroux, "MiSTRAL: An Architecture for Low-Latency Analytics on Massive Time Series," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2013, pp. 15–21.

[56] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A Real-time Analytical Data Store," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 157–168.

[57] S. Huang, Y. Chen, X. Chen, K. Liu, X. Xu, C. Wang, K. Brown, and I. Halilovic, "The Next Generation Operational Data Historian for IoT Based on Informix," in

Proceedings of the International Conference on Management of Data (SIGMOD). ACM, 2014, pp. 169–176.

[58] J. W. Williams, K. S. Aggour, J. Interrante, J. McHugh, and E. Pool, "Bridging High Velocity and High Volume Industrial Big Data Through Distributed In-Memory Storage & Analytics," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 932–941.

[59] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, "Bolt: Data management for connected homes," in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2014, pp. 243–256.

[60] S. Cejka, R. Mosshammer, and A. Einfalt, "Java embedded storage for time series and meta data in Smart Grids," in *Proceedings of the 6th International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2015, pp. 434–439.

[61] M. Faschang, S. Cejka, M. Stefan, A. Frischenschlager, A. Einfalt, K. Diwold, F. P. Andrén, T. Strasser, and F. Kupzog, "Provisioning, deployment, and operation of smart grid applications on substation level," *Computer Science - Research and Development*, vol. 32, no. 1–2, pp. 117–130, March 2017.

[62] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A Fast, Scalable, In-Memory Time Series Database," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1816–1827, August 2015.

[63] N. D. Mickulicz, R. Martins, P. Narasimhan, and R. Gandhi, "When Good-Enough is Enough: Complex Queries at Fixed Cost," in *Proceedings of the 1st International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 2015, pp. 89–98.

[64] J. L. Pérez and D. Carrera, "Performance Characterization of the servIoTicy API: an IoT-as-a-Service Data Management Platform," in *Proceedings of the 1st International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 2015, pp. 62–71.

[65] Á. Villalba, J. L. Pérez, D. Carrera, C. Pedrinaci, and L. Panziera, "servIoTicy and iServe: a Scalable Platform for Mining the IoT," *Procedia Computer Science*, vol. 52, pp. 1022–1027, 2015.

[66] M. P. Andersen and D. E. Culler, "BTrDB: Optimizing Storage System Design for Timeseries Processing," in *Proceedings of the 14th Conference on File and Storage Technologies (FAST)*. USENIX, 2016, pp. 39–52.

[67] M. P. Andersen, S. Kumar, C. Brooks, A. von Meier, and D. E. Culler, "DISTIL: Design and Implementation of a Scalable Synchrophasor Data Processing

System," in *Proceedings of the 6th International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2015, pp. 271–277.

[68] M. E. Khalefa, U. Fischer, T. B. Pedersen, and W. Lehner, "Model-based Integration of Past & Future in TimeTravel," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 12, pp. 1974–1977, August 2012.

[69] U. Fischer, D. Kaulakienė, M. E. Khalefa, W. Lehner, T. B. Pedersen, L. Šikšnys, and C. Thomsen, "Real-Time Business Intelligence in the MIRABEL Smart Grid System," in *Revised Selected Papers from the 6th International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*. Springer, 2012, pp. 1–22.

[70] U. Fischer, F. Rosenthal, and W. Lehner, "F^{2}DB: The Flash-Forward Database System," in *Proceedings of the 28th International Conference on Data Engineering (ICDE)*. IEEE, 2012, pp. 1245–1248.

[71] U. Fischer, C. Schildt, C. Hartmann, and W. Lehner, "Forecasting the Data Cube: A Model Configuration Advisor for Multi-Dimensional Data Sets," in *Proceedings of the 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 853–864.

[72] G. Bakkalian, C. Koncilia, and R. Wrembel, "On Representing Interval Measures by Means of Functions," in *Proceedings of the 6th International Conference on Model and Data Engineering (MEDI)*. Springer, 2016, pp. 180–193.

[73] B. Bębel, M. Morzy, T. Morzy, Z. Królikowski, and R. Wrembel, "OLAP-Like Analysis of Time Point-Based Sequential Data," in *Advances in Conceptual Modeling - 2012 ER Workshops CMS, ECDM-NoCoDA, MoDIC, MORE-BI, RIGiM, SeCoGIS, WISM*. Springer, 2012, pp. 153–161.

[74] C. Koncilia, T. Morzy, R. Wrembel, and J. Eder, "Interval OLAP: Analyzing Interval Data," in *Proceedings of the 16th International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*. Springer, 2014, pp. 233–244.

[75] B. Mozafari and N. Niu, "A Handbook for Building an Approximate Query Engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 3, pp. 3–29, September 2015.

[76] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the 21st SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. ACM, 2002, pp. 1–16.

[77] K. S. Perera, M. Hahmann, W. Lehner, T. B. Pedersen, and C. Thomsen, "Efficient Approximate OLAP Querying Over Time Series," in *Proceedings of the 20th International Database Engineering & Applications Symposium (IDEAS)*. ACM, 2016, pp. 205–211.

[78] H. Li, N. Qiu, M. Chen, H. Li, Z. Dai, M. Zhu, and M. Huang, "FASTDB: An Array Database System for Efficient Storing and Analyzing Massive Scientific Data," in *Proceedings of the International Workshops and Symposiums on Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer, 2015, pp. 606–616.

[79] F. Rusu and Y. Cheng, "A Survey on Array Storage, Query Languages, and Systems," *CoRR*, February 2013, arXiv:1302.0103v2.

[80] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-Memory Big Data Management and Processing: A Survey," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 27, no. 7, pp. 1920–1948, July 2015.

[81] "COMPOSE - Collaborative Open Market to Place Objects at your Service," http://www.compose-project.eu/, Viewed: 2019-08-14.

[82] "Micro-Request-Based Aggregation, Forecasting and Scheduling of Energy Demand, Supply and Distribution (MIRABEL)," http://www.mirabel-project.eu/, Viewed: 2019-08-14.

[83] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li *et al.*, "The MADlib Analytics Library," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 12, pp. 1700–1711, August 2012.

[84] X. Xu, S. Huang, Y. Chen, C. Wang, I. Halilovic, K. Brown, and M. Ashworth, "A Demonstration of SearchonTS: An Efficient Pattern Search Framework for Time Series Data," in *Proceedings of the 23rd International Conference on Information and Knowledge Management (CIKM)*. ACM, 2014, pp. 2015–2017.

[85] A. Cuzzocrea, G. Fortino, and O. Rana, "Managing Data and Processes in Cloud-Enabled Large-Scale Sensor Networks: State-of-the-Art and Future Research Directions," in *Proceedings of the 13th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE/ACM, 2013, pp. 583–588.

[86] A. Cuzzocrea, "Temporal Aspects of Big Data Management: State-of-the-Art Analysis and Future Research Directions," in *Proceedings of the 22nd International Symposium on Temporal Representation and Reasoning (TIME)*. IEEE, 2015, pp. 180–185.

[87] J. Ramnarayan, S. Menon, S. Wale, and H. Bhanawat, "Demo: SnappyData: A Hybrid System for Transactions, Analytics, and Streaming," in *Proceedings of the 10th International Conference on Distributed and Event-based Systems (DEBS)*. ACM, 2016, pp. 372–373.

[88] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav, "SnappyData: A Hybrid

Transactional Analytical Store Built On Spark," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2016, pp. 2153–2156.

[89] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-Stores vs. Row-Stores: How Different Are They Really?" in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2008, pp. 967–980.

[90] T. G. Papaioannou, M. Riahi, and K. Aberer, "Towards Online Multi-Model Approximation of Time Series," in *Proceedings of the 12th International Conference on Mobile Data Management (MDM)*, vol. 1. IEEE, 2011, pp. 33–38.

[91] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm, "A time-series compression technique and its application to the smart grid," *The VLDB Journal (VLDBJ)*, vol. 24, no. 2, pp. 193–218, April 2015.

[92] A. Kumar, R. McCann, J. Naughton, and J. M. Patel, "Model Selection Management Systems: The Next Frontier of Advanced Analytics," *ACM SIGMOD Record*, vol. 44, no. 4, pp. 17–22, December 2015.

[93] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai, "Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams," *Distributed and Parallel Databases*, vol. 18, no. 2, pp. 173–197, September 2005.

[94] T. Johnson and V. Shkapenyuk, "Data Stream Warehousing In Tidalrace," in *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR)*, 2015.

[95] "DiCyPS - Center for Data-Intensive Cyber-Physical Systems," http://www.dicyps.dk/dicyps-in-english/, Viewed: 2019-08-14.

Paper B

ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra

Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen

The paper has been published in the
Proceedings of the VLDB Endowment (PVLDB), Volume 11, Number 11,
Pages 1688–1701, July, 2018, DOI: 10.14778/3236187.3236215

Abstract

Industrial systems, e.g., wind turbines, generate big amounts of data from reliable sensors with high velocity. As it is unfeasible to store and query such big amounts of data, only simple aggregates are currently stored. However, aggregates remove fluctuations and outliers that can reveal underlying problems and limit the knowledge to be gained from historical data. As a remedy, we present the distributed TSMS ModelarDB *that uses* models *to store sensor data. We thus propose an online, adaptive multi-model compression algorithm that maintains data values within a user-defined error bound (possibly zero). We also propose (i) a database schema to store time series as models, (ii) methods to push-down predicates to a key-value store utilizing this schema, (iii) optimized methods to execute aggregate queries on models, (iv) a method to optimize execution of projections through static code-generation, and (v) dynamic extensibility that allows new models to be used without recompiling the TSMS. Further, we present a general modular distributed TSMS architecture and its implementation, ModelarDB, as a portable library, using Apache Spark for query processing and Apache Cassandra for storage. An experimental evaluation shows that, unlike current systems, ModelarDB hits a sweet spot and offers fast ingestion, good compression, and fast, scalable online aggregate query processing at the same time. This is achieved by dynamically adapting to data sets using multiple models. The system degrades gracefully as more outliers occur and the actual errors are much lower than the bounds.*

The layout has been revised.

1 Introduction

For critical infrastructure, e.g., renewable energy sources, large numbers of high quality sensors with wired electricity and connectivity provide data to monitoring systems. The sensors are sampled at regular intervals and while invalid, missing, or out-of-order data points can occur, they are rare and all but missing data points can be corrected by established cleaning procedures. Although practitioners in the field require high-frequent historical data for analysis, it is currently impossible to store the huge amounts of data points. As a workaround, simple aggregates are stored at the cost of removing outliers and fluctuations in the time series.

In this paper, we focus on how to store and query massive amounts of high quality sensor data ingested in real-time from many sensors. To remedy the problem of aggregates removing fluctuations and outliers, we propose that high quality sensor data is compressed using *model-based* compression. We use the term *model* for any representation of a time series from which the original time series can be recreated within a *known error bound* (possibly zero). For example, the linear function $y = ax + b$ can represent an increasing, decreasing, or constant time series and reduces storage requirements from one value per data point to only two values: a and b. We support both lossy and lossless compression. Our lossy compression preserves the data's structure and outliers and the user-defined error bound allows a trade-off between accuracy and required storage. This contrasts traditional sensor data management where models are used to infer data points with less noise [1].

To establish the state-of-the-art used in industry for storing time series, we evaluate the storage requirements of commonly used systems and big data file formats. We

Table B.1: Comparison of common storage solutions

Storage Method	**Size (GiB)**
CSV Files	582.68
PostgreSQL 10.1	782.87
RDBMS-X – Row	367.89
RDBMS-X – Column	166.83
Apache Cassandra 3.9	111.89
Apache Parquet Files	106.94
Apache ORC Files	13.50
InfluxDB 1.4.2 – Tags	4.33
InfluxDB 1.4.2 – Measurements	4.33
ModelarDB	2.41–2.84

select the systems based on DB-Engines Ranking [2], discussions with companies in the energy sector, and our survey [3]. Two RDBMSs and a TSMS are included due to their widespread industrial use, although being optimized for smaller data sets than the distributed solutions. The big data file formats, on the other hand, handle big data sets well, but do not support streaming ingestion for online analytics. We use sensor data with a 100ms sampling interval from an energy production company. The schema and data set (Energy Production High Frequency), are described in Section 7. The results, in Table B.1 including our system *ModelarDB*, show the benefit of using a TSMS or columnar storage for time series. However, the storage reduction achieved by ModelarDB is much more significant, even with a 0% error bound, and clearly demonstrates the advantage of model-based storage for time series.

To efficiently manage high quality sensor data, we find the following properties paramount for a TSMS: (i) *Distribution:* Due to huge amounts of sensor data, a distributed architecture is needed. (ii) *Stream Processing:* For monitoring, ingested data points must be queryable after a small user-defined time lag. (iii) *Compression:* Fine-grained historical values can reveal changes over time, e.g., performance degradation. However, raw data is infeasible to store without compression, which also helps query performance due to reduced disk I/O. (iv) *Efficient Retrieval:* To reduce processing time for querying a subset of the historical data, indexing, partitioning and/or time-ordered storage is needed. (v) *AQP:* Approximation of query results within a user-defined error bound can reduce query response time and enable lossy compression. (vi) *Extensibility:* Domain experts should be able to add domain-specific models without changing the TSMS, and the system should automatically use the best model.

While methods for compressing segments of a time series using one of multiple models exist [4–6], we found no TSMS using multi-model compression for our survey [3]. Also, the existing methods do not provide all the properties listed above. They either provide no latency guarantees [4, 6], require a trade-off between latency and compression [5], or limit the supported model types [4, 6]. ModelarDB, in contrast, provides all these features, and we make the following contributions to model-based storage and query processing for big data systems:

- A general-purpose architecture for a modular model-based TSMS providing the listed paramount properties.
- An efficient and adaptive algorithm for online multi-model compression of time series within a user-defined error bound. The algorithm is model-agnostic, extensible, combines lossy and lossless compression, allows missing data points, and offers both low latency and high compression ratio at the same time.
- Methods and optimizations for a model-based TSMS:
  - A database schema to store multiple time series as models.
  - Methods to push-down predicates to a key-value store used as a model-based physical storage layer.
  - Methods to execute optimized aggregate functions directly on models without requiring a dynamic optimizer.

  - Use of static code-generation to optimize projections.
  - Dynamic extensibility making it possible to add additional models without changing or recompiling the TSMS.
- Realization of our architecture as the distributed TSMS ModelarDB, consisting of the portable ModelarDB Core interfaced with unmodified versions of Apache Spark for query processing and Apache Cassandra for storage.
- An evaluation of ModelarDB using time series from the energy domain. The evaluation shows how the individual features and contributions effectively work together to dynamically adapt to the data sets using multiple models, yielding a unique combination of good compression, fast ingestion, and fast, scalable online aggregate query processing. The actual errors are shown to be much lower than the allowed bounds and ModelarDB degrades gracefully when more outliers are added.

The paper is organized as follows. Definitions are given in Section 2. Section 3 describes our architecture. Section 4 details ingestion and our model-based compression algorithm. Section 5 describes query processing and Section 6 ModelarDB's distributed storage. Section 7 presents an evaluation, Section 8 related work, and Section 9 conclusion and future work.

2 Preliminaries

We now provide definitions which will be used throughout the paper. We also exemplify each using a running example.

Definition B.1 (Time Series)
A *time series* TS is a sequence of *data points*, in the form of time stamp and value pairs, ordered by time in increasing order $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$. For each pair (t_i, v_i), $1 \leq i$, the time stamp t_i represents the time when the value $v_i \in \mathbb{R}$ was recorded. A time series $TS = \langle (t_1, v_1), \ldots, (t_n, v_n) \rangle$ consisting of a fixed number of n data points is a *bounded time series*.

As a running example we use the time series $TS = \langle (100, 28.3), (200, 30.7), (300, 28.3), (400, 28.3), (500, 15.2), \ldots \rangle$, each pair represents a time stamp in milliseconds since the recording of measurements was initiated and a recorded value. A bounded time series can be constructed, e.g, from the subset of data points of TS where $t_i \leq 300$, $1 \leq i$.

Definition B.2 (Regular Time Series)
A time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ is considered *regular* if the time elapsed between each data point is always the same, i.e., $t_{i+1} - t_i = t_{i+2} - t_{i+1}$ for $1 \leq i$ and *irregular* otherwise.

Our example time series TS is a regular time series as 100 milliseconds elapse between each of its adjacent data points.

Definition B.3 (Sampling Interval)
The *sampling interval* of a regular time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ is the time elapsed between each pair of data points in the time series $SI = t_{i+1} - t_i$ for $1 \leq i$.

As 100 milliseconds elapse between each pair of data points in TS, it has a sampling interval of 100 milliseconds.

Definition B.4 (Model)
A *model* is a representation of a time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ using a pair of functions $M = (m_{est}, m_{err})$. For each t_i, $1 \leq i$, the function m_{est} is a real-valued mapping from t_i to an estimate of the value for the corresponding data point TS. m_{err} is a mapping from a time series TS and the corresponding m_{est} to a positive real value representing the error of the values estimated by m_{est}.

For the bounded subset of TS a model M can be created using, e.g., a linear function with $m_{est} = -0.0024t_i + 29.5$, $1 \leq i \leq 5$, and an error function using the uniform error norm so $m_{err} = max(|v_i - m_{est}(t_i)|)$, $1 \leq i \leq 5$. This model-based representation of TS has an error of $|15.2 - (-0.0024 \times 500 + 29.5)| = 13.1$ caused by the data point at t_5. The difference between the estimated and recorded values would be much smaller without this data point.

Definition B.5 (Gap)
A *gap* between a regular bounded time series $TS_1 = \langle (t_1, v_1), \ldots, (t_s, v_s) \rangle$ and a regular time series $TS_2 = \langle (t_e, v_e), (t_{e+1}, v_{e+1}), \ldots \rangle$ with the same sampling interval SI and recorded from the same source, is a pair of time stamps $G = (t_s, t_e)$ with $t_e = t_s + m \times SI$, $m \in \mathbb{N}_{\geq 2}$, and where no data points exist between t_s and t_e.

The concept of a gap is illustrated in Figure B.1. For simplicity we will refer to multiple time series from the same source separated by gaps as a single time series containing gaps.

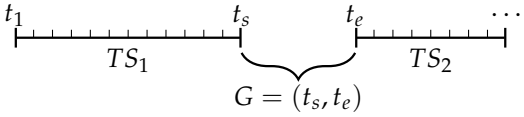

Figure B.1: Illustration of a gap G between t_s and t_e

TS does not contain any gaps. However, the time series $TS_g = \langle (100, 28.3), (200, 30.7), (300, 28.3), (400, 28.3), (500, 15.2), (800, 30.2), \ldots \rangle$ does. While the only difference between TS and TS_g is the data point $(800, 30.2)$ a gap is now present as no data points exist with the time stamps 600 and 700. Due to the gap, TS_g is an irregular time series, while TS is a regular time series.

Definition B.6 (Regular Time Series with Gaps)
A *regular time series with gaps* is a regular time series, $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ where $v_i \in \mathbb{R} \cup \{\perp\}$ for $1 \leq i$. For a regular time series with gaps, a gap $G = (t_s, t_e)$ is a sub-sequence where $v_i = \perp$ for $t_s < t_i < t_e$.

The irregular time series $TS_g = \langle(100, 28.3), (200, 30.7), (300, 28.3), (400, 28.3), (500, 15.2), (800, 30.2), \ldots\rangle$ with an undefined SI due to the presence of a gap, can be represented as the regular time series with gaps $TS_{gr} = \langle(100, 28.3), (200, 30.7), (300, 28.3), (400, 28.3), (500, 15.2), (600, \perp), (700, \perp), (800, 30.2), \ldots\rangle$ with $SI = 100$ milliseconds.

Definition B.7 (Segment)
For a bounded regular time series with gaps $TS = \langle(t_s, v_s), \ldots, (t_e, v_e)\rangle$ with sampling interval SI, a *segment* is a 6-tuple $S = (t_s, t_e, SI, G_{ts}, M, \epsilon)$ where G_{ts} is a set of timestamps for which $v = \perp$ and where the values of all other timestamps for TS are defined by the model M within the error bound ϵ.

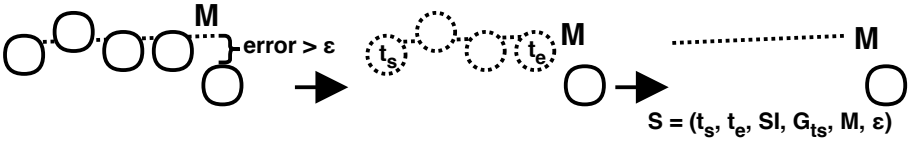

Figure B.2: Model-based representation of data points

In the example for Definition B.4, the model M represents the data point at t_5 with an error that is much larger than for the other data points of TS. For this example we assume the user-defined error bound to be 2.5 which is smaller than the error of M at 13.1. To uphold the error bound a segment $S = (100, 400, 100, \emptyset, (m_{est} = -0.0024t_i + 29.5, m_{err} = max(|v_i - m_{est}(t_i)|)), 2.5), 1 \leq i \leq 4$, can be created. As illustrated in Figure B.2, segment S contains the first four data points of TS represented by the model M within the user-defined error bound of 2.5 as $|30.7 - (-0.0024 \times 200 + 29.5)| = 1.68 \leq 2.5$. As additional data points are added to the time series, new segments can be created to represent each sub-sequence of data points within the user-defined error bound.

In this paper we focus on the use case of multiple regular times with gaps being ingested and analyzed in a central TSMS.

3 Architecture

The architecture of ModelarDB is modular to enable re-use of existing software deployed in a cluster, and is split into three sets of components with well-defined purposes: data ingestion, query processing and segment storage. ModelarDB is designed around ModelarDB Core, a portable library with system-agnostic functionality for model-based time series management, caching of metadata and a set of predefined models. For distributed query processing and storage ModelarDB Core integrates with existing systems through a set of interfaces, making the portable core simple to use with existing infrastructure. The architecture is shown in Figure B.3. Data flow between components is shown as arrows, while the sets of components are separated by dashed lines.

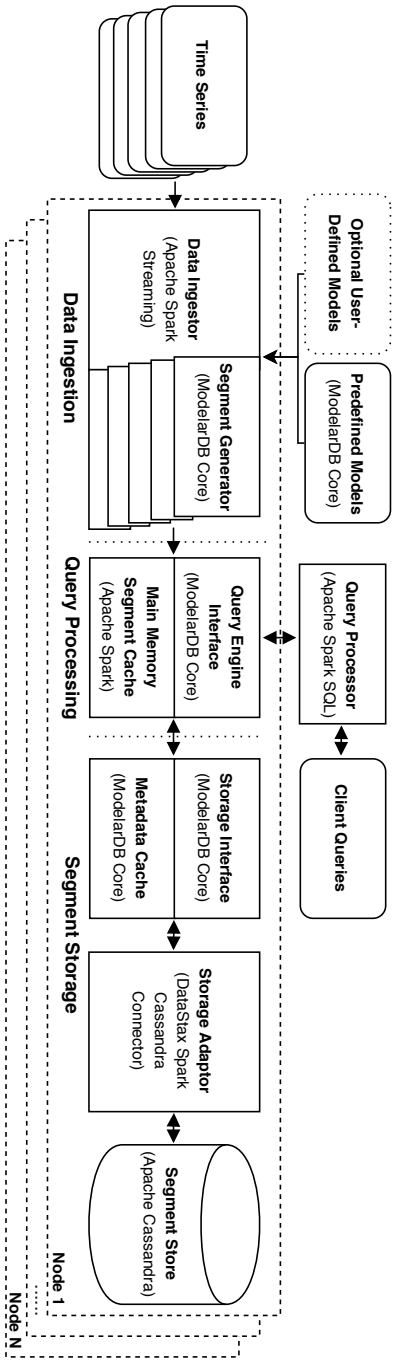

Figure B.3: Architecture of a ModelarDB node, each contains query processing and storage to improve locality

Our implementation of the architecture integrates ModelarDB Core with the stock versions of Apache Spark [7–11] for distributed query processing, while Apache Cassandra [12, 13] or a JDBC compatible RDBMS can be used for storage. As distributed storage is a paramount property of ModelarDB we focus on Cassandra in the rest of the paper. Each component in Figure B.3 is annotated with the system or library providing the functionality of that component in ModelarDB. Spark and Cassandra are used due to both being mature components of the Hadoop ecosystem and well integrated through the DataStax Spark Cassandra Connector. To allow unmodified instances of Spark and Cassandra already deployed in a cluster to be used with ModelarDB, it is implemented as a separate `JAR` file that embeds ModelarDB Core and only utilizes the public interfaces provided by Spark and Cassandra. As a result, ModelarDB can be deployed by submitting the `JAR` file as job to an unmodified version of Spark. In addition, a single-node ingestor has been implemented to support ingestion of data points without Spark. Support for other query processing systems, e.g. Apache Flink [14, 15], can be added to ModelarDB by implementing a new engine class. Support for other storage systems, e.g. Apache HBase [16] or MongoDB [17], simply requires that a storage interface provided by ModelarDB Core be implemented.

When ingesting, ModelarDB partitions the time series and assigns each subset to a core in the cluster. Thus, data points are ingested from the subsets in parallel and time series within each subset are ingested concurrently for unbounded time series. The ingested data points are converted to a model-based representation using an appropriate model automatically selected for each dynamically sized segment of the time series. In addition to the predefined models, user-defined models can be dynamically added without changes to ModelarDB. Segments constructed by the segment generators are kept in memory as part of a distributed in-memory cache. As new segments are emitted to the stream, batches of old segments are flushed to the segment store. Both segments in the cache and in the store can be queried using SQL. By keeping the most recent set of segments in memory, efficient queries can be performed on the most recent data. Queries on historical data have increased query processing time as the segments must be retrieved from Cassandra and cached. Subsequent queries will be performed on the cache.

By reusing existing systems, functionality for fault tolerance can be reused as demonstrated in [18]. As a result, ModelarDB can provide mature and well-tested fault-tolerance guarantees and allows users of the system to select a system for storage and a query processing engine with trade-offs appropriate for a given use case. However, as the level of fault tolerance of ModelarDB depends on the query engine and data store, we only discuss it at the architectural level for our implementation. For ModelarDB data loss can occur at three stages: data points being ingested, segments in the distributed memory cache, and segments written to disk. Fault tolerance can be guaranteed for data points being ingested by using a reliable data source such as Apache Kafka, or by ingesting from each time series at multiple nodes. Fault tolerance for segments in memory and on disk can be ensured through use of distributed replication. In our implementation, loss of segments can be prevented by compiling ModelarDB with

replication enabled for Spark and Cassandra. In the rest of this paper we do not consider replication, since ModelarDB reuses the replication in Spark and Cassandra without any modification. As each data point is ingested by one node, data points will be lost if a node fails. However, as our main use case is analyzing tendencies in time series data, some data loss can be acceptable to significantly increase ingestion rate [19].

In addition to fault tolerance, by utilizing existing components the implementation of ModelarDB can be kept small, reducing the burden of ensuring correctness and adding new features. ModelarDB is implemented in 1675 lines of Java code for ModelarDB Core and 1429 lines of Scala code for the command-line and the interfaces to existing systems. ModelarDB Core is implemented in Java to make it simple to interface with the other JVM languages and to keep the translation from source to bytecode as simple as possible when optimizing performance. Scala was used for the other components due to increased productivity from pattern matching, type inference, and immutable data structures. The source code is available at `https://github.com/skejserjensen/ModelarDB`.

4 Data Ingestion

To use the resources evenly, ingestion is performed in parallel based on the number of threads available for that task and the sampling rate of the time series. The set of time series is partitioned into disjoint subsets SS and assigned to the available threads so the data points per second of each subset are as close to equal as possible. Providing each thread with the same amount of data points to process, ensures resources are utilized uniformly across the cluster to prevent bottlenecks. The partitioning method used by ModelarDB is based on [20], and minimizes $max(data\_points\_per\_minute(S_1)) - min(data\_points\_per\_minute(S_2))$ for $S_1, S_2 \in SS$.

4.1 Model-Agnostic Compression Algorithm

To make it possible to extend the set of models provided by ModelarDB Core, we propose an algorithm for segmenting and compressing regular time series with gaps in which models using lossy or lossless compression can be used. By optimizing the algorithm for regular time series with gaps as per our use case described in Section 1, the timestamp of each data point can be discarded as they can be reconstructed using the sampling interval stored for each time series and the start time end time stored as part of each segment. To alleviate the trade-off between high compression and low latency required by existing multi-model compression algorithms, we introduce two segment types, namely a *temporary segment (ST)* and a *finalized segment (SF)*. The algorithm emits STs based on a user-defined maximum latency in terms of data points not yet emitted to the stream, while SFs are emitted when a new data point cannot be represented by the set of models used. The general idea of our algorithm is shown in Figure B.4 and uses a list of models from which one model is active at a time as

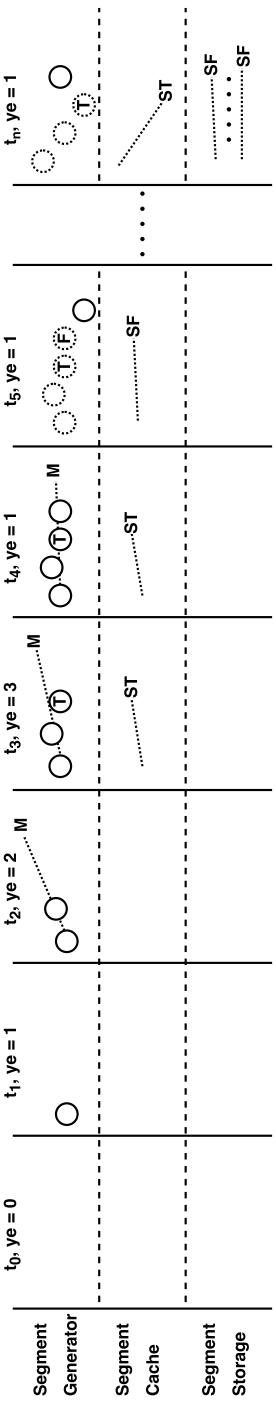

Figure B.4: The multi-model compression algorithm, maximum for yet to be emitted (ye) data points is three

proposed by [4]. For this example we set the maximum latency to be three data points, use a single model in the form of a linear function, and ingest the time series TS from Section 2. At t_1 and t_2, data points are added to a temporary buffer while a model M is incrementally fitted to the data points. As our method is model-agnostic, each model defines how it is fitted to the data points and how the error is computed. This allows models to implement the most appropriate method for fitting data points, e.g., models designed for streaming can fit one data point a time, while models that must be recomputed for each data point can perform chunking. At t_3, three data points have yet to be emitted, see ye, and the model is emitted to the main memory segment cache as a part of a ST. For illustration, we mark the last data point emitted as part of a ST with a T, and the last data points emitted as part of a SF with an F. As M might be able to represent more data points, the data points are kept in the buffer and the next data point is added at t_4. At t_5, a data point is added which M cannot represent within the user-defined error bound. As our example only includes one model, a SF is emitted to the main memory segment cache and the data points represented by the SF deleted from the buffer as shown by dotted circles, before the algorithm starts anew with the next data point. As the SF emitted represents the data point ingested at t_4, ye is not incremented at t_5 to not emit data points already represented by a SF as a part of a ST. Last, at t_n, when the cache reaches a user-defined *bulk write size*, the segments are flushed to disk.

Our compression algorithm is shown in Algorithm B.1. First variables are initialized in Line 8-11, this corresponds to t_0 in Figure B.4. To ensure the data points can be reproduced from each segment, in Line 14-16, if a gap exists all data points in the *buffer* are emitted as one or more SFs. If the number of data points in the buffer is lower than what is required to instantiate any of the provided *models* (a linear function requires two data points) a segment containing uncompressed values is emitted. In Line 17-20 the data point is appended to the buffer, and *previous* is set to the current data point. The data point is appended to *model*, allowing the model to update its internal parameters and the algorithm to check if the model can represent the new data point within the user-defined error bound or length limit. Afterwards, the number of data points not emitted as a segment is incremented. This incremental process is illustrated by states t_1 to t_4 in Figure B.4. If *latency* data points have not been emitted, a ST using the current model is emitted in Line 21-23. The current model is kept as it might represent additional data points. This corresponds to t_3 in Figure B.4 as a ST is emitted due to $latency = 3$. If the current model cannot be instantiated with the data points in the buffer, a ST containing uncompressed values is emitted. When *model* no longer can represent a data point within the required error bound, the next model in the list of *models* is selected and initialized with the buffered data points in Line 25-27. As the *model* represents as many data points from *buffer* as possible when initialized, and any subsequent data points are rejected, no explicit check of if the model can represent all data points in the buffer is needed. Instead, this check will be done as part of the algorithm's next iteration when a new data point is appended. When the list of *models* becomes empty, a SF containing the model with the

Algorithm B.1 Online model-agnostic (lossy and lossless) multi-model compression algorithm with latency guarantees

```
1:  Let ts be the time series of data points.
2:  Let models be the list of models to select from.
3:  Let error be the user defined error bound.
4:  Let limit be the limit on the length of each segment.
5:  Let latency be the latency in not emitted data points.
6:  Let interval be the sampling interval of the time series.
7:
8:  model ← head(models)
9:  buffer ← create_list()
10: yet_emitted ← 0
11: previous ← nil
12: while has_next(ts) do
13:     data_point = retrieve_next_data_point(ts)
14:     if time_diff(previous, data_point) > interval then
15:         flush_buffer(buffer)
16:     end if
17:     append_data_point_to_buffer(data_point, buffer)
18:     previous ← data_point
19:     if append_data_point_to_model(data_point, model, error, limit) then
20:         yet_emitted ← yet_emitted + 1
21:         if yet_emitted = latency then
22:             emit_temporary_segment(model, buffer)
23:             yet_emitted ← 0
24:         end if
25:     else if has_next(models) then
26:         model ← next(models)
27:         initialize(model, buffer)
28:     else
29:         emit_finalized_segment(models, buffer)
30:         model ← head(models)
31:         initialize(model, buffer)
32:         yet_emitted ← min(yet_emitted, lengh(buffer))
33:     end if
34: end while
35: flush_buffer(buffer)
```

highest compression ratio is emitted in Line 29. To allow for models using lossless compression we compute the compression ratio as the reduction in bytes not the number of values to be stored: $compression\_ratio = (data\_points\_represented(model) \times$

size_of(*data_point*))/*size_of*(*model*). As model selection is based on the compression ratio, the segment emitted by *emit_finalized_segment* might not represent all data points in the buffer. In Line 30-32 *model* is set to the first model in the list and initialized with any data points left in the buffer. If data points not emitted by a ST were emitted as part of the SF, *yet_emitted* is decremented appropriately. This process of emitting a SF corresponds to t_5 of Figure B.4, where the latest data point is left in the buffer and one data point is emitted first by a SF. In Line 35 as all data points have been received, the buffer is flushed so all data points are emitted as SFs.

4.2 Considerations Regarding Data Ingestion

Two methods exist for segmenting time series: connected and disconnected. A connected segment starts with the previous segment's last data point, while a disconnected segment starts with the first data point not represented by the previous segment. Our algorithm supports both by changing if *emit_finalized_segment* keeps the last data point of a segment when it is emitted. The use of connected segments provides two benefits. If used with models supporting interpolation, the time series can be reconstructed with any sampling interval as values between any two data points can be interpolated. Also, connected segments can be stored using only a single time stamp as the end time of one segment is the start time of the next. However, for multi-model compression of time series [5, 21] demonstrated an increased compression ratio for disconnected segments if the start and end time of each segment are stored for use with indexing. The decreased size is due to the increased flexibility when fitting disconnected segments as no data point from the previous segment is included [5, 21]. Since time series may have gaps, the start and end time of a segment must be stored to ensure all data points ingested can be reconstructed. As a result, the rest of this paper will only be concerned with disconnected segments.

To represent gaps, there are two methods: flushing the stream of data points when a gap is encountered, or storing the gaps explicitly as a pair of time stamps $G = (t_s, t_e)$. When we evaluated both we observed no significant difference in compression ratio. However, storing gaps explicitly requires additional computation as any operation on segments must skip gaps which also complicates the implementation. Storing gaps explicitly requires *flush_buffer*(*buffer*) in Line 15 in Algorithm B.1 be substituted with *timestamp*(*previous*) and *timestamp*(*data_point*) being added to a gap buffer and that the functions emitting segments are modified to include gaps as part of the segment. ModelarDB flushes the stream of data points as shown in Algorithm B.1; but explicit storage of gaps can be enabled.

4.3 Implementation of User-Defined Models

For a user to optionally add a new model and segment in addition to those predefined in ModelarDB Core, each must implement the interfaces in Table B.2. `Tid` is a unique id assigned to each time series. By having a segment class, a model object can store

Table B.2: Interface for models and segments, ● is a required method and ○ is an optional method

Model		
`new(Error, Limit)`	●	Return a new model with the user-defined error bound and length limit.
`append(Data Point)`	●	Append a data point if it and all previous do not exceed the error bound.
`initialize([Data Point])`	●	Clear the existing data points from the model and append the data points from the list until one exceeds the error bound or length limit.
`get(Tid, Start Time, End Time, SI, Parameters, Gaps)`	●	Create a segment represented by the model from serialized parameters.
`get(Tid, Start Time, End Time, SI, [Data Point], [Gap])`	●	Create a segment from the models state and the list of data points.
`length()`	●	Return the number of data points the model currently represents.
`size()`	●	Return the size in bytes currently required for the models parameters.

Segment		
`get(Timestamp, Index)`	●	Return the value from the underlying model that matches the timestamp and index, both are provided to simplify implementation of this interface.
`parameters()`	●	Return the segment specific parameters necessary to reconstruct it.
`sum()`	○	Compute the sum of the values of data points represented by the segment
`min()`	○	Compute the minimum value of data points represented by the segment.
`max()`	○	Compute the maximum value of data points represented by the segment.

data while ingesting data points without increasing the size of its segment. As a result, models can implement model-specific optimizations such as chunking, lazy fitting or memoization. For aggregate queries to be executed directly on a segment, the optional methods must be implemented. An implementation of `sum` for a segment using a linear function as the model is shown in Listing B.1. For this model, the sum can be computed without recreating the data points by multiplying the average of the values with the number of represented data points. In Line 2–3 the number of data points is computed. In Line 4–5 the minimum and maximum value of the segment. Last, in Line 6-7 the sum is computed by multiplying the average with the number of data points. As a result, the sum can be computed in ten arithmetic operations and without a loop.

```
1 public double sum() {
2   int timespan = this.endTime - this.startTime;
3   int size = (timespan / this.SI) + 1;
4   double first = this.a * this.startTime + this.b;
5   double last = this.a * this.endTime + this.b;
6   double average = (first + last) / 2;
7   return average * size;
8 }
```

Listing B.1: `sum` implemented for a linear model

To demonstrate we use the segment from Section 2 with the start time 100, the end time 400, the sampling interval 100, and the linear function as $-0.0024t_i + 29.5$ as model. For a more realistic example we increase the end time to 7300. First the number of data points represented by the segment is calculated $((7300 - 100)/100) + 1 = 73$, followed by the value of the first $-0.0024 \times 100 + 29.5 = 29.26$, and last data point $-0.0024 \times 7300 + 29.5 = 11.98$. The average value for the data points represented by the segment is then $(29.26 + 11.98)/2 = 20.62$, with the sum of the represented values given by $20.62 \times 73 = 1505.26$. Our example clearly shows the benefit of using models for queries as computing the sum is reduced from 73 arithmetic operations to 10, or in terms of complexity, from linear to constant time complexity.

All models must exhibit the following behavior. A model yet to append enough data points to instantiate the model must return an invalid compression ratio `NaN` so it is not selected to be part of a segment. Second, if a model rejects a data point, all following data points must be rejected until the model is reinitialized. Last, as consequence of using an extensible set of models, the method for computing the error of a model's approximation must be defined by the model. The combination of user-defined models and the model selection algorithm provides a framework expressive enough to express existing approaches for time series compression. For TSMSs that compress time series as statically sized sub-sequences using one compression method, such as Facebook's Gorilla [19], a single model which rejects data points based on the limit parameter can be used. For methods that use multiple lossy models in a predefined sequence, such as [4], the same models can be implemented and re-used with any system that integrates ModelarDB Core, with the added benefit that the ordering of the models are not hardcoded as part of the algorithm as in [4] but simply a parameter.

For evaluation we implement a set of general-purpose models from the literature.

We base our selection of models on [22] demonstrating substantial increases in compression ratio for models supporting dynamically sized segments and high compression ratio for some constant and linear models, in addition to existing multi-model approaches predominately selecting constant and linear models [4, 21]. Also we select models that can be fitted incrementally to efficiently fit the data points online. To ensure the user-provided error bound is guaranteed for each data point, only models providing an error bound based on the uniform error norm are considered [23]. Last, we select models with lossless and lossy compression, allowing ModelarDB to select the approach most appropriate for each sub-sequence. We thus implement the following models: the constant PMC-MR model [24], the linear Swing model [25], both modified so the error bound can be expressed as the percentage difference between the real and approximated value, and the lossless compression algorithm for floating-point values proposed by Facebook [19] modified to use `floats`. A model storing raw values is used by ModelarDB when no other model is applicable.

5 Query Processing

5.1 Generic Query Interface

Segments emitted by the segment generators are put in the main-memory cache and made available for querying together with segments in storage. ModelarDB provides a unified query interface for segments in memory and storage using two views. The first view represents segments directly while the second view represents segments as data points. This *segment view* uses the schema `(Tid int, StartTime timestamp, EndTime timestamp, SI int, Mid int, Parameters blob)` and allows for aggregate queries to be executed on the segments without reconstructing the data points. The attribute `Tid` is the unique time series id, `SI` the sampling interval, `Mid` the id of the model used for the segment, and `Parameters` the parameters for the model. The *data point view* uses the schema `(Tid int, TS timestamp, Value float)` to enable queries to be executed on data points reconstructed from the segments. Implementing views at these two levels of abstraction allows query processing engines to directly interface with the data types utilized by ModelarDB Core. Efficient aggregates can be implemented as user-defined aggregate functions (UDAFs) on the segment view, and predicate push-down can be implemented to the degree that the query processing engine supports it. While only having the data point view would provide a simpler query interface, interfacing a new query processing engine with ModelarDB Core would be more complex as aggregate queries to the data point view should be rewritten to use the segment view [1, 26, 27].

5.2 Query Interface in Apache Spark

Through Spark SQL ModelarDB provides SQL as its query language. Since Spark SQL only pushes the required columns and the predicates of the `WHERE` clause to

the data source, aggregates are implemented as UDAFs on the segment view. While full query rewriting is not performed, the data point view retrieves segments through the segment view which pushes the predicates of the `WHERE` clause to the segment store. As a result, the segment store needs only support predicate push-down from the segment view, and never from both views. Our current implementation supports `COUNT`, `MIN`, `MAX`, `SUM`, and `AVG`. The UDAFs use the optional methods from the segment interface shown in Table B.2 if available, otherwise the query is executed on data points. As UDAFs in Spark SQL cannot be overloaded, two sets of UDAFs are implemented. The first set operate on segments as rows and have the suffix `_S`. The second set operate on segments as structs and have the suffix `_SS`. Queries on segments can be filtered at the segment level using a `WHERE` clause. Thus, for queries on the segment view to be executed with the same granularity as queries on the data point view, functions are provided to restrict either the start time (`START`), end time (`END`), or both (`INTERVAL`) of segments. While ModelarDB at the moment only supports a limited number of built-in aggregate functions through the segment view, to demonstrate the benefit of computing aggregates using models, any aggregate functions provided as part of Spark SQL can be utilized through the data point view. In addition, existing software developed to operate on time series as data points, e.g., for time series similarity search, can utilize the data point view. Last, using the APIs provided by Spark SQL any distributive or algebraic aggregation function can be added to both the data point view and the segment view.

5.3 Execution of Queries on Views

Examples of queries on the views are shown in Listing B.2. Line 1-2 show two queries calculating the sum of all values ingested for the time series with `Tid = 3`. The first computes the result from data points reconstructed from the segments while the second query calculates the result directly from the segments. The query on Line 4-5 computes the averages of the values for data points with a timestamp after `2012-01-03 12:30`. The `WHERE` clause filters the result at the segment level and `START` disregards the data that is older than the timestamp provided. Last, in Line 7-8 a query is executed on the data point view as if the data points were stored.

```
1 SELECT SUM(Value) FROM DataPoint WHERE Tid = 3
2 SELECT SUM_S(*) FROM Segment WHERE Tid = 3
3
4 SELECT AVG_SS( START(*, '2012-01-03 12:30') )
5 FROM Segment WHERE EndTime >= '2012-01-03 12:30'
6
7 SELECT * FROM DataPoint WHERE Tid = 3
8 AND TS < '2012-04-22 12:25'
```

Listing B.2: Query examples supported in ModelarDB

To lower query latency, the main memory segment cache, see Figure B.3, stores the most recently emitted or queried SFs and the last ST emitted for each time series. Then, to ensure queries do not return duplicate data points, the start time of a ST is updated

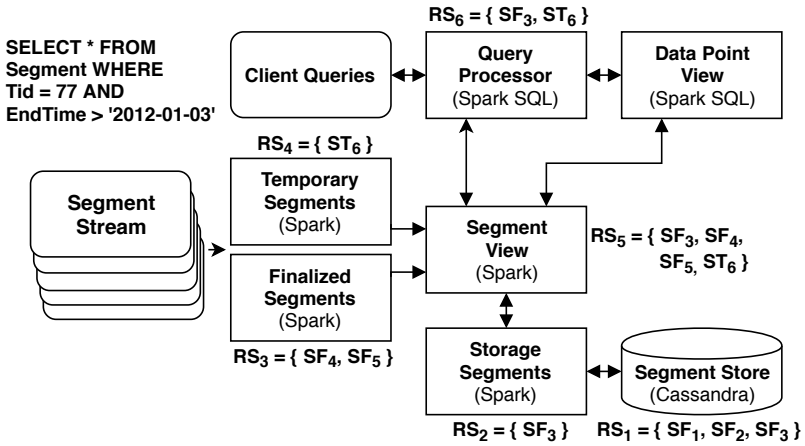

Figure B.5: Query processing in ModelarDB

when a SF with the same `Tid` is emitted so the time intervals do not overlap. STs where `StartTime > EndTime` are dropped. Last, the SF cache is flushed when it reaches a user-defined *bulk write size*.

Query processing in ModelarDB is primarily concerned with filtering out segments and data points from the query result. An example is shown in Figure B.5 with a query for segments that contain data points from the sensor with `Tid = 77` from after the date `2012-01-03`. Assume that only SF_3 and ST_6 satisfy both predicates and that SF_3 is not in the cache. First, the `WHERE` clause predicates are pushed to the segment store, see RS_1, to retrieve the relevant segment. The segment retrieved from disk is cached, see RS_2, and the cache is then unioned with the STs and SFs in memory, shown as RS_3 and RS_4, to produce the set RS_5. RS_5 is filtered according to the `WHERE` clause to possibly remove segments provided by a segment store with imprecise evaluation of the predicates (i.e., with false positives) and to remove irrelevant segments from the in-memory cache. The final result is shown as RS_6. Queries on the data point view are processed by retrieving relevant segments through the segment view. The data points are then reconstructed and filtered based on the `WHERE` clause.

5.4 Code-Generation for Projections

In addition to predicate push-down, the views perform projections such that only the columns used in the query are provided. However, building rows dynamically based on the columns requested creates a significant performance overhead. As the columns of each view are static, optimized projection methods can be generated at compile time without the additional overhead and complexity of dynamic code generation.

The method generated for the data point view is shown in Listing B.3. On Line 3, the list of requested columns is converted to column indexes and concatenated in the

```
1  def getDataPointGridFunction
2    (columns: Array[String]): (DataPoint => Row) = {
3    val target = getTarget(columns, dataPointView)
4    (target: @switch) match {
5      //Permutations of ('tid')
6      case 1 => (dp: DataPoint) => Row(dp.tid)
7      ...
8      //Permutations of ('tid', 'ts', 'value')
9      ...
10     case 321 => (dp: DataPoint) => Row(dp.value,
11       new Timestamp(dp.timestamp), dp.tid)
12   }
13 }
```

Listing B.3: Selection of method for a projection

requested order to create a unique integer. This works for both views as each has less than ten columns and allows the projection method to be retrieved using a `switch` instead of comparing the contents of arrays. On Line 4, the projection method is retrieved using a `match` statement which is compiled to an efficient `lookupswitch` [28].

6 Segment Storage

Figure B.6 shows a generic schema for storing segments with the metadata necessary for ModelarDB. It has three tables: `Time Series` for storing metadata about time series (the current implementation requires only the sampling interval), `Model` for storing the model type contained in each segment, and last `Segment` for storing segments with the model parameters as blobs. The bulk of the data is stored in the segment table. Compared to other work [4, 21, 29], the inclusion of both a `Tid` and the `Time Series` table allows queries for segments from different time series with different sampling intervals to be served by one `Segment` table.

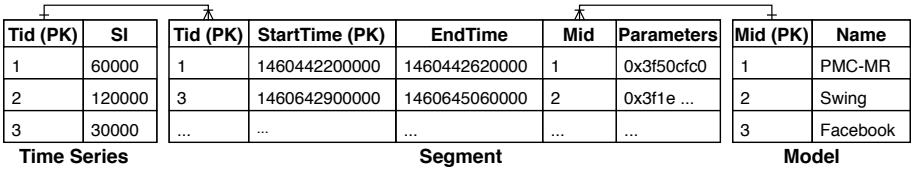

Figure B.6: Generic schema for storage of segments

6.1 Segment Storage in Apache Cassandra

Compression is the primary focus for ModelarDB's storage of segments in Cassandra. As Cassandra expects each column in a table to be independent, using `Tid, StartTime, EndTime` as the primary key only indicates to Cassandra that each partition is fully sorted by `StartTime`. As a result, adding `StartTime` and `EndTime`

to the primary key does not allow direct lookup of segments. However, as segments are partitioned by `Tid`, inside each partition the `EndTime` would be sorted as a consequence of `StartTime` being sorted. We utilize this for ModelarDB by partitioning each table on their respective ids, and use `EndTime` as the clustering column for the segment table so segments are sorted ascendingly by `EndTime` on disk. This allows the `Size` of a segment to be stored instead of `StartTime`, for a higher compression ratio, while allowing ModelarDB to exploit the partitioning and ordering of the segment table when executing queries as Cassandra can filter segments on `EndTime` while Spark loads segments until the required `StartTime` is reached. The `StartTime` column cannot be omitted due to the presence of gaps as explained in Section 4.2. To support indexing methods suitable for a specific storage system like in [29], secondary indexes can be implemented in ModelarDB as part of the storage interface shown in Figure B.3.

6.2 Predicate Push-Down

The columns for which predicate push-down is supported in our implementation are shown in Figure B.7. Each cell in the table shows how a predicate on a specific column is rewritten before it is pushed to the segment view or storage. Cells for the column `StartTime` marked with *Spark takeWhile* indicate that Spark reads rows from Cassandra in batches until the predicate represented by the cell is false for a segment. As explained above, this allows `StartTime` to be replaced with the column `Size` which stores the number of data points in the segment. This reduces the storage needed for start time without sacrificing its precision. When a segment is loaded, the start time of the segment can be recomputed as `StartTime = EndTime - (Size * SI)`, allowing Spark to load segments until the predicate represented by the cell is false for a segment. Non-equality queries on `Tid` are rewritten as Cassandra only supports equality queries on a partitioning key.

7 Evaluation

We compare ModelarDB to the state-of-the-art big data systems and file formats used in industry: Apache ORC [30, 31] files stored in HDFS [32], Apache Parquet [33] files stored in HDFS, InfluxDB [34], and Apache Cassandra [12]. InfluxDB is running on a single node as the open-source version does not support distribution. The number of nodes used for each experiment is shown in the relevant figures. Multi-model compression for time series is also evaluated in [4, 5, 21]. We first present the cluster, data sets and queries used in the evaluation, then we describe each experiment.

7.1 Evaluation Environment

The cluster consists of one *master* and six *worker* nodes connected by *1 Gbit Ethernet*. All nodes have an *Intel Core i7-2620M* 2.70 GHz, *8 GiB of 1333 MHz DDR3* memory

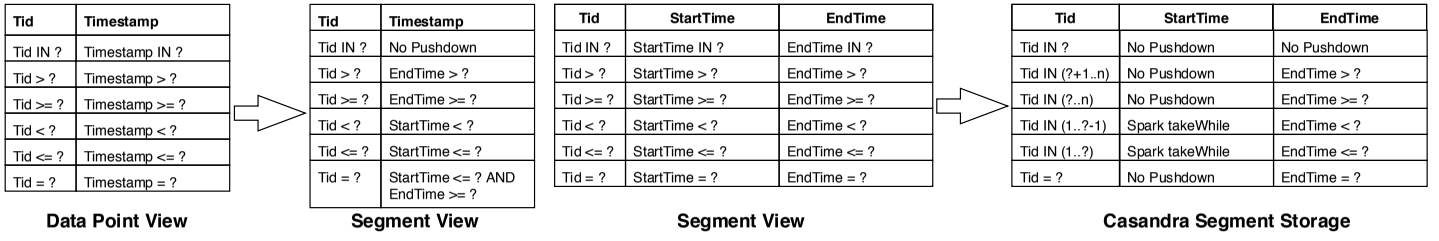

Figure B.7: The two-step methods for predicate push-down utilized by ModelarDB

Table B.3: The parameters we use for the evaluation

ModelarDB	**Value**
Error Bound	0%, 1%, 5%, **10%**
Limit	50
Latency	0
Bulk Write Size	50,000

Spark	**Value**
spark.driver.memory	4 GiB
spark.executor.memory	3 GiB
spark.streaming.unpersist	false
spark.sql.orc.filterPushdown	true
spark.sql.parquet.filterPushdown	true

Model	**Representation**	**Type of Compression**
PMC-MR [24]	Constant Function	Lossy Compression
Swing [25]	Linear Function	Lossy Compression
Facebook [19]	Array of Delta Values	Lossless Compression
Uncompressed Values	Array of Values	No Compression

and a 7,200 RPM hard-drive. Each node runs *Ubuntu 16.04 LTS*, *InfluxDB 1.4.2*, *InfluxDB-Python 2.12*, *Pandas 0.17.1*, *HDFS from Hadoop 2.8.0*, *Spark 2.1.0*, *Cassandra 3.9* and *DataStax Spark Cassandra Connector 2.0.2* on top of *EXT4*. The master is a *Primary HDFS NameNode*, *Secondary HDFS NameNode* and *Spark Master*. Each worker serves as an *HDFS Datanode*, a *Spark Slave*, and a *Cassandra Node*. Cassandra does not require a master node. Only the software necessary for an experiment is kept running and replication is disabled for all systems. Disk space utilization is found with *du*. The time series are stored using the same schema as the Data Point View: `Tid` as a `int`, `TS` using each storage method's native timestamp type, and `Value` as a `float`. InfluxDB is an exception as it only supports `double`. Ingestion for all storage methods is performed using `float`. For Parquet and ORC, one file is created per time series using Spark and stored in HDFS with one folder created for each data set and file format pair, for InfluxDB time series are stored as one measurement with the `Tid` as a tag, and for Cassandra we partition on `Tid` and order each partition on `TS` and `Value` for the best compression.

The configuration of each system is, in general, left with its default values. However, the memory available for Spark and either Cassandra or HDFS, is statically allocated to prevent crashes. To divide memory between query processing and data storage, we limit the amount of memory Spark can allocate per node, so the rest is available to Cassandra/HDFS and Ubuntu. Memory allocation is limited through Spark to ensure consistency across all experiments. The appropriate amount of memory for Spark is found by assigning half of the memory on each system to Spark and then reduce the memory allocated for Spark until all experiments can run successfully. We enable predicate push-down for Parquet and ORC. The parameters used are shown in Table B.3, with ModelarDB specific parameters in the upper left table, changes to Spark's default parameters in the upper right table, and the models implemented in ModelarDB Core shown in the bottom table. The parameter values are found to work well with the data sets and the hardware configuration. The error bound is 10% when not stated explicitly.

7.2 Data Sets and Queries

The data sets we use for the evaluation are regular time series where gaps are uncommon. Each data set is stored as CSV files with one time series per file and one data point per line.

Energy Production High Frequency This data set is referred to as EH and consists of time series from energy production. The data was collected by us from an OPC Data Access server using a Windows server connected to an energy producer. The data has an approximate sampling interval of 100ms. As pre-processing we round the timestamps and remove data points with equivalent timestamps due to rounding. This pre-processing step is only required due to limitations of our collection process and not present in a production setup. The data set is 582.68 GiB in size.

REDD The public Reference Energy Disaggregation Data Set (REDD) [35] is a data set of energy consumption from six houses collected over two months. We use the files containing energy usage in each house per second. Three of the twelve files have been sorted to correct a few out-of-order data points and the files from house six removed due to irregular sampling intervals. As REDD fits into memory on a single node, we extend it by replicating each file 2,500 times by multiplying all values of each file with a random in the range $[0.001, 1.001)$ and round each value to two decimals places to ensure our results are not impacted by identical files. 2,500 is selected due to the amount of storage in our cluster. The data set is 487.52 GiB in size, and is referred to as Extended REDD (ER). We use this public data set to enable reproducibility.

Energy Production This data set is referred to as EP and primarily consists of time series for energy production and is provided by an energy trading company. The data is collected over 508 days, has a sampling interval of 60s, and is 339 GiB in size. The data set also contains entity specific measurements such as wind speed for a wind turbine and horizontal irradiance for solar panels.

Queries The first set of queries (S-AGG) consists of small aggregate and `GROUP BY` queries and represents online analytics on one or a few time series, e.g., correlated

sensors, as analytical queries are the intended ModelarDB use case. Both types of queries are restricted by `Tid` using a `WHERE` clause, with the `GROUP BY` queries operating on five time series each and `GROUP` on `Tid`. The second set (L-AGG) consists of large aggregate and `GROUP BY` queries, which aggregate the entire data set and each `GROUP BY` query `GROUP`s on `Tid`. L-AGG is designed to evaluate the scalability of the system when performing its intended use case. The third set (P/R) contains time point and range queries restricted by `WHERE` clauses with either `TS` or `Tid` and `TS`. P/R represents a user extracting a sub-sequence from a time series, which is not the intended ModelarDB use case, but included for completeness. We do not evaluate SQL `JOIN` queries as they are not commonly used with time series, and similarity search is not yet built into ModelarDB.

7.3 Experiments

Ingestion Rate To remove the network as a possible bottleneck, the ingestion rate is mainly evaluated locally on a worker node. For each system/format we ingest channel one from house one of ER from gzipped CSV files (14.67 GiB) on local disks. Except for InfluxDB, ingestion is performed using a local instance of Spark through *spark-shell* with its default parameters. As no mature Spark Connector to our knowledge exists for InfluxDB, we use the InfluxDB-Python client library [36]. The input files are parsed using Pandas and InfluxDB-Python is configured with a batch size of 50,000. For Cassandra we increase the parameter `batch_size_fail_threshold_in_kb` to 50 MiB to allow larger batches. To determine ModelarDB's scalability we also evaluate its ingestion rate on the cluster in two different scenarios: *Bulk Loading (BL)* without queries and *Online Analytics (OA)* with aggregate queries continuously executed on random time series using the Segment View. When using a single worker, ModelarDB uses the single-node ingestor, and when it is distributed, Spark streaming with one receiver per node, a micro-batch interval of five seconds, and latency set to zero so each data point is part of a segment only once.

The results are shown in Figure B.8. As expected InfluxDB and Cassandra had the

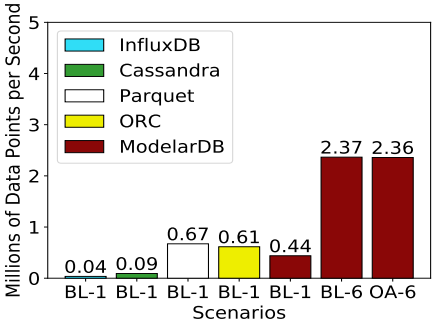

Figure B.8: Ingestion, ER

lowest ingestion rate, as they are designed to be queried while ingesting. ModelarDB also supports executing queries during ingestion but still provides 11 times and 4.89 times faster ingestion than InfluxDB and Cassandra, respectively. Parquet and ORC provided 1.52 and 1.39 times increase compared to ModelarDB, respectively. However, an entire file must be written before Parquet and ORC can be queried, making them unsuitable for online analytics due to the inherent latency of this approach. This compromise is not needed for ModelarDB as queries can be executed on data as it is ingested. When bulk loading on the six node cluster the ingestion rate for ModelarDB increases 5.39 times, a close to linear speedup. The ingestion is nearly unaffected, a 5.36 times increase, when doing online analytics in parallel. In summary, ModelarDB, achieves high ingestion rates while allowing online analytics, unlike the alternatives.

Effect of Error Bound and Outliers The trade-off between storage efficiency and error bound is evaluated using all three data sets. The models used and size of each data set are found when stored in ModelarDB with the *error bound* set to values from 0% to 10%. We compare the storage efficiency of ModelarDB with the systems used in industry. In addition, we evaluate the performance of ModelarDB's adaptive compression method when outliers are present, by adding an increasing number of outliers to each data set. The outliers are randomly created such that the average distance between two consecutive outliers is N and the value of each outlier is set to $(\textit{Value of Data Point to Replace} + 1) * 2$.

The storage used for EH is seen in Figure B.9. Of the existing systems, InfluxDB performs the best, but even with a 0% error bound, ModelarDB reduces the size of EH 1.52 times compared to InfluxDB. This is expected as the PMC-MR model can be used to perform run-length encoding while changing values are managed with delta-compression using the Facebook model. Increasing the error bound to 10% provides a further 1.18 times reduction, while the average actual error is only 0.005%. The results for ER are seen in Figure B.10. Compared to InfluxDB, ModelarDB provides much better compression: 2.40 times for 1%, 7.02 times for 5%, and 9.31 times 10% error bound. For ER, ORC is best of the existing systems, but ModelarDB further reduces by 2.13 times for 1%, 6.24 times for 5%, and 8.27 times for 10% error bound. In addition,

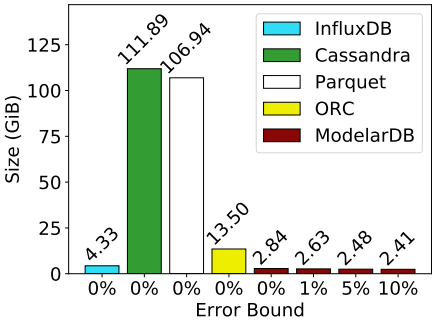

Figure B.9: Storage, EH

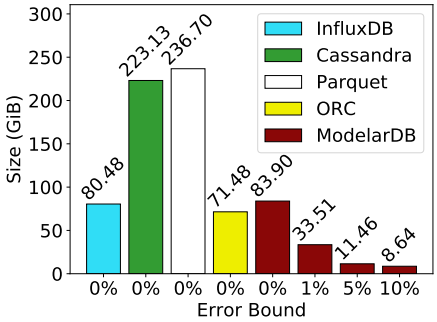

Figure B.10: Storage, ER

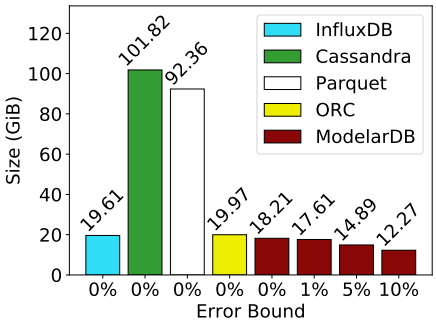

Figure B.11: Storage, EP

average actual error for ER is only 0.22% with 1% bound, 1.25% for 5% bound and 2.50% for 10% bound. Even with a 0% bound, ModelarDB uses just 1.17 times more storage than ORC. EP results are shown in Figure B.11. Here, ModelarDB provides the best compression, even at 0% error bound, however, the difference is smaller than for EH and ER. This is expected, as the EH and ER sampling intervals are 600 and 60 times lower, respectively, yielding more data points with similar values due to close time proximity. ModelarDB also manages to keep the average actual error for EP low at only 0.08% for 1%, 0.48% for 5% and 0.73% for a 10% bound.

The models utilized for each data set are shown in Figure B.12—B.14. Overall PMC-MR and Facebook are the most utilized models, with Swing used sparingly except for EP with 5% and 10% error. Note, that Swing also is utilized for ER and EP with a 0% error bound as perfectly linear increases and decreases of values do exist in the data sets. Last, except for EH, multiple models are used extensively. These results clearly show the benefit and adaptivity of multi-model compression as each combination of data set and error bound can be handled efficiently using different combinations of the models.

The effect of outliers is shown in Figure B.15. As expected the storage used

Figure B.12: Models, EH

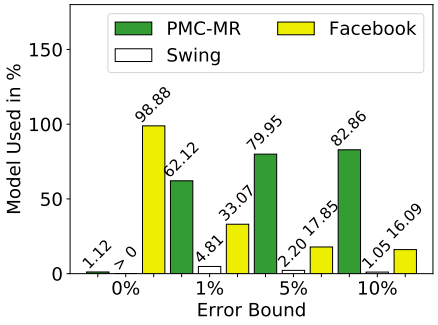

Figure B.13: Models, ER

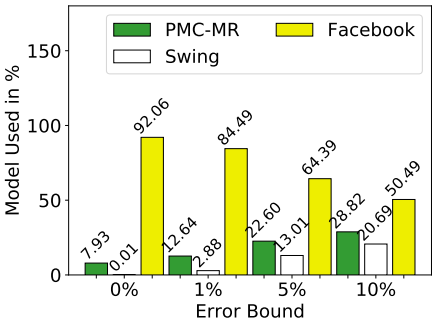

Figure B.14: Models, EP

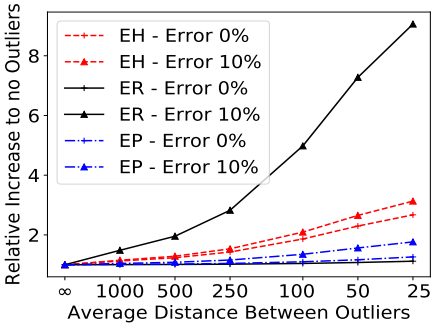

Figure B.15: Outlier Effect

increases with the number of outliers, but the increase depends on the data set and error bound. For all data sets, ModelarDB degrades gracefully as additional outliers are added to the data set. As the values of N decrease below 250 the relative size increases more rapidly as the high number of outliers severely restrict the length of the segments ModelarDB can construct. The results also show that ModelarDB is more robust against outliers when a 0% error bound is used. With a 10% error bound the relative increase for EH and EP is slightly higher than for a 0% error bound, while the relative size increase for ER with a 10% error bound in the extreme case of $N = 25$ is 9.06 while it is only 1.12 with a 0% error bound. This is expected as ER has a high compression ratio with a 10% error bound and the high number of outliers prevents ModelarDB from constructing long segments. The results show that although designed for time series with few outliers, ModelarDB degrades gracefully as the amount of outliers increases. In summary, ModelarDB provides as good compression as the existing formats when a 0% error bound is used, and much better compression for even a small error bound, by combining different models depending on the data and error bound.

Scale-out To evaluate ModelarDB's scalability we first compare it to the existing systems when executing L-AGG on ER using the test cluster. Then, to determine the scalability of ModelarDB on large clusters, we execute L-AGG on Microsoft Azure using 1—32 Standard_D8_v3, the node type is selected based on the documentation for Spark, Casandra and Azure [37–39]. The configuration from the test cluster is used with the exception that Spark and Cassandra each have access to 50% of each node's memory as no crashes were observed with this initial configuration. For each experiment REDD is duplicated, using the method described above, and ingested so each node stores compressed data equivalent to the node's memory. This makes caching the data set in memory impossible for Spark and Cassandra. The number of nodes and data size are scaled in parallel based on existing evaluation methodology for distributed systems [40]. Queries are executed using the most appropriate method for each system: InfluxDB's command-line interface (CLI), ModelarDB's Segment View (SV) and Data Point View (DPV), and for Cassandra, Parquet, and ORC a Spark SQL Data Frame

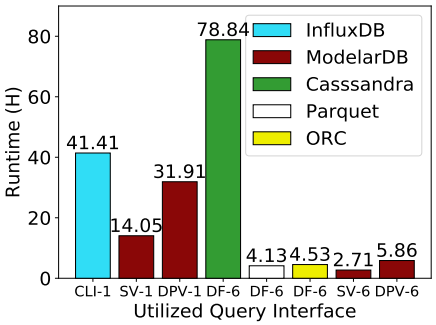

Figure B.16: L-AGG, ER

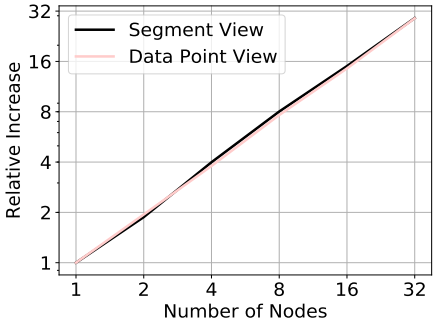

Figure B.17: Scale-out

(DF). We evaluate the query performance using a DF and a cached Data Frame (DFC) as shown in Figure B.25. However, as DFCs increased the run-time, as the data was inefficiently spilled to disk, we only use DFs for the other queries.

The results are shown in Figure B.16—B.17. For both views ModelarDB achieves close to linear scale-up. This is expected as queries can be answered with no shuffle operations as all segments of a time series are co-located. However, using SV, the query processing time is significantly reduced as SV does not reconstruct the data points which reduces both CPU and memory usage. On one node SV is 2.27 times faster than DPV, and 2.16 times faster with six nodes. For L-AGG on ER, ModelarDB is faster than all the existing systems. Compared to InfluxDB, on one node, ModelarDB is 2.95 times and 1.30 times faster for SV and DPV, respectively. Using six nodes, ModelarDB is 1.52 times and 1.67 times faster than Parquet and ORC, respectively. In summary, ModelarDB scales almost linearly while providing better performance than the competitors.

Effect of Optimizations To evaluate the code generation and predicate push-down optimizations, we execute L-AGG and P/R on ER both with and without the optimizations. As a comparison to static code generation, we implement a straightforward

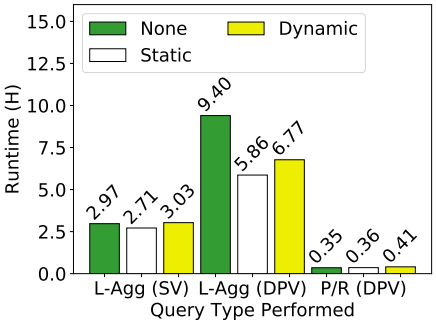

Figure B.18: Projection, ER

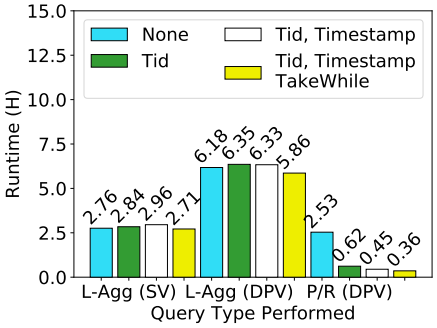

Figure B.19: Predicate, ER

dynamic code generator using `scala.tools.reflect.ToolBox` and Spark's `mapPartitions` transformation. By default ModelarDB uses static code-generation for projections and predicate push-down for `Tid`, `Timestamp`, and `takeWhile`.

The results for projection in Figure B.18 show that generating optimized code for projections decreases the run-time up to 1.60 times compared to constructing each row dynamically. However, using our implementation of dynamic code generation increases the run-time compared to our static code generation. The results for predicate push-down are seen in Figure B.19. Predicate push-down has little effect on the query processing time for L-AGG, but the reduction is more pronounced for P/R where we see a 7.03 times reduction. This is to be expected as all queries in L-AGG must read the entire data set from disk, while all queries in P/R can be answered using only a small subset.

Further Query Processing Performance To further evaluate the query performance of ModelarDB, we execute S-AGG and P/R on all data sets using the query interfaces described for scale-out.

The results for S-AGG are shown in Figures B.20—B.22. Once again the run-time is reduced using the SV. While InfluxDB performs slightly better than ModelarDB for S-AGG on ER and EP, it is limited in terms of scalability and ingestion rate, as shown above where ModelarDB executes L-AGG on ER 2.95 times faster than InfluxDB. Compared to the distributed systems, ModelarDB provides as good, or better query processing times for nearly all cases. On EH, ModelarDB is 1.4 times faster than ORC and 152.62 times faster than Cassandra. For EH, Parquet is faster, but for all other data sets ModelarDB is faster and uses much less storage space. For ER, which is a core use case scenario, ModelarDB reduces the query processing time by staggering 34.57, 286.03 and 45.99 times, compared to Cassandra, Parquet and ORC, respectively. Last, for EP, Cassandra and ModelarDB provide the lowest query processing time with ModelarDB being 11.33 times faster. Thus for our experiments ModelarDB improves the query processing time significantly compared to the other distributed systems, in core cases by large factors, while it for small-scale aggregate queries

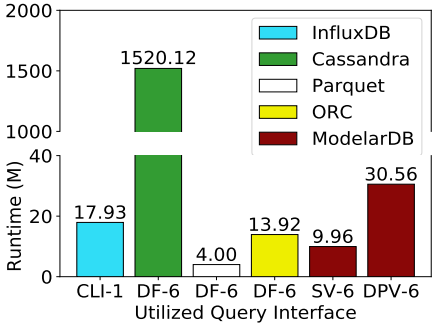

Figure B.20: S-AGG, EH

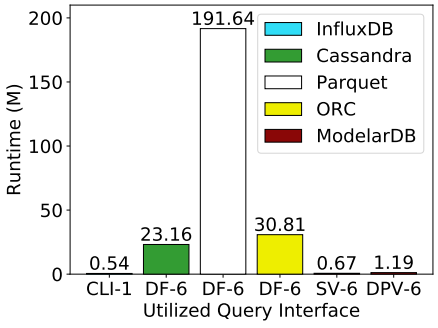

Figure B.21: S-AGG, ER

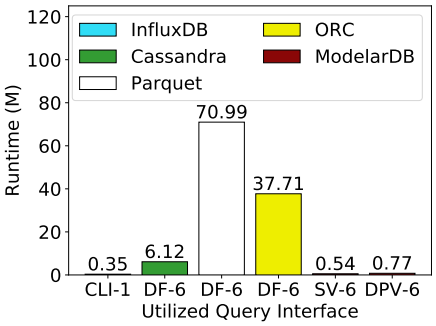

Figure B.22: S-AGG, EP

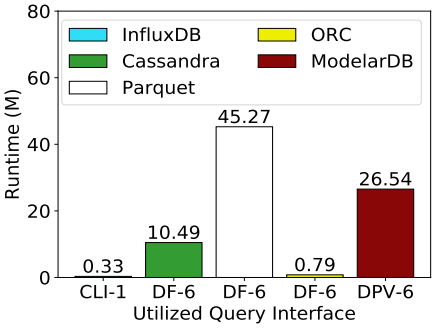

Figure B.23: P/R, EH

remains competitive with an optimized single node system.

The results for P/R are shown in Figures B.23—B.25. For P/R, InfluxDB and Cassandra perform the best, which contrasts our scale-out experiment where they perform the worst. Clearly these systems are optimized for queries on a small subset of a data set, while Parquet, ORC, and ModelarDB, are optimized for aggregates on large data sets. While P/R queries are not a core use case, ModelarDB provides equivalent performance for P/R queries in most cases and is only significantly slower for EH. When compared to ORC and Parquet, ModelarDB is in general faster and in the best case, ER, provides 1.63 times faster query processing time. In one single instance, for EH, ORC is 33.59 times faster than ModelarDB due to better predicate push-down, as disabling predicate push-down for ORC increases the run-time from 47.64 seconds to 1 hour and 40 minutes. However, unlike Parquet and ORC, time series ingested by ModelarDB can be queried online. An interesting result was that DFCs increased the query processing time, particularly for the first query. This indicates that the data set is read from and spilled to disk due to a lack of memory during the initial query with subsequent queries executed using Spark's on-disk format.

For our experiments, the other systems require a trade-off as they are *either* good at

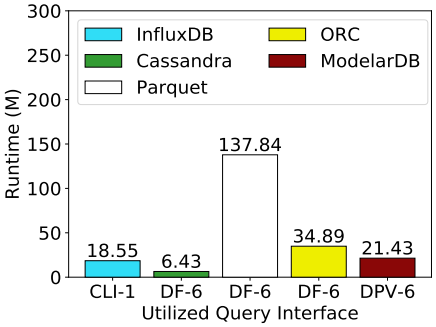

Figure B.24: P/R, ER

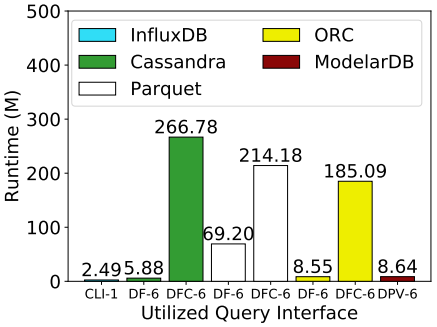

Figure B.25: P/R, EP

large aggregate queries *or* point/range and small aggregate queries and support online analytics, but not both. ModelarDB hits a sweet spot, improving the state-of-the-art for online analytics by providing fast ingestion, better compression and scalability for large aggregate queries while remaining competitive with other systems for small aggregate and point-range queries. This combination of features is not provided by any competitor.

8 Related Work

Management of sensor data using models has received much attention as the amounts of data have increased. We provide a discussion of the most relevant related methods and systems. For a survey of model-based sensor data management see [41], while a survey of TSMSs can be found in [3].

Methods have been proposed for online construction of approximations with minimal error [42], or maximal segment length for better compression [23]. As the optimal model can change over time, methods using multiple mathematical models have been developed. In [5] each data point in the time series is approximated by all models in a set. A model is removed from the set if it cannot represent a data point within the error bound. When the set of models is empty, the model with the highest compression ratio is stored and the process restarted. A relational schema for segments was discussed in [21]. The Adaptive Approximation (AA) algorithm [6] uses functions as models with an extensible method for computing the coefficients. The AA algorithm approximates each data point in the time series until a model exceeds the error bound and a local segment is created for that model and the model reset. After all models have constructed one local segment, the segments using the lowest number of parameters are stored. An algorithm based on regression was proposed in [4]. It approximates data points with a single polynomial model and then increases the number of coefficients as the error bound becomes unsatisfiable. As the user-defined maximum number of coefficients is reached, the model with the highest compression ratio is stored and the time series rewound to the last data point of the stored segment. In this paper, we propose a multi-model compression algorithm for time series that improves on the state-of-the-art as it supports user-defined models, supports lossy and lossless models, and removes the trade-off between compression and latency inherent to the existing methods.

In addition to techniques for representing time series as models, RDBMSs with support for models have also been proposed. MauveDB [26] supports using models for data cleaning without exporting the data to another application. Models are explicitly constructed from a table with raw data and then maintained by the RDBMS. Model-based views created with a static sampling interval serve as the query interface for the models. FunctionDB [27] natively supports polynomial functions as models. The RDBMS's query processor evaluates queries directly on these models when possible, with the query results provided as discrete values. Model fitting is performed manually

by fitting a specific model to a table. Maintenance of the constructed models is outside the scope of the paper. Plato [1] allows for user-defined models. Queries are executed on models if the necessary functions are implemented and discrete values if not. The granularity at which to instantiate a model for a query can be specified with a grid operator or left to Plato. Fitting models to a data set is done manually as automated model selection is left as future work. Tristan [43], based on the MiSTRAL architecture [44], approximates time series as sequences of fixed-length time series patterns using dictionary compression. Before ingestion a dictionary must be trained offline on historical data. During ingestion a fixed number of data points are buffered before the compression is applied and the dictionary updated with new patterns if necessary. For approximate query processing a subset of the patterns stored for a time series is used. A distributed approach to model-based storage of time series using an in-memory tree-based index, a key-value store, and MapReduce [45] was proposed by [29]. The segmentation is performed using Piecewise Constant Approximation (PCA) [41]. Each segment is stored and indexed twice, once by time and once by value. Query processing is performed by locating segments using the index, retrieving segments from the store using mappers, and last, instantiating each model using reducers. ModelarDB hits a sweet spot and provides functionality in a single extensible TSMS not present in the existing systems: storage and query processing for time series within a user-defined error bound [1, 26, 27], support for both fixed and dynamically sized user-defined models that can be fitted online without requiring offline training of any kind [43], and automated selection of the most appropriate model for each part of a time series while also storing each segment only once [29].

9 Conclusion & Future Work

Motivated by the need for storage and analysis of big amounts of data from reliable sensors, we presented a general architecture for a modular model-based TSMS and a concrete system using it, *ModelarDB*. We proposed a model-agnostic extensible and adaptive multi-model compression algorithm that supports both lossless and lossy compression within a user-defined error bound. We also presented general methods and optimizations usable in a model-based TSMS: (i) a database schema for storing multiple distinct time series as (possibly user-defined) models, (ii) methods to push-down predicates to a key-value store that utilizes the presented schema, (iii) methods to execute optimized aggregate functions directly on models without requiring a dynamic optimizer, (iv) static code-generation to optimize execution of projections, (v) dynamic extensibility that allows user-defined models to be added and used without recompiling the TSMS. The architecture was realized as a portable library which was interfaced with Apache Spark for query processing and Apache Cassandra for storage. Our evaluation showed that, unlike current systems, ModelarDB hits a sweet spot and achieves fast ingestion, good compression, almost linear scale-out, and fast aggregate query processing, in the same system, while also supporting online queries. The

evaluation further demonstrated how the contributions effectively work together to adapt to the data set using multiple models, that the actual errors are much lower than the bounds, and how ModelarDB degrades gracefully as outliers are added.

In future work we are planning to extend ModelarDB in multiple directions: (i) Increase query performance by developing new techniques for indexing segments represented by user-defined models, performing similarity search directly on user-defined models, and performing dynamic optimizations utilizing that the time series are represented as models. (ii) Reduce the storage needed for large volumes of sensor data by representing correlated streams of sensor data as a single stream of segments. (iii) Further simplify use of ModelarDB by removing or automatically inferring parameters.

10 Acknowledgements

This research was supported by the DiCyPS center funded by Innovation Fund Denmark [46], and Microsoft Azure for Research [47].

References

[1] Y. Katsis, Y. Freund, and Y. Papakonstantinou, "Combining Databases and Signal Processing in Plato," in *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR)*, 2015.

[2] "DB-Engines Ranking," https://db-engines.com/en/ranking, Viewed: 2019-08-14.

[3] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time Series Management Systems: A Survey," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 29, no. 11, pp. 2581–2600, November 2017.

[4] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm, "A time-series compression technique and its application to the smart grid," *The VLDB Journal (VLDBJ)*, vol. 24, no. 2, pp. 193–218, April 2015.

[5] T. G. Papaioannou, M. Riahi, and K. Aberer, "Towards Online Multi-Model Approximation of Time Series," in *Proceedings of the 12th International Conference on Mobile Data Management (MDM)*, vol. 1. IEEE, 2011, pp. 33–38.

[6] J. Qi, R. Zhang, K. Ramamohanarao, H. Wang, Z. Wen, and D. Wu, "Indexable online time series segmentation with error bound guarantee," *World Wide Web (WWW)*, vol. 18, no. 2, pp. 359–401, March 2015.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2010.

[8] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters," in *Proceedings of the 4th Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2012.

[9] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP)*. ACM, 2013, pp. 423–438.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2015, pp. 1383–1394.

[11] "Apache Spark," https://spark.apache.org/, Viewed: 2019-08-14.

[12] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[13] "Apache Cassandra," http://cassandra.apache.org/, Viewed: 2019-08-14.

[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 38, no. 4, pp. 28–38, December 2015.

[15] "Apache Flink," https://flink.apache.org/, Viewed: 2019-08-14.

[16] "Apache HBase," https://hbase.apache.org/, Viewed: 2019-08-14.

[17] "Apache MongoDB," https://www.mongodb.com/, Viewed: 2019-08-14.

[18] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient In-Memory Spatial Analytics," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2016, pp. 1071–1085.

[19] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A Fast, Scalable, In-Memory Time Series Database," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1816–1827, August 2015.

[20] R. E. Korf, "Multi-Way Number Partitioning," in *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2009, pp. 538–543.

[21] T. G. Papaioannou, M. Riahi, and K. Aberer, "Towards Online Multi-Model Approximation of Time Series," https://infoscience.epfl.ch/record/164651/files/tech_report.pdf, EPFL LSIR, Tech. Rep., 2011.

[22] N. Q. V. Hung, H. Jeung, and K. Aberer, "An Evaluation of Model-Based Approaches to Sensor Data Compression," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 25, no. 11, pp. 2434–2447, November 2013.

[23] G. Luo, K. Yi, S.-W. Cheng, Z. Li, W. Fan, C. He, and Y. Mu, "Piecewise Linear Approximation of Streaming Time Series Data with Max-error Guarantees," in *Proceedings of the 31st International Conference on Data Engineering (ICDE)*. IEEE, 2015, pp. 173–184.

[24] I. Lazaridis and S. Mehrotra, "Capturing Sensor-Generated Time Series with Quality Guarantees," in *Proceedings of the 19th International Conference on Data Engineering (ICDE)*. IEEE, 2003, pp. 429–440.

[25] H. Elmeleegy, A. K. Elmagarmid, E. Cecchet, W. G. Aref, and W. Zwaenepoel, "Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 145–156, August 2009.

[26] A. Deshpande and S. Madden, "MauveDB: Supporting Model-based User Views in Database Systems," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2006, pp. 73–84.

[27] A. Thiagarajan and S. Madden, "Querying Continuous Functions in a Database System," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2008, pp. 791–804.

[28] "Java Virtual Machine Specification," https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-3.html#jvms-3.10, Viewed: 2019-08-14.

[29] T. Guo, T. G. Papaioannou, and K. Aberer, "Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud," *Big Data Research*, vol. 1, pp. 52–65, August 2014.

[30] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major Technical Advancements in Apache Hive," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 1235–1246.

[31] "Apache ORC," https://orc.apache.org/, Viewed: 2019-08-14.

[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[33] "Apache Parquet," https://parquet.apache.org/, Viewed: 2019-08-14.

[34] "InfluxData InfluxDB," https://www.influxdata.com/, Viewed: 2019-08-14.

[35] J. Z. Kolter and M. J. Johnson, "REDD: A Public Data Set for Energy Disaggregation Research," in *Proceedings of the Workshop on Data Mining Appliations in Sustainbility (SustKDD)*. ACM, 2011.

[36] "InfluxDB API Client Libraries," https://docs.influxdata.com/influxdb/v1.4/tools/api_client_libraries/, Viewed: 2019-08-14.

[37] "Azure Databricks," https://azure.microsoft.com/en-us/pricing/details/databricks/, Viewed: 2019-08-14.

[38] "Apache Cassandra - Hardware Choices," http://cassandra.apache.org/doc/latest/operating/hardware.html, Viewed: 2019-08-14.

[39] "Apache Spark - Hardware Provisioning," https://spark.apache.org/docs/2.1.0/hardware-provisioning.html, Viewed: 2019-08-14.

[40] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st Symposium on Cloud Computing (SOCC)*. ACM, 2010, pp. 143–154.

[41] S. Sathe, T. G. Papaioannou, H. Jeung, and K. Aberer, "A Survey of Model-based Sensor Data Acquisition and Management," in *Managing and Mining Sensor Data*. Springer, 2013, pp. 9–50.

[42] T. Guo, Z. Yan, and K. Aberer, "An Adaptive Approach for Online Segmentation of Multi-Dimensional Mobile Data," in *Proceedings of the Eleventh International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*. ACM, 2012, pp. 7–14.

[43] A. Marascu, P. Pompey, E. Bouillet, M. Wurst, O. Verscheure, M. Grund, and P. Cudre-Mauroux, "TRISTAN: Real-Time Analytics on Massive Time Series Using Sparse Dictionary Compression," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 291–300.

[44] A. Marascu, P. Pompey, E. Bouillet, O. Verscheure, M. Wurst, M. Grund, and P. Cudre-Mauroux, "MiSTRAL: An Architecture for Low-Latency Analytics on Massive Time Series," in *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 2013, pp. 15–21.

[45] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2004, pp. 137–150.

[46] “DiCyPS - Center for Data-Intensive Cyber-Physical Systems,” http://www.dicyps.dk/dicyps-in-english/, Viewed: 2019-08-14.

[47] “Microsoft Azure for Research,” https://www.microsoft.com/en-us/research/academic-program/microsoft-azure-for-research/, Viewed: 2019-08-14.

Paper C

Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB

Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen

Unpublished Manuscript.

Abstract

To monitor critical infrastructure, high quality sensors sampled at a high frequency are increasingly installed. However, due to the big amounts of data produced, only simple aggregates are stored. This removes outliers and hides fluctuations that could indicate problems. As a solution we propose compressing time series with dimensions using a model-based method we name MMGC*. MMGC adaptively compresses groups of correlated time series using an extensible set of model types within a user-defined error bound (possibly zero). To partition time series into groups, we propose a set of primitives for efficiently describing correlation for data sets of varying sizes. We also propose efficient query processing algorithms for executing multi-dimensional aggregate queries on models instead of data points. Last, we provide an open-source implementation of our methods as extensions to the model-based TSMS ModelarDB. ModelarDB interfaces with the stock versions of Apache Spark and Apache Cassandra and can thus reuse existing infrastructure. Through an evaluation we show that, compared to widely used systems, our extended ModelarDB provides up to 13 times faster ingestion due to high compression, 65 times better compression due to the adaptivity of MMGC, 391 times faster aggregate queries as they are executed on models, and close to linear scalability while also being extensible and supporting online query processing.*

The layout has been revised.

1 Introduction

Companies managing energy production benefit from monitoring with a high degree of coverage and having data points sampled at a high frequency to manage their critical infrastructure. Energy producing entities, such as wind turbines and solar panels, are thus monitored by regularly sampling high quality sensors with wired power and connectivity. As a result, invalid, missing and out-of-order readings are rare, and all except missing values can be corrected using established methods. In addition to data points, metadata, e.g., location and sensor type, is stored for each time series to support analysis along multiple dimensions. The collected data is used by park owners and turbine manufactures for management and warranty purposes. Based on discussions with both owners and manufacturers, we learned that storing such high volumes of raw sensor data is either infeasible or prohibitively expensive. Instead only simple aggregates are stored, e.g., 1–10 minute averages, removing outliers and fluctuations as a result. Use of a dynamic sampling rate can reduce the amount of data to store by only storing critical events at high detail. However, what constitutes a critical event is not known by the park owners so a constant high sampling rate is needed. As a remedy, *model-based* storage uses *models* to reconstruct the original time series within a *known error bound* (possibly zero) [1, 2]. Model-based storage of time series has been improved through MMC and MGC. MMC utilizes that the structure of time series changes over time and compresses each time series using multiple model types [3–7]. MGC exploits that time series are correlated, e.g., temperature sensors in close proximity likely report similar values, and compresses correlated time series as one stream of models [8, 9]. MGC is illustrated in Figure C.1 where a linear function ($v = a \times t + b$) is used to represent three correlated time series, creating a mapping from a timestamp t to an approximated value v for the three values observed at that timestamp.

However, to our knowledge no method for MMC exploits the correlation between time series, while existing methods for MGC each only utilizes one model type. In this paper, we focus on the novel problem of compressing groups of correlated time

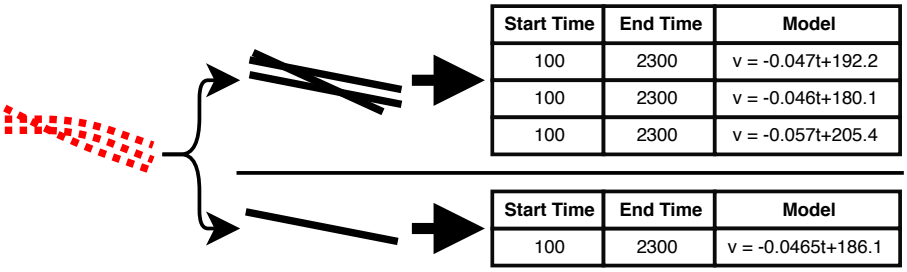

Figure C.1: Time series compressed and stored as one model per time series (Top), or as one model for all time series (Bottom)

series with user-defined dimensions using both MMC and MGC. We name this new type of compression *Multi-model Group Compression (MMGC)*. We demonstrate that MMGC is suitable for use with a TSMS by extending the open-source MMC TSMS *ModelarDB* [7] with MMGC. To differentiate between the two versions of ModelarDB we use *ModelarDB$_{v1}$* for the original version and *ModelarDB$_{v2}$* for the new version extended with MMGC. We also demonstrate how multi-dimensional aggregate queries can be performed much more efficiently on models compared to data points. As a result, ModelarDB$_{v2}$ provides a high compression ratio for time series data, distributed storage and query processing for scalability, stream processing for low latency, and efficient support for multi-dimensional aggregate queries of time series. In summary, we make the following contributions in the area of big data systems:

- The concept of Multi-model Group Compression and extensions of existing model types for compressing groups of time series.
- Primitives for partitioning time series into groups of correlated time series using their dimension hierarchy and user hints.
- Algorithms for performing simple and multi-dimensional aggregate queries on models representing groups of time series.
- The TSMS ModelarDB implementing our methods for partitioning, Multi-model Group Compression, and query processing.
- An evaluation of ModelarDB with the algorithms for partitioning, Multi-model Group Compression, and query processing.

The structure of the paper is as follows. Definitions are provided in Section 2. Section 3 provides an overview of ModelarDB$_{v2}$. Section 4 documents our partitioning primitives, while Section 5 describes our MGC extensions to existing model types. In Section 6 our query processing algorithms are described. An evaluation of ModelarDB$_{v2}$ is given in Section 7. Related work is presented in Section 8. Last, Section 9 provides our conclusion and future work.

2 Preliminaries

We now provide definitions for use in the paper. As we extend ModelarDB$_{v1}$, Definitions C.1–C.4 and C.6–C.7 are mainly reproduced from [7].

Definition C.1 (Time Series)
A *time series* TS is a sequence of *data points*, in the form of timestamp and value pairs, ordered by time in increasing order $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$. For each pair (t_i, v_i), $1 \leq i$, the timestamp t_i represents when the value $v_i \in \mathbb{R}$ was recorded. A time series $TS = \langle (t_1, v_1), \ldots, (t_n, v_n) \rangle$ with a fixed number of n data points is a *bounded time series*.

Definition C.2 (Regular Time Series)
A time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ is considered *regular* if the time elapsed between each data point is always the same, i.e., $t_{i+1} - t_i = t_{i+2} - t_{i+1}$ for $1 \leq i$ and *irregular* otherwise.

Definition C.3 (Sampling Interval)
The *sampling interval* of a regular time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ is the time elapsed between each pair of data points in the time series $SI = t_{i+1} - t_i$ for $1 \leq i$.

Consider for example the time series $TS = \langle (100, 188.5), (200, 181.8), (300, 179.15), (400, 172.4), (500, 169.7), \ldots \rangle$. The timestamps represent milliseconds since recording started. To construct a bounded time series, we can consider a subset of the data points where, e.g., $100 \leq t_i \leq 500$. Both versions of TS are regular and have a SI of 100 milliseconds.

Definition C.4 (Model)
A *model* of a time series $TS = \langle (t_1, v_1), \ldots \rangle$ is a pair of functions $M = (m, e_m)$. For each t_i, $1 \leq i$, m is a real-valued mapping from t_i to an estimate of the value v_i for the corresponding data point in TS. e_m is a mapping from TS and the corresponding m to a non-negative real value representing the error of the values estimated by m.

Definition C.5 (Model Type)
A *model type* is a partial function $m_t(TS, \epsilon)$, which when defined for a bounded time series TS and a non-negative real number ϵ returns a model $M = (m, e_m)$ of TS such that $e_m(TS, m) \leq \epsilon$. We call ϵ the *error bound*.

For example, Swing [10] is a model type which computes the parameters a and b for a linear function $at_i + b$ to represent the values of the bounded time series TS within ϵ using the uniform norm. We say a model is *fitted* to a time series when its parameters are computed. An example of a model is the representation of TS within $\epsilon = 5$ according to the uniform norm $e_m = \max(|v_i - m(t_i)|)$, $1 \leq i \leq 5$ using the linear function $m = -0.047t_i + 192.2$, $100 \leq t_i \leq 500$. This model has an error of $|169.7 - (-0.047 \times 500 + 192.2)| = 1$ as the uniform norm uses the largest difference between a real and an approximated value as the error.

Definition C.6 (Gap)
A *gap* between a regular bounded time series $TS_1 = \langle (t_1, v_1), \ldots, (t_s, v_s) \rangle$ and a regular time series $TS_2 = \langle (t_e, v_e), (t_{e+1}, v_{e+1}), \ldots \rangle$ with the same sampling interval SI and recorded from the same source, is a pair of timestamps $G = (t_s, t_e)$ with $t_e = t_s + m \times SI$, $m \in \mathbb{N}_{\geq 2}$, and where no data points exist between t_s and t_e.

Definition C.7 (Regular Time Series with Gaps)
A *regular time series with gaps* is a regular time series, $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ where $v_i \in \mathbb{R} \cup \{\perp\}$ for $1 \leq i$. For a regular time series with gaps, a gap $G = (t_s, t_e)$ is a sub-sequence where $v_i = \perp$ for $t_s < t_i < t_e$.

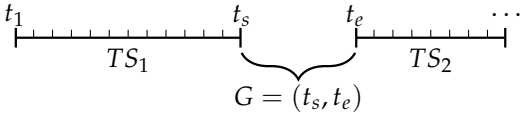

Figure C.2: Illustration of a gap G between t_s and t_e

A gap is shown in Figure C.2. For simplicity time series from the same source separated by gaps are referred to as a time series with gaps. For example $TS_g = \langle(100, 188.45), (200, 181.8), (300, 179.15), (400, 172.4), (500, 169.7), (1100, 141.5), \ldots\rangle$ has the gap $G = (500, 1100)$. As TS_g contains a gap it is irregular with an undefined SI. However, TS_g can also be represented as a regular time series with gaps $TS_{rg} = \langle(100, 188.45), (200, 181.8), (300, 179.15), (400, 172.4), (500, 169.7), (600, \bot), (700, \bot), (800, \bot), (900, \bot), (1000, \bot), (1100, 141.5), \ldots\rangle$ with $SI = 100$ milliseconds.

Definition C.8 (Dimension)
A *dimension* with members M is a 3-tuple $D = (member : \mathbb{TS} \rightarrow M, level : M \rightarrow \{0, 1, \ldots, n\}, parent : M \rightarrow M)$ where *(i)* M is hierarchically organized descriptions of the time series in the set of time series $\mathbb{TS}$ with the special value $\top \in M$ as the top element of the hierarchy; *(ii) level* is surjective; *(iii)* For $TS \in \mathbb{TS}$, $level(member(TS)) = n$ and $\nexists m \in M$ where $level(m) > n$; *(iv)* For $TS \in \mathbb{TS}$, $m \in M$ and $k \neq \top$, if $level(m) = k$ then $level(parent(member(TS))) = k - 1$; *(v)* $parent(\top) = \top$; *(vi)* $level(\top) = 0$.

A time series belongs to a dimension's most detailed level that has no descendants. Each member (except $\top$) at a level k has a parent at level $k - 1$. This allows users to do analysis at different levels by grouping on a level. To better describe the relation of the time series to real-world entities we will be writing dimensions using named levels. For example, for time series collected from wind turbines a location dimension could be defined as *Turbine* $\rightarrow$ *Park* $\rightarrow$ *Region* $\rightarrow$ *Country* $\rightarrow \top$. For a time series TS, the function $member(TS)$ then provides a member for the *Turbine* level, while $parent(member(TS))$ provides a member for the *Park* level. If TS is collected from a sensor on a wind turbine with id 9834 placed in Aalborg, the member for the first level is $member(TS) = 9834$, while the member for the next level is $parent(9834) = Aalborg$ until *parent* returns $\top$ indicating the top of the hierarchy.

Definition C.9 (Time Series Group)
A *time series group* is a set of regular time series, possibly with gaps, $TSG = \{TS_1, \ldots, TS_n\}$, where for $TS_i, TS_j \in TSG$ it holds that they have the same sampling interval SI and that $t_{1i} \bmod SI = t_{1j} \bmod SI$ where t_{1i} and t_{1j} are the first timestamp of TS_i and TS_j, respectively.

For example, $TSG = \{TS, TS_{rg}\}$ is a time series group which contains the time series TS and the regular time series with gaps TS_{rg}, both with $SI = 100$ milliseconds.

The irregular time series TS_g cannot be in the set as it does not have $SI = 100$ milliseconds. Different time series can also be correlated even if they consist of very different values. For example, $TS_1 = \langle(100, 188.45), (200, 181.8), (300, 179.15)$ and $TS_2 = \langle(100, 282.7), (200, 272.7), (300, 268.73)$ are correlated, but consist of different values. Scaling the values of TS_1 by 1.5, however, makes the values of TS_1 and TS_2 approximately equal and they can thus be compressed together. During query processing, the scaling must obviously also be taken in account to reproduce the original values.

To uphold an error bound for a model-based representation, a time series can be split into segments which are dynamically sized subsequences. For each segment a model represents its values and the segments of a given time series may use different model types. Subsequences of multiple time series in a group may even be represented by a single segment such that a single model is used to approximate values for all time series in the group.

Definition C.10 (Segment)
A *segment* is a 5-tuple $S = (t_s, t_e, SI, G_{ts} : TSG \rightarrow 2^{\{t_s, t_s+SI, \ldots, t_e\}}, M)$ representing the data points for a bounded time interval of a time series group TSG. The 5-tuple consists of start time t_s, end time t_e, sampling interval SI, a function G_{ts} which for the $TS \in TSG$ gives the set of timestamps for which $v = \perp$ in TS, and where the values of all other timestamps are defined by the model M multiplied by a scaling constant $C_{TS} \in \mathbb{R}$.

As an example of a segment we use the series $TS_1 = (100, 187.5), (200, 182.8), (300, 178.1), (400, 173.4), (500, 183.7)$, $TS_2 = (100, 175.5), (200, 170.9), (300, 166.3), (400, 161.7), (500, 179.1)$ and $TS_3 = (100, 189.7), (200, 184), (300, 178.3), (400, 174.6), (500, 172.9)$. Representing these time series with the linear function $m = -0.0465t_i + 186.1$ creates an approximation with the error $|183.7 - (-0.0465 \times 500 + 186.1)| = 20.85$ when using the uniform norm. If the error bound, e.g., is 10, the segment $S = (100, 400, 100, G_{ts} = \emptyset, (m = -0.0465t_i + 186.1, e_m = \max(|v_i - e_m(t_i)|)))$, $1 \leq i \leq 4$, is created.

In this paper we focus on using MMGC to compress unbounded regular times series, possibly with gaps and dimensions, while the time series are being ingested by a TSMS and analyzed using data warehouse style Online Analytical Processing (OLAP) queries.

3 Overview

3.1 Architecture

ModelarDB$_{v2}$ is designed as a portable library, ModelarDB$_{v2}$ Core, that is simple to interface with existing software. We interface it with the stock versions of Apache Spark for query processing and Apache Cassandra for storage in a master/worker

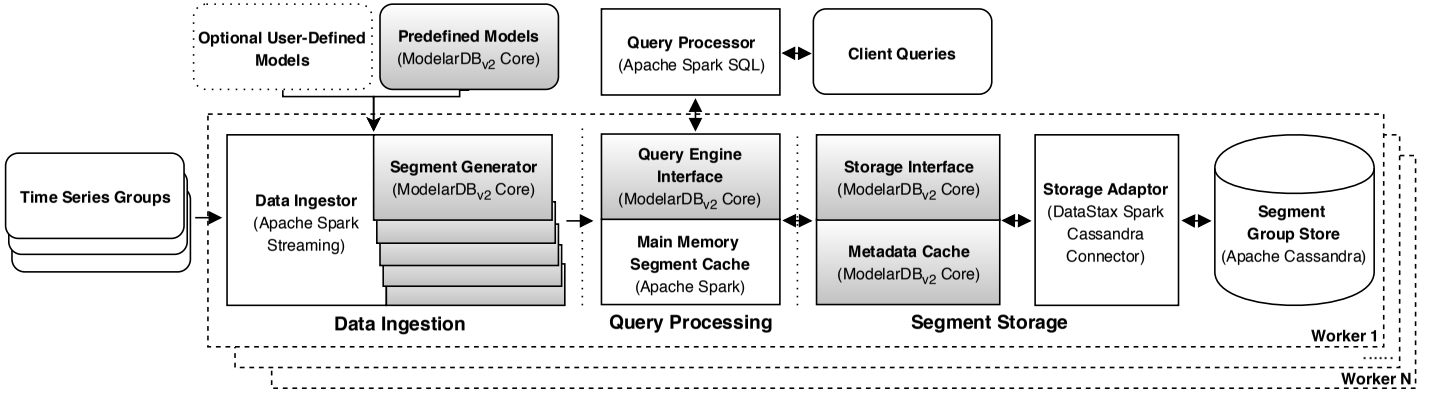

Figure C.3: Architecture of ModelarDB$_{v2}$ workers. Time series are grouped using only metadata and user hints before workers start ingesting

architecture. ModelarDB$_{v2}$ implements MMGC by adding a *Partitioner* to the master node and by changing ModelarDB$_{v1}$'s components [7]. The Partitioner groups correlated time series using metadata (source and dimension hierarchy), as historical data might not be available and due to the large overhead of discovering correlation in time series ingested by distributed workers. Discovering correlation through data mining is thus considered an orthogonal problem. Last, to prevent data skew the Partitioner partitions the set of time series groups into subsets SSG and distributes them across the cluster so the numbers of data points received per second for each thread are as close to equal as possible. The partitioning method used is based on [11], and minimizes $\max_{SG_1 \in SSG}(data\_points\_per\_minute(SG_1)) - \min_{SG_2 \in SSG}(data\_points\_per\_minute(SG_2))$. During ingestion the system automatically selects an appropriate model type for each dynamically sized sub-sequence of each time series group. Three model types, extended to support MGC, are included in ModelarDB$_{v2}$ Core: the constant PMC-Mean model (PMC-Mean) [12], the linear Swing model (Swing) [10], and the lossless compression algorithm for floating-point values proposed for the TSMS Gorilla (Gorilla) [13]. Users can optionally add model types through an API without recompiling ModelarDB$_{v2}$. For query processing ModelarDB$_{v2}$ uses SQL and expands the *Segment View* and *Data Point View* proposed for ModelarDB$_{v1}$ [7]. The Segment View allows aggregates to be executed on segments, e.g., `SUM` on a linear model uses constant time, while queries on the Data Point View are executed on reconstructed data points.

The architecture of the worker nodes in ModelarDB$_{v2}$ is split into three sets of components as shown in Figure C.3. In the figure each component is annotated with the software providing that functionality and components modified for ModelarDB$_{v2}$ have a gray gradient. *Data Ingestion* ingests time series and constructs models within a user-defined error bound; *Query Processing* caches recently constructed and queried segments and processes queries at either the segment or data point level; *Segment Storage* provides a uniform interface with predicate push-down for the persistent segment group store.

3.2 Ingestion and Representation of Gaps

The multi-model ingestion method used by ModelarDB$_{v2}$ for both bounded and unbounded time series is shown in Figure C.4, and compresses each group of time series

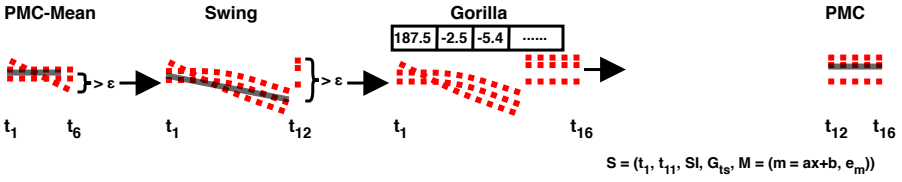

Figure C.4: Multi-model ingestion as performed by ModelarDB$_{v2}$

using a user-configurable list of model types. While users optionally can add additional model types, we use the three model types included in ModelarDB$_{v2}$ in the following example. At t_1 the first group of data points is received and added to a buffer, then the first model type (PMC-Mean) fits a model to them. At each SI the current model is updated if possible within the error bound so that it also represents the new group of data points. All models included in ModelarDB$_{v2}$ Core are incrementally updated instead of fitting a model to all buffered data points at each SI. At t_6 a group of data points is received that the current model cannot represent without exceeding the error bound ϵ for other data points in the current segment. Therefore, the model type is replaced with the next in the list (Swing). If possible, a new model is fitted to all buffered data points and ingestion continues, otherwise the next model type is used. For model types using lossless compression, e.g., Gorilla, the model is limited by a user-configurable length instead of an error bound. At t_{16} the last model type in the list (Gorilla) cannot fit a model to the received group of data points. Then the model providing the highest compression ratio (a model of type Swing in this example) is temporarily cached in memory and stored persistently on disk, the data points it represents cleared from the buffer, and the ingestion restarted from the first model type with the data points not yet represented. ModelarDB$_{v2}$ stores each group of time series as dynamically sized segments, with each segment represented by the model type providing the best compression within an error bound (possibly zero). If none of the model types can represent any of the buffered data points, e.g., a linear function requires two values, a fallback model type is used to store the raw buffered values. To simplify management of gaps and improve filtering during query processing, the start time and end time are stored for each segment. In addition, segments are stored disconnected (no overlapping data points) to increase the compression compared to connected segments [3, 4].

If a gap occurs ModelarDB$_{v2}$ creates a new segment and stores Tids in the segment as shown in Figure C.5. As the group shown consists of three time series a model is fitted to three values at t_1. At time $t_s + SI$, a gap occurs in TS_2 and a new model is fitted to the values from only two time series. To indicate that the model created only represents a subset of the time series, the Tids of the time series not represented are

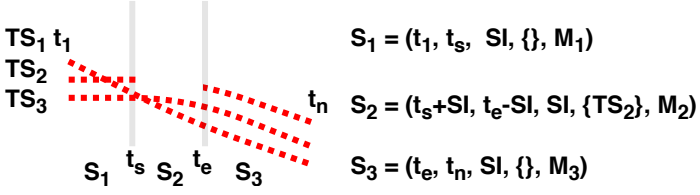

Figure C.5: Creating segments to not store gaps as timestamps

stored in the segment, see S_2. When data points are received from all time series again, the process is repeated, see t_e and S_3. Thus, a segment represents data points for a static number of time series. This method is used as it simplifies implementation of user-defined model types as ModelarDB$_{v2}$ manages gaps.

3.3 Storage Schema

The storage schema used by ModelarDB$_{v2}$ to support MMGC is shown in Figure C.6. The `Time Series` table stores `SI`, a scaling constant `Scaling` (C_{TS} in Definition C.10) and denomalized user-defined dimensions for each time series, with each identified by a `Tid`. Only the `SI` is mandatory. The `Gid` is the group a time series has been partitioned into and is computed by ModelarDB$_{v2}$ using user hints. `Scaling` is a constant that is applied to values during ingestion and query processing. With a scaling constant correlated time series with different values can be compressed together. The `Model` table maps a `Mid` to the Java Classpath of that model type. Last, the `Segment` table contains all ingested data points as dynamically sized segments.

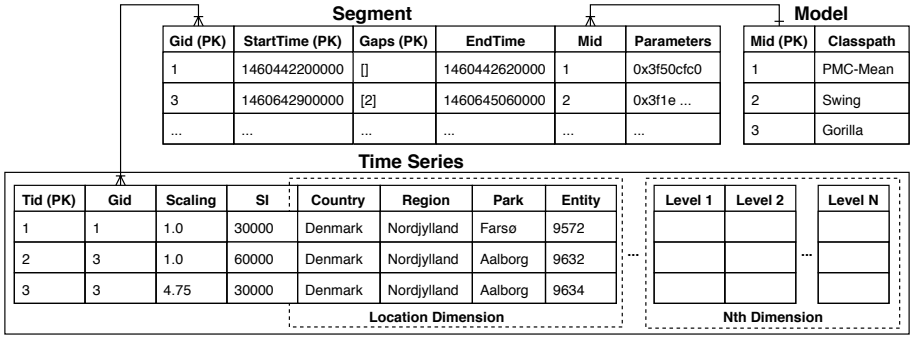

Figure C.6: Schema for storing time series groups using segments

In data warehouse terms, the `Segment` table functions as a fact table with new segments continuously appended during ingestion. Its primary key includes `Gaps` to prevent duplicate keys due to the dynamic splitting described in Section 4.2. The user-defined dimensions are stored denormalized as part of the *Time Series* table. However, no explicit time dimension is required as aggregate queries in the time dimension can be computed efficiently using only `StartTime` and `EndTime` as described in Section 6.3. For Cassandra two modifications are made to the general schema. First, to more efficiently support predicate push-down, the primary key for Segment is changed to `Gid, EndTime, Gaps` [7]. The values in `Gaps` are stored as 64-bit integers with each bit representing if a gap has occurred for that time series in the group. Second, as the column `StartTime` is not used for indexing, it is changed so the size of the segment is stored instead to save space. The `StartTime` can be efficiently recomputed as `StartTime = EndTime - (Size × SI)` [7].

4 Partitioning of Time Series

4.1 Partitioning of Correlated Time Series

To provide the benefit of model-based storage and query processing while ensuring low latency, models must be fitted online [7]. However, in a distributed system, time series compressed together should be ingested on one node to prevent excessive network traffic from limiting the scalability of the system. So, to prevent migration of data in the cluster, the time series must be partitioned based only on metadata or previously collected data. Using existing methods, time series can be partitioned into correlated groups based on historical data [14–17]. However, as historical data might not exist and even a small data set of only 50,000 time series creates $\binom{50{,}000}{2} \approx 1.25 \times 10^9$ pairs of possibly very large time series to compare for correlation, simply computing what time series are correlated quickly becomes infeasible. Thus, we focus on the problem of defining correlation using only metadata by proposing a set of primitives allowing users to specify correlation between time series directly or based on their dimension hierarchy. By providing a diverse set of primitives for describing correlation, users

Algorithm C.1 Group time series using the primitives

```
1:  Let 𝕋𝕊 be the list of time series.
2:  Let 𝔻 be the dimensions for all time series.
3:  Let Correlations be a list of correlation clauses.
4:
5:  TSG ← createSingleTimeSeriesGroups(𝕋𝕊)
6:  for clause ∈ Correlations do
7:      groupsModified ← true
8:      while groupsModified do
9:          groupsModified ← false
10:         for i ← 1 to |TSG| do
11:             for j ← i + 1 to |TSG| do
12:                 (TSG_1, TSG_2) ← getPairAt(TSG, i, j)
13:                 if correlated(clause, TSG_1, TSG_2, 𝔻) then
14:                     TSG[i] ← mergeGroups(TSG_1, TSG_2)
15:                     removeGroupAfterLoop(TSG_2, TSG)
16:                     groupsModified ← true
17:                 end if
18:             end for
19:         end for
20:     end while
21: end for
22: return Groups
```

are free to only use their domain knowledge or analyze historical data if the needed data and compute resources are available, making computing correlation from data an orthogonal problem [14–17].

Our primitives can be combined to efficiently describe correlation for data sets with different quantities of time series and dimensions. They are specified in ModelarDB$_{v2}$'s configuration file as `modelardb.correlation` clauses. All primitives in a clause are combined with an `AND` operator and multiple clauses are combined with an `OR` operator. Using these user hints ModelarDB$_{v2}$ creates groups of correlated time series to be ingested together. The primitives allow correlation to be specified as sets of time series, members that must be equal, levels for which members must be equal, or the *distance* between all of the dimensions (see below). Grouping is performed as shown in Algorithm C.1. First, a group is initialized per time series in Line 5. In Line 6–16, for each user-defined correlation *clause*, groups are merged until a fixpoint is reached in the number of groups. The function *correlated* in Line 13 evaluates the primitives in the *clause* and ensures that all time series in both groups are correlated before a pair of groups are merged. As the clauses are applied in the order they are defined, users control their priority. In essence, Algorithm C.1 computes cliques (as correlations are non-transitive) in a graph with time series as vertices and *correlated* defining the edges. However, as edges are dynamically defined as correlated groups are found, all possible edges need not be materialized.

To specify that specific time series are correlated their source (file or socket) must be provided, e.g., `4LR9a_Temperature.gz 4LR9b_Temperature.gz`. For correlated time series that do not contain similar values, a scaling constant can be added per time series. While this allows precise control over the groups, it quickly becomes too time consuming as the number of time series increases. The other primitives are based on the notion that time series correlation can be derived from their dimensions. For example, temperature sensors in close proximity likely produce similar values. The similarity of a dimension for two groups can be computed as their LCA level. This is the lowest level in a dimension where all time series in the two groups have equal members starting from $\top$. An example can be seen in Figure C.7.

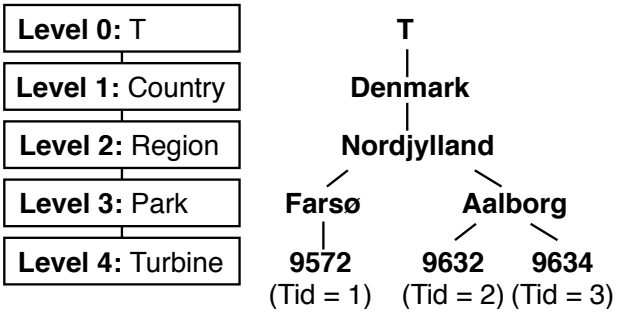

Figure C.7: An example Location dimension for wind turbines where the LCA for $Tid = 2$ and $Tid = 3$ is the level *Park*

To specify correlation based on members, the user must provide either a triple consisting of a dimension, a level, and a member or a pair with a dimension and an LCA level. The triple `Measure 1 Temperature`, e.g., specifies that time series sharing the member `Temperature` at level one of the `Measure` dimension are correlated. The pair `Location 2` states that if the LCA level is equal to or higher than two for the `Location` dimension, the time series are correlated. Zero specifies that all levels must be equal, and a negative number n that all but the lowest $|n|$ levels must equal. When specifying a scaling constant for many time series, it can be defined for time series with a shared member as a 4-tuple containing dimension, level, member, and scaling constant. These primitives are appropriate for a data set with few dimensions but many time series.

For data sets with both a large number of time series and dimensions, the user can specify correlation as the *distance* $[0.0; 1.0]$ between dimensions. The intuition is that time series with much overlap between their members are correlated. For example, for the Location dimension in Figure C.7, time series sharing members at the Turbine level are more likely to be correlated than if they only share members at the Country level. The distance 0.0 specifies that all members must match for the time series to be grouped, and 1.0 that all time series should be grouped. Values in-between specify different degrees of overlap. The user can inject domain knowledge by changing the impact of a dimension using a *weight* for which the default value is 1.0. Distances above 1.0 due to user-defined weights are reduced to 1.0. For distance-based correlation the rule of thumb is to use the lowest non-zero value such that only time series with many overlapping members are grouped. The lowest distance can be calculated as $(1/\max(\mathit{Levels}))/|\mathbb{D}|$ where *Levels* is the set of levels in each dimension and $\mathbb{D}$ is the dimensions.

To evaluate each type of primitive, *correlated* uses different functions. The function for determining correlation using a distance threshold is shown in Algorithm C.2. The definition of the remaining functions follows directly from the description of the primitives. In Line 11 in Algorithm C.2 *distance* is computed as $(\mathit{height} - \mathit{ancestor})/\mathit{height}$ to reduce the impact of groups with equivalent members only at the top of the hierarchy. In Line 12 the dimension's distance is multiplied by its user-defined weight before being added to the accumulator. In Line 14–15 the distance between the two time series groups is normalized to the range $0.0 - 1.0$ and compared to the user-defined *threshold* to determine if the two time series groups are correlated. As an example, for the Location dimension shown in Figure C.7, the normalized distance between the time series with $\mathit{Tid} = 2$ and $\mathit{Tid} = 3$ can be computed as $1 \times ((4-3)/4) = 0.25$.

4.2 Dynamically Splitting Groups

As events can change the values of a time series, e.g., a wind turbine might be damaged, ModelarDB$_{v2}$ splits a group if its time series become temporarily uncorrelated. A split can be performed after emission of a segment as it indicates that the structure of a time series has changed so the next data point would exceed the error bound. An example is

Algorithm C.2 Use of distance to indicate correlation

```
1: Let clause be a user-defined correlation clause.
2: Let TSG_1, TSG_2 be time series groups.
3: Let 𝔻 be the dimensions for all time series.
4:
5: threshold ← getThreshold(clause)
6: sumDistance ← 0
7: for d ∈ 𝔻 do
8:     ancestor ← computeLCA(TSG_1, TSG_2, d)
9:     height ← getHeight(d, 𝔻)
10:    weight ← getWeight(d, 𝔻)
11:    distance ← (height − ancestor) / height
12:    sumDistance ← sumDistance + weight × distance
13: end for
14: normalized ← sumDistance / length(𝔻)
15: return min(normalized, 1.0) <= threshold
```

shown in Figure C.8. While ModelarDB$_{v2}$ discards data points emitted as segments, the entire time series are included in the example to show how they change over time. At t_m the group is ingested using the Segment Generator SG_0. At t_n the time series in the group are no longer correlated and segments with poor compression are emitted. The group is then split into two and ingestion continues with SG_1 and SG_2. SG_0 is used to synchronize ingestion for the splits to simplify joining and joins the split groups if they again become correlated. Then at t_i the group ingested by SG_1 is split causing each time series to be ingested separately.

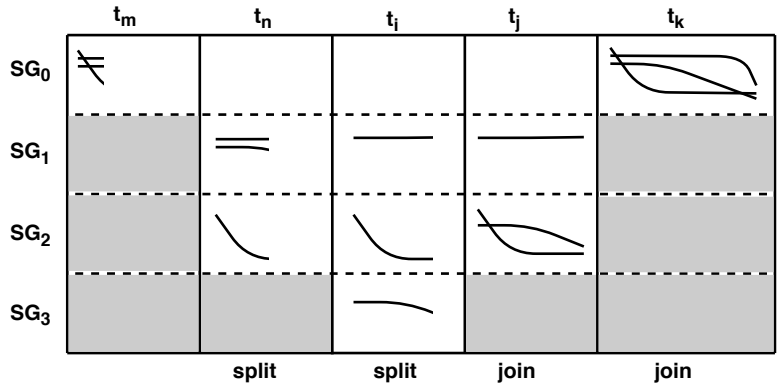

Figure C.8: Ingestion of *TSG* with dynamic splitting and joining

To minimize the number of non-beneficial splits and the overhead of determining when to split, ModelarDB$_{v2}$ uses two heuristics: poor compression ratio and the error

Algorithm C.3 Potentially splitting groups of time series temporarily

```
1:  Let TSG be a time series group.
2:  Let ffl be the user-defined error bound.
3:  Let Buffer be the data points buffered for TSG.
4:
5:  Splits ← createSet()
6:  while notEmpty(TSG) do
7:      TS_1 ← getTimeSeries(TSG)
8:      TSG_n ← createGroup()
9:      for TS_2 ∈ TSG do
10:         DP_1 ← dataPoints(TS_1, Buffer)
11:         DP_2 ← dataPoints(TS_2, Buffer)
12:         if allWithinDoubleBound(DP_1, DP_2, ffl) then
13:             addTimeSeriesToGroup(TS_2, TSG_n)
14:             removeTimeSeriesFromGroup(TS_2, TSG)
15:         end if
16:     end for
17:     sg ← createSegmentGenerators(TSG_n)
18:     addSegmentGeneratorToSet(sg, Splits)
19: end while
20: return Splits
```

between ingested data points. First, ModelarDB$_{v2}$ checks if the compression ratio of a new segment is below a user-configurable fraction of the average (default is $1/10$). If the compression ratio is lower and data points are currently buffered, Algorithm C.3 is executed. The algorithm groups time series if their buffered data points are within twice the user-defined error bound (2ϵ), using the error metric of the included model types (Line 9–14). Thus, groups of size one to the size of the group TSG can be created. In Figure C.8 two groups are created at both t_n (SG_1 and SG_2) and t_i (SG_1 and SG_3). Time series in a gap are grouped together.

To demonstrate joining we continue with the example in Figure C.8. At t_j two time series become correlated again and are merged into one group. Last, at t_k all the time series are correlated again so SG_0 takes over ingestion. The algorithm for restoring a split group is shown in Algorithm C.4 and is similar to Algorithm C.3. However, when joining groups it is only necessary to compare one time series from each as a group consists of correlated time series (otherwise a split would have occurred). To simplify joining groups Algorithm C.4 is only potentially executed at the end of each SI so all groups have received data points for the same time period. As a segment being emitted indicates a significant change of the values ingested by a group, a split group is only marked for joining after emitting a number of segments. The number of segments that must be emitted are doubled after each attempt to join a split group to

Algorithm C.4 Potentially restoring a split group of time series

```
1: Let JoinCandidates be groups marked for joining.
2: Let Splits be the set of all split time series groups.
3: Let ffl be the user-defined error bound.
4: Let Buffer be the data points buffered for Splits.
5:
6: Joined ← createSet()
7: while notEmpty(JoinCandidates) do
8:     TSG_1 ← getGroup(JoinCandidates)
9:     TS_1 ← getFirstTimeSeries(TSG_1)
10:    DPR_1 ← getReverseDataPoints(TS_1, Buffer)
11:    for TSG_2 ∈ Splits do
12:        TS_2 ← getFirstTimeSeries(TSG_2)
13:        DPR_2 ← getReverseDataPoints(TS_2, Buffer)
14:        shortest ← computeShortestLength(DPR_1, DPR_2)
15:        length ← withinDoubleBound(DPR_1, DPR_2, ffl)
16:        if shortest > 0 and shortest = length then
17:            TSG_1 ← mergeGroups(TSG_1, TSG_2)
18:            removeGroupFromSet(TSG_2, Splits)
19:            removeGroupFromSet(TSG_2, JoinCandidates)
20:        end if
21:    end for
22:    sg ← createSegmentGenerators(TSG_1)
23:    addGroupToSet(sg, Joined)
24: end while
25: return Joined
```

reduce the overhead of joining. The intuition is that each failed attempted at joining further indicates that the current splits are preferable.

5 Model Extensions

To benefit from MMGC a set of models is required. However, as most model-based compression methods for time series are designed for individual time series [1, 2], existing model types must be extended to support MGC to be used with ModelarDB$_{v2}$. We first describe a simple method for extending any model type with MGC by storing multiple models per segment, and then two model type specific approaches that allow a group to use one model per segment.

5.1 Multiple Models per Segment

A baseline method for adding MGC support to any model type is to split the data points received and fit separate models to them. The resulting models are then stored together in one segment, without any changes to the model type or model. However, to store multiple models in one segment, each representing the values of different time series, the models must represent values for the same time interval. This is intuitively simple to ensure by verifying that each model will not exceed the error bound before fitting them to a new data point. However, this is unnecessary as explained next.

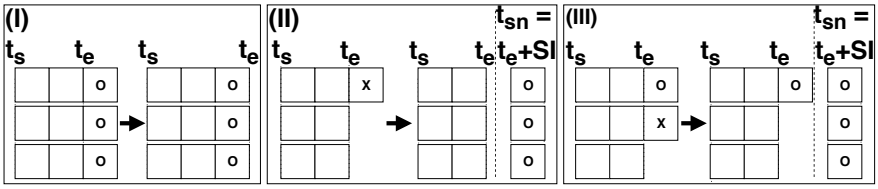

Figure C.9: Fitting models, (O) indicates it can fit the value, (X) that it cannot, and a dashed line that a new segment is created

Three cases can occur when models are fitted as shown in Figure C.9. For case (I) all models can be fitted to the data point received from their respective time series. The opposite occurs in case (II) as the first model cannot be fitted to the data point it received within the user-defined error bound. For case (I) and case (II) it is trivial to see that all resulting models represent the same time interval. In case (III), the first model can be fitted to the data point received, however, the second model cannot. As the models in the segment no longer represent the same time interval, the end time t_e of the segment is simply not incremented to $t_e + SI$. As each model represents all previously ingested values, the end time t_e of a segment can be safely reduced in increments of SI until $t_s > t_e$. For models where the number of parameters depends on the number of data points fitted, e.g., models of type Gorilla, the leftover parameters should be deleted. Afterwards new models are fitted to the next set of data points. While the use of n models stored in one segment reduces the amount of duplicate metadata from n copies to one and is simple to implement, it does not reduce the storage required for the values. To further improve compression, each model must represent multiple time series using one set of parameters.

5.2 Single Model per Segment

To fully exploit MMGC it is necessary to use multiple model types which can represent many time series in a single model. We found that the model types used by ModelarDB$_{v1}$ can be extended to efficiently compress a group of time series using a single model based on two general ideas. For model types using lossless compression, e.g., Gorilla, values from multiple time series should be stored in time ordered blocks. This allows exploitation of both temporal correlation and correlation across time series

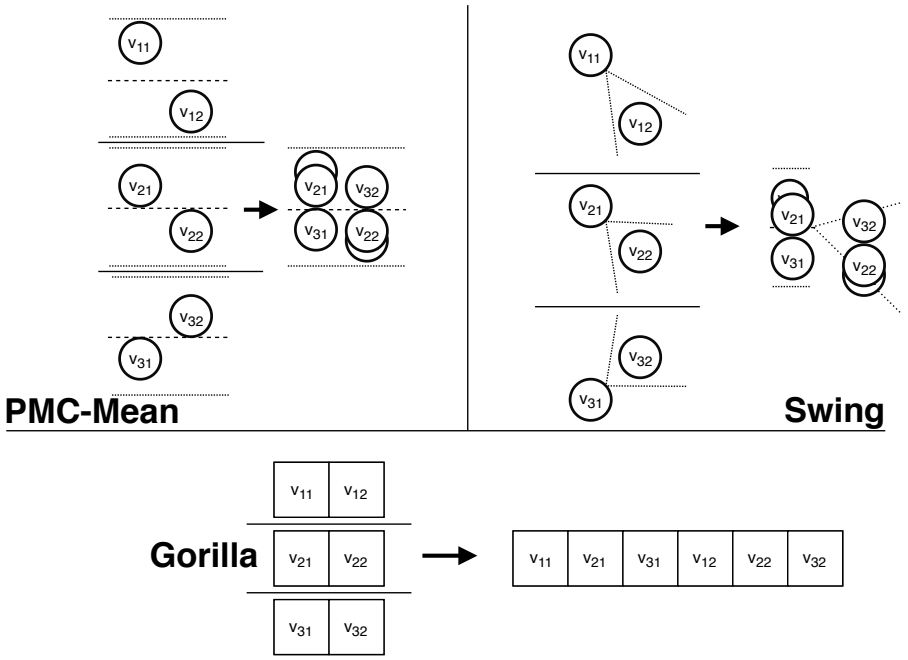

Figure C.10: Methods for modifying model types to support MGC. For lossy models the x-axis represents the time and the y-axis represents the value. Full lines separate the models, dashed lines show the average value for PMC-Mean, and dotted lines show the error bound.

at each SI. For model types that fit models to data points by verifying that values are within an upper and lower bound according to the uniform norm, e.g., PMC-MR and Swing, only the data points with the minimum and maximum value for each SI are required to validate the bounds. As a result, the set of values $V \subset \mathbb{R}$ for a timestamp t can be reduced to a range of values represented by the 3-tuple $v_a = (t, \min(V), \max(V))$. We now show in detail how MMGC can be performed efficiently using the model types provided in ModelarDB$_{v2}$ Core.

For PMC-Mean, the set of values V from a group of time series is represented as $avg(V)$ within the error bound ϵ of $\min(V)$ and $\max(V)$. As a result, PMC-Mean requires no changes as the model type only tracks the current minimum, maximum and average value. See PMC-Mean in Figure C.10. Swing produces a linear function that passes through the initial value and then maintains the upper bound and lower bound required for it to represent the values within ϵ. The initial value can be computed using PMC-Mean. The values for each SI are then added one data point at a time. See Swing in Figure C.10. For Gorilla, values from data points with the same timestamp are stored in blocks. As the time series in a group are correlated, $n - 1$ values in each block will have only a small delta compared to the first value and only require a few bits to encode. See Gorilla in Figure C.10. To demonstrate the benefit of our MGC extensions

we compress three real-life time series representing the temperature of co-located wind turbines. Compared to using only MMC, enabling MMGC in ModelarDB$_{v2}$ reduces the storage required by 28.67% with a 0% error bound, by 26.63% for 1%, by 33.16% for 5%, and by 43.50% for 10%.

6 Query Processing

6.1 Query Interface

As a model M can reconstruct the data points it represents within the error bound, queries can be executed on these. However, many aggregate queries can be answered from models, e.g., for a constant or linear function `MIN`, `MAX`, `SUM` and `AVG` can computed in constant time [7]. For these model types, aggregate queries can be answered in linear time in the number of models instead of the number of data points. As a model represents at least one data point, but typically many more, computing aggregates from models greatly reduces query time as shown in Section 7. To support this, ModelarDB$_{v2}$ provides a Segment View with the schema `(Tid int, StartTime timestamp, EndTime timestamp, SI int, Mid int, Parameters blob, Gaps blob, Dimensions)` and a Data Point View with the schema `(Tid int,TS timestamp,Value float, Dimensions)`. `Dimensions` represents the columns storing the denormalized user-defined dimensions. The user-defined dimensions are cached in-memory and joined with segments and data points when required during query processing using an in-memory hash-join. Using the Segment View, ModelarDB$_{v2}$ supports executing aggregate queries on segments using user-defined aggregate functions, which for simple aggregates are suffixed with `_S`, e.g., `MAX_S`. Functions performing aggregation in the time dimension are suffixed with aggregate and level in the time hierarchy, e.g., `CUBE_AVG_HOUR`. All aggregate functions divide the result by the scaling constant of each time series as part of the *iterate* step. Queries performing aggregation using the user-defined dimension hierarchy can be executed using a `GROUP BY` on the appropriate columns in the Segment View, reducing the problem to computing a simple aggregate on segments. As a result, in this section we describe how simple aggregate queries and multi-dimensional aggregate queries in the time dimension can be executed on segments for distributive and algebraic functions [18].

6.2 Aggregate Queries

To allow queries to be expressed at the time series level instead of the time series group level, a mapping between *Gid*s and *Tid*s is performed as part of query processing using the identifiers stored in the time series table shown in Figure C.6. As a result, queries provided by the user and the result returned from ModelarDB$_{v2}$ only reference *Tid*s, with *Gid*s being utilized for predicate push-down so the segment group store only needs to index one id per segment. While ModelarDB$_{v1}$ only supports predicate push-down

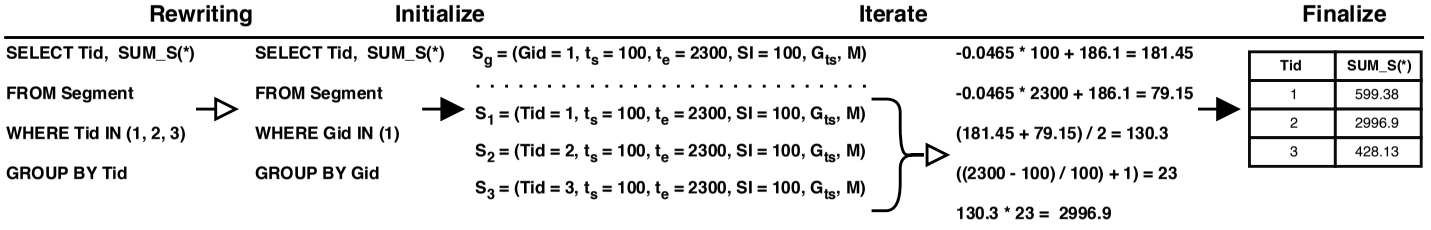

Figure C.11: An aggregate performed on the linear model $-\mathbf{0.0465t} + \mathbf{186.1}$ representing a group of three time series

for `Tid`, `StartTime` and `EndTime` [7], ModelarDB$_{v2}$ also supports predicate push-down for user-defined dimensions by rewriting instances of a member in the `WHERE` clause to the *Gid*s of the groups that include time series with that dimensional member.

An example of a simple aggregate query executed on the Segment View is shown in Figure C.11. First, all *Tid*s in the query are rewritten to their corresponding *Gid*s so users only need to know the *Tid*s. Then for each segment the aggregate function specified in the query is executed. The aggregate function also applies the scaling constant. After the aggregate has been computed for all segments, the final aggregate is computed from the intermediate results, e.g., by computing an average. The pseudo-code for executing aggregate queries using the Segment View is shown in Algorithm C.5.

Algorithm C.5 Execution of simple aggregates on the Segment View

```
1:  Let predicates be the WHERE clause of the SQL query.
2:  Let groups be a mapping from Gid to Tid and reverse.
3:  Let members be a mapping from Members to Gid.
4:  Let initialize be the function preparing storage for the results.
5:  Let iterate be the segment aggregation function.
6:  Let finalize be a function for aggregating results.
7:
8:  predicates ← rewriteQuery(predicates, groups, members)
    ▷ Executed on workers with the result sent to the master
9:  result ← initialize()
10: segments ← retrieveSegments(predicates, groups)
11: for segment ∈ segments do
12:     result ← iterate(segment, result)
13: end for
14: results ← mergeResults()
15: return finalize(results)
```

In Line 8 the SQL query is rewritten to query segments in terms of time series groups by replacing the *Tid*s and members in the SQL queries `WHERE` clause with the matching *Gid*s at the master before the query is sent to each worker node (Rewriting). In Line 9–10 each worker node initializes memory for storing the intermediate values and retrieves relevant segments from its data store (Initialize). Then for each segment, in Line 11–12 the aggregate function passed as argument is executed on each segment (Iterate). Finally, in Line 15, to support both distributive and algebraic functions, computation that must be performed on the intermediate results is performed (Finalize).

6.3 Aggregation in the Time Dimension

As the schema shown in Figure C.6 stores the start time and end time as part of each segment, aggregates in the time dimension can be computed using only the segment

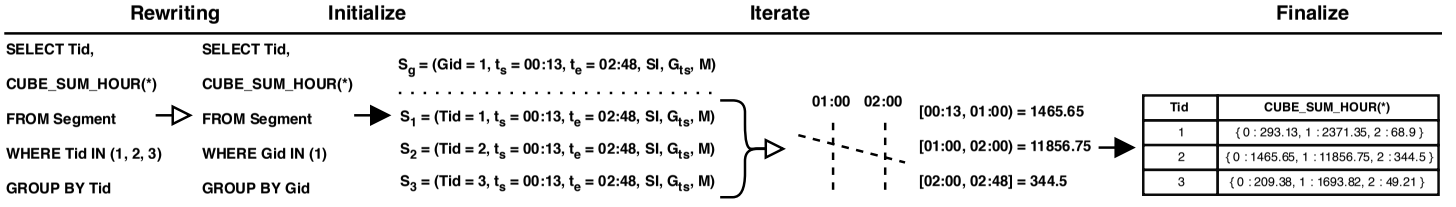

Figure C.12: Aggregation in the time dimension on a linear model representing a group of three time series

table without an expensive join with a separate time dimension. An example of an aggregation in the time dimension using segments is shown in Figure C.12. The query computes the sum per hour for the time series with $Tid = 1$, $Tid = 2$ and $Tid = 3$, using the function `CUBE_SUM_HOUR` to compute the result efficiently on segments rather than on data points. After rewriting the query, the aggregate is computed for the interval from $t_s =$ 00:13 until 01:00 which is the next timestamp delimiting two aggregation intervals. Afterwards, the aggregate is computed for the interval from 01:00 until 02:00. Last, the aggregate is computed for the interval from 02:00 to and including $t_e =$ 02:48. The last value is computed with an inclusive end time as ModelarDB$_{v2}$, to increase the compression ratio, does not store connected segments [7].

The pseudo-code for executing aggregate queries in the time dimensions using the Segment View is shown in Algorithm C.6. The algorithm follows the same structure as Algorithm C.5. First, in Line 9 the query is rewritten in terms of *Gid*s instead of *Tid*s and members (Rewriting), before each worker initializes memory for storing the intermediate results and retrieves relevant segments in Line 10–11 (Initialize). In Line 12–24 the algorithm iterates over each segment and computes an intermediate aggregate for each of the requested time intervals (Iterate). Last, the final result is computed and returned in Line 28 to support distributive and algebraic functions (Finalize).

7 Evaluation

7.1 Overview and Evaluation Environment

We compare ModelarDB$_{v2}$ to state-of-the-art big data formats/systems used in industry (Apache ORC and Parquet in HDFS, Apache Cassandra), and the TSMS InfluxDB due to its popularity in industry (together termed *industry formats*), as well as ModelarDB$_{v1}$ [7]. Apache Spark is used for query processing, except for InfluxDB as the open-source version is single-node. To evaluate MMGC in isolation from ModelarDB$_{v2}$'s more mature implementation, we experiment with both MMGC enabled (ModelarDB$_{v2}$+G) and disabled (ModelarDB$_{v2}$−G). We expect ModelarDB$_{v2}$−G to win when queries only require a single/few time series per group, e.g., simple aggregate and point-range queries, while ModelarDB$_{v2}$+G performs better for compression, ingestion and queries on entire groups. Users can enable MMGC depending on their use-case. Our seven node local cluster is described in Table C.1: it has one *master* node being *Primary HDFS NameNode*, *Secondary HDFS NameNode* and *Spark Master*, and six *worker*s being *Cassandra Node*s, *HDFS Datanode*s, and *Spark Slave*s. Default configurations are used as much as possible. Changed values are shown in Table C.1 and selected as they work well with the hardware and data sets used. For parameters we change during the experiments, all values are shown, with the default in bold. Spark memory is limited by *spark.driver.memory* and *spark.executor.memory*, set at the max values where Cassandra or HDFS do not crash. We use the extended model types from Section 5.2 PMC-Mean [12], Swing [10], and Gorilla [13]. We only run the

Algorithm C.6 Rewrite and execution of aggregate queries with a roll-up in the time dimension using the Segment View

```
1: Let predicates be the WHERE clause of the SQL query.
2: Let groups be a mapping from Gid to Tid and reverse.
3: Let members be a mapping from Members to Gid.
4: Let level be the roll-up level in the time hierarchy.
5: Let initialize be the function preparing storage for the results.
6: Let iterate be the segment aggregation function.
7: Let finalize be a function for aggregating results.
8:
9: predicates ← rewriteQuery(predicates, groups, members)
   ▷ Executed on workers with the result sent to the master
10: result ← initialize()
11: segments ← retrieveSegments(predicates, groups)
12: for segment ∈ segments do
13:     originalEndTime ← extractEndTime(segment)
14:     startTime ← extractStartTime(segment)
15:     endTime ← ceilToLevel(startTime, level)
16:     if originalEndTime <= endTime then
17:         result ← aggregateInterval(iterate, segment,
                startTime, originalEndTime, result)
18:     else
19:         while endTime < originalEndTime do
20:             result ← aggregateInterval(iterate, segment,
                    startTime, endTime, result)
21:             startTime ← endTime
22:             endTime ← updateForLevel(startTime, level)
23:         end while
24:         result ← aggregateInterval(iterate, segment,
                startTime, originalEndTime, result)
25:     end if
26: end for
27: results ← mergeResults()
28: return finalize(results)
```

necessary software and disable replication. Disk space usage is measured with all data on one node using *du*, and execution time using *time* or *System.currentTimeMillis()*. Graphs show the number of workers on the x-axis labels and use striped bars for ModelarDB$_{v2}$+G.

Table C.1: Evaluation environment

Hardware	
Processor	Intel Core i7-2620M
Memory	8GiB of 1333 MHz DDR3
Storage	7,200RPM Hard-Drive
Network	1 Gbit Ethernet

Software	
Ubuntu GNU/Linux	v16.04 LTS on ext4
ModelarDB	v2.0
— *Model Error Bound*	0%, 1%, 5%, **10%**
— *Model Length Limit*	50
— *Dynamic Split Fraction*	10
— *Bulk Write Size*	50,000
InfluxDB	v1.4.2
InfluxDB-Java	v2.10
— *Batch Size*	50,000
Apache Hadoop	v2.8.0
Apache Spark	v2.1.0
— *spark.driver.memory*	4 GiB
— *spark.executor.memory*	3 GiB
— *spark.streaming.unpersist*	false
— *spark.streaming.stopGracefullyOnShutdown*	true
— *spark.sql.orc.filterPushdown*	true
— *spark.sql.parquet.filterPushdown*	true
Apache Cassandra	v3.9
— *commitlog_segment_size_in_mb*	128 MiB
— *batch_size_warn_threshold_in_kb*	5 MiB
— *batch_size_fail_threshold_in_kb*	50 MiB
DataStax Spark Cassandra Connector	v2.0.3

$\text{ModelarDB}_{\text{v2}}$ is designed for unbounded time series, but our comparison with industry formats uses two static real-life data sets. For ingesting unbounded time series, we continuously replicate the values of a static time series using a C program. The industry formats use the schema: `(Tid int, TS timestamp, Value float, Dimensions)`. `timestamp` is each format's native timestamp type. Cassandra uses

`(Tid,TS,Value)` as primary key, InfluxDB stores time series as one measurement with the `Tid` as a tag, and for ORC and Parquet we create a file per series stored on HDFS in a folder named `Tid=n` so Spark can prune by `Tid`.

7.2 Data Sets and Queries

Data Set "EP" This real-life energy trader data set consists of regular time series with gaps from energy production collected over 508 days, has $SI = 60$ seconds, has two dimensions: *Production: Entity* $\rightarrow$ *Type* and *Measure: Concrete* $\rightarrow$ *Category*, contains 45,353 time series, and uses 339 GiB as uncompressed CSV files.

Data Set "EH" This real-life data set consists of regular time series with gaps from wind parks. The data was collected by us with an approximate $SI = 100$ milliseconds using an OPC Data Access server running on Windows. A pre-processing step rounds timestamps to the nearest 100 milliseconds and removes data points with equivalent timestamps (due to collection limitations not present in a production setup). It has two dimensions: *Location: Entity* $\rightarrow$ *Park* $\rightarrow$ *Country* and *Measure: Concrete* $\rightarrow$ *Category*, contains 286 time series, and uses 582.68 GiB as uncompressed CSV files.

Queries For evaluation, we use small simple aggregate queries for interactive analysis (S-AGG), large scale simple aggregate queries for scalability (L-AGG), medium scale multi-dimensional aggregate queries for reporting (M-AGG), and point/range queries for extracting sub-sequences (P/R). Half of the S-AGG queries aggregate one time series while the rest `GROUP BY Tid` for five time series. The L-AGG queries aggregate the full data set where half `GROUP BY Tid`. M-AGG is multi-dimensional aggregates on measures of energy production. Half `GROUP BY` month and dimension, while the rest `GROUP BY` month, dimension and `Tid`. P/R is point/range queries with `WHERE` clauses on either `TS` or `Tid` and `TS`.

7.3 Experiments

Ingestion Rate First, we evaluate the ingestion rate when performing *bulk loading* without queries (B) on one node for a direct comparison with InfluxDB. For each format we ingest energy production measures from EP (3500 gzipped CSV files, 6.59 GiB) using *spark-shell* with its default parameters. Dimensions are read from a 6.7 MiB CSV file. For the industry formats, dimensions are appended to the data points from an in-memory cache. Second, we measure ModelarDB$_{v2}$ scalability on all six worker nodes using both bulk loading without queries (B) and *online analytics (O)* with aggregate queries executed on random time series using the Segment View during ingestion. ModelarDB$_{v2}$ uses a specialized ingestor on a single node, and Spark Streaming with a 5 sec micro-batch interval and one receiver per node on the cluster. Last, we evaluate the stability of ModelarDB$_{v2}$'s ingestion rate by ingesting an unbounded time series over a socket on one node for 2.5 days, measuring the time for each ingestion of the repeated time series.

The results are shown in Figure C.13. As expected, due to better compression

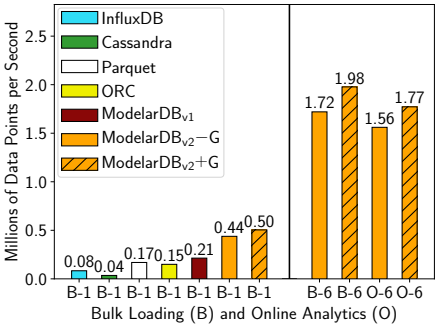

Figure C.13: Ingestion, EP

ModelarDB$_{v2}$+G provides a 2.38–12.5 times higher ingestion rate compared to the other formats. ModelarDB$_{v2}$+G is 2.38 times faster than ModelarDB$_{v1}$ due to a more mature implementation and the use of MMGC. On six workers ModelarDB$_{v2}$+G achieves a 3.96 times speedup for bulk loading and a 3.54 times speedup for online analytics. The ingestion rates are stable over long intervals: the average daily ingestion time only increases by 0.6% from the first to the last day. Temporary short-term decreases in ingestion speed seem to match JVM garbage collections, as expected. In summary, ModelarDB$_{v2}$ provides a stable ingestion rate that is higher than the other formats due to an efficient model-agnostic ingestion method and state-of-the-art compression while also supporting online analytics.

Effect of Error Bound and Grouping We evaluate ModelarDB$_{v2}$ compression using EP and EH. We ingest using 0%, 1%, 5%, and 10% error bounds and the best combination of correlation primitives found for each data set with our limited domain knowledge. For formats not supporting approximation, the error bound is 0%. For each data set we show storage and model types used, and the actual average error as $(\sum_{n=1}^{|DP|} |rv_n - av_n| / \sum_{n=1}^{|DP|} |rv_n|) \times 100$ where DP is the ingested data points, av_n is the nth approximated value and rv_n is the nth real value. As many EP time series are correlated MMGC should significantly reduce the storage required, while for the less-correlated EH series MMGC should only yield benefits with high error bounds. We compare our metadata-based grouping method with compressing EP and EH using groups of time series with equivalent min and max values. Last, we use instrumentation to determine the overhead of our algorithms for dynamically splitting (Algorithm C.3) and joining (Algorithm C.4) groups during ingestion.

EP results are shown in Figure C.14 with correlation `Production 0, Measure 1 ProductionMWh` as EP contains no location information but has multiple energy production measurements per entity. This creates 39,164 groups from 45,353 time series, with splitting and joining using only 0.74% of the total run-time in the worst case. Compared to industry formats ModelarDB$_{v2}$ uses up to 16.17 times less storage, while the highest average error is only 0.34% (10% error-bound). Surprisingly, ModelarDB$_{v2}$−G provides similar compression compared to ModelarDB$_{v1}$ despite

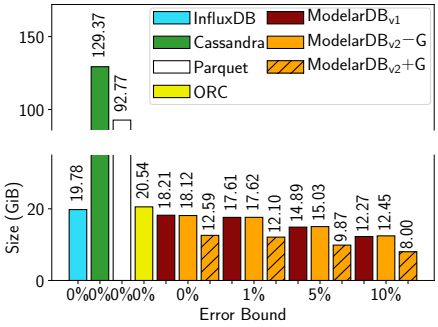

Figure C.14: Storage, EP

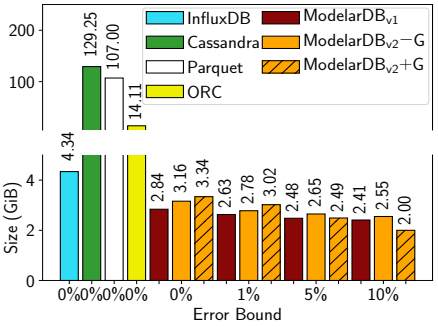

Figure C.15: Storage, EH

using the PMC-Mean constant model type, which is more restrictive but yields lower actual error compared to ModelarDB$_{v1}$'s PMC-MR [7]. Compared to ModelarDB$_{v2}$−G, ModelarDB$_{v2}$+G reduces the storage used by 1.44–1.56 times as the energy production measures compress well together. EH results are shown in Figure C.15 with correlation defined by the lowest distance (`0.16666667`) using our rule of thumb in Section 4.1. This creates 102 groups from 286 time series, with splitting and joining using only 0.32% of the total run-time in the worst case. ModelarDB$_{v2}$ reduces the storage required by up to 64.63 times compared to the other formats, while the highest average error is only 2.03% (10% error-bound). For low error-bounds ModelarDB$_{v2}$+G performs slightly worse than ModelarDB$_{v1}$, due to its more restrictive PMC-Mean model type [7]. Despite MMGC being nonapplicable for low error bounds on EH, ModelarDB$_{v2}$ always outperforms the industry formats and ModelarDB$_{v2}$+G uses 1.21 times less storage than ModelarDB$_{v1}$ for a 10% error-bound.

Figure C.16–C.19 show that all model types are used for both of the data sets. ModelarDB$_{v2}$+G adapts by representing more data points using Gorilla. This is expected as only one time series has to exhibit a constant or linear trend for PMC-Mean or Swing to be used with ModelarDB$_{v2}$−G, while all time series in a group must exhibit

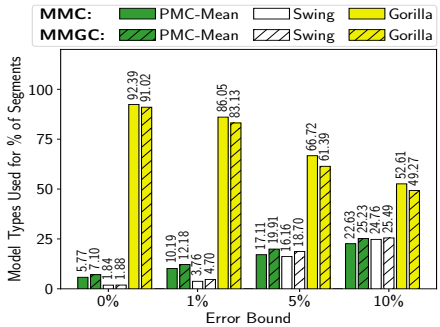

Figure C.16: Model types (in segments), EP

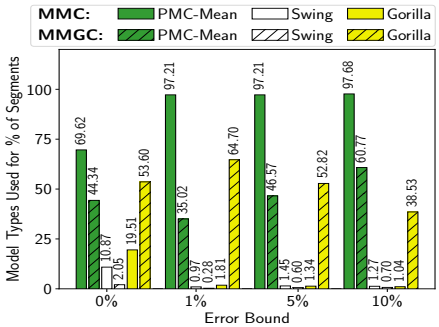

Figure C.17: Model types (in segments), EH

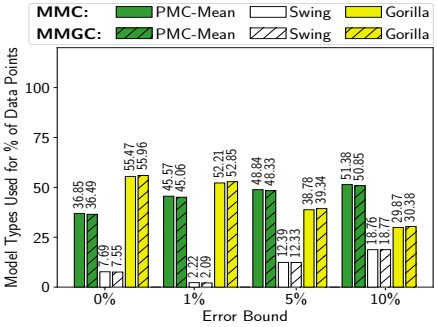

Figure C.18: Model types (in data points), EP

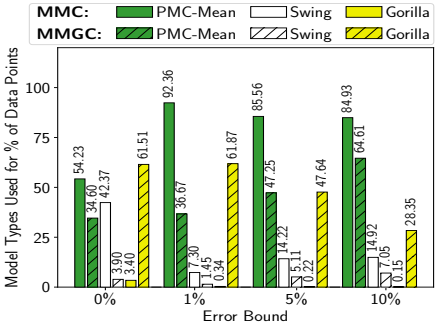

Figure C.19: Model types (in data points), EH

the same trend to efficiently use these two model types with ModelarDB$_{v2}$+G. When compressing EP using value-based grouping, 8,487 groups are created as time series with similar measures are grouped. As ModelarDB$_{v2}$ uses 64 bits to store gaps, a few groups larger than 64 time series were split. The storage required is 8.00–6.49 GiB (0–10% error bound). EP contains no *Location* dimension, which explains the difference in compression to our metadata-data based method. For EH, 187 groups are created, using 2.99–2.35 GiB (0–10% error bound), and thus value-based grouping performs similarly to our distance-based method. In both methods, time series with similar measures collected from the same area are grouped. While value-based groups in some cases provide better compression than our metadata-based method, it requires prior analysis of the full data set. Any correlations found by prior analysis can then be specified using our ModelarDB$_{v2}$ primitives. This also demonstrates that additional grouping methods can be added to ModelarDB$_{v2}$. In summary, ModelarDB$_{v2}$ yields better compression than existing formats by using model-based compression and dynamically selecting appropriate combinations of model types for each data set and error bound pair with both MMGC being enabled and disabled.

Effect of Distance We evaluate our distance-based grouping method by ingesting EP and EH with all distances up to 0.50 where both starts using more space. The number of dimensions and levels limit the possible distances, e.g., EP distances are in increments of 0.25. As both EP dimensions have two levels their distance impact is the same. However, as `Measure` is a stronger correlation indicator, the `Production` weight is increased so only time series with equivalent `Production` members are grouped.

The results for EP and EH are shown in Figure C.20. As expected, only the lowest distance provides a decrease in the storage required, as higher distances create inappropriate groups. This fits with our rule of thumb to use the lowest distance as a start. For EP, times series with similar measures from the same entity are grouped, giving only a 1.14–1.29 times increase in storage compared to our manually tuned results (up to 14.23 times lower than the other formats). For EH, time series with the same measure from each entity in a park are grouped, with the compression

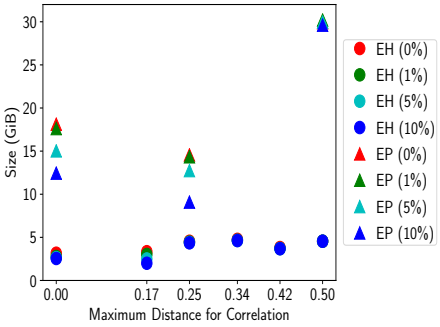

Figure C.20: Distance

outperforming our manual tuning. In summary, even without domain knowledge, MMGC can be used successfully with distance-based grouping and our rule of thumb.

Scale-out We evaluate ModelarDB$_{v2}$'s scalability in two experiments. First, we compare it against the other formats for L-AGG on the local cluster. Second, we evaluate its scalability on L-AGG with MMGC enabled using 1–32 Standard_D8_v3 MS Azure nodes. This node type is selected based on Spark, Cassandra, and Azure documentation [19–21]. The local cluster configuration is used on Azure, with the exception that Spark is allowed 50% of each node's memory as no crashes occur with this configuration. EP is duplicated until the data ingested by each node exceeds its memory, so ModelarDB$_{v2}$+G cannot simply cache all data in memory. The values of each duplicated data set are multiplied with a random value in the range [0.001, 1.001) to avoid repeated values. Queries are executed using the most appropriate method for each system: InfluxDB's command-line interface (CLI), ModelarDB's Segment View (SV) and Data Point View (DPV), and a Spark SQL Data Frame (S) for Cassandra, Parquet, and ORC.

Local results are shown in Figure C.21. ModelarDB$_{v2}$+G outperforms almost all of the other formats with Parquet being only 2.26 times faster despite the benefits of its

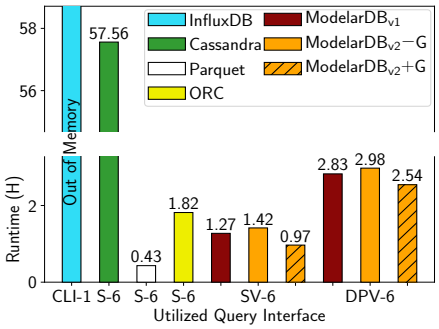

Figure C.21: L-AGG, EP

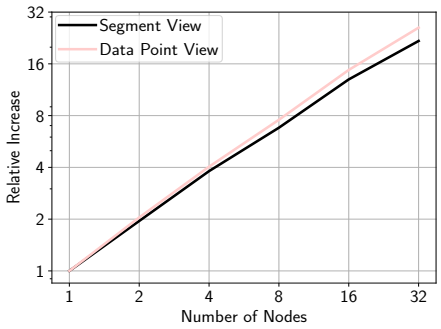

Figure C.22: Scale-out, L-AGG

column-based layout for simple aggregate queries on one column. However, compared to ModelarDB$_{v2}$+G Parquet is 2.94 times slower to ingest data, does not allow for online analytics, and uses up to 11.60 times more storage for EP. L-AGG fails on InfluxDB due to memory limitations on a single node with all 8 GiB available to it and the OS, and the open-source version does not support distribution. ModelarDB$_{v2}$+G executes L-AGG on a single worker in just 6.63 hours using SV. We have previously shown that the slower ModelarDB$_{v1}$ outperforms InfluxDB at scale [7]. Azure results are shown in Figure C.22 with ModelarDB$_{v2}$+G scaling linearly for both SV and DPV. This is expected as ModelarDB$_{v2}$ assigns each time series group to a node, allowing queries to be answered without shuffling. In summary, ModelarDB$_{v2}$ provides either faster (up to 59.34 times) or comparable query performance compared to the other formats for large scale queries and scales linearly when adding additional nodes.

Additional Query Processing Performance To further evaluate ModelarDB$_{v2}$'s query performance we execute S-AGG, P/R and M-AGG on both data sets using the same query interfaces. M-AGG cannot be executed for ModelarDB$_{v1}$ as it does not support dimensions, nor InfluxDB as it can only aggregate fixed size intervals, e.g., an hour or a day [22, 23]. InfluxDB has no `DATEPART` functionality, so aggregates over, e.g., the days of months are not supported [24] unlike ModelarDB$_{v2}$.

S-AGG results are shown in Figures C.23–C.24. ModelarDB$_{v2}$+G is slower as expected as it must read a group to access even a single time series. For EP, ModelarDB$_{v2}$ is much faster than Cassandra and only slightly slower than the other formats. As EP consists of small time series, this shows that ModelarDB$_{v2}$'s MMGC support has a very small overhead. For EH, only Parquet and ORC are faster (7.35 times and 1.55 times, respectively) than ModelarDB$_{v2}$−G due to their column-based layout. However, ModelarDB$_{v2}$−G has up to 2.93 times faster ingestion and uses up to 41.96 times less storage than Parquet and ORC. Last, the benefit of Algorithm C.5 is clear as SV is up to 4.13 times faster compared to DPV. Point and range queries are not the intended use-case for ModelarDB$_{v2}$, as they require reading large segments representing multiple time series from disk, but are evaluated for completeness. The

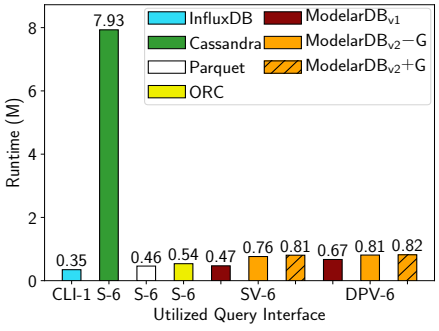

Figure C.23: S-AGG, EP

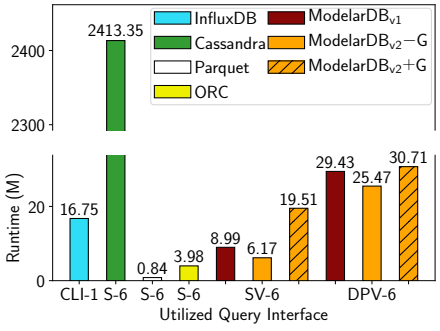

Figure C.24: S-AGG, EH

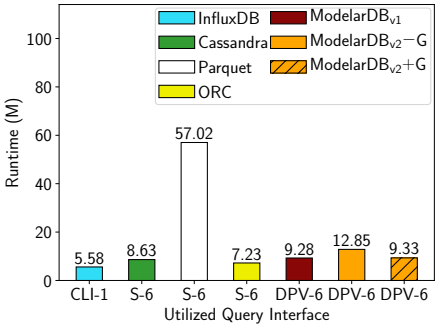

Figure C.25: P/R, EP

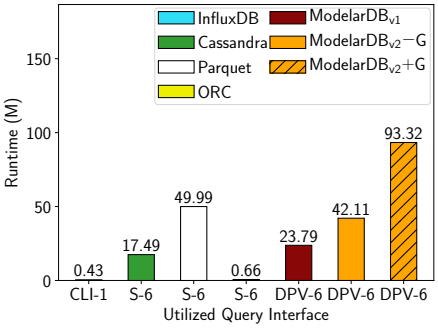

Figure C.26: P/R, EH

results are shown in Figures C.25–C.26. For EP ModelarDB$_{v2}$+G performs better than ModelarDB$_{v2}$−G as it executes range queries extracting a short sub-sequence from all time series faster. However, to support MMGC, predicate push-down for point-range queries is implemented less efficiently in ModelarDB$_{v2}$ than in ModelarDB$_{v1}$. As a result, the query performance of ModelarDB$_{v1}$ and ModelarDB$_{v2}$+G for P/R on EP is similar. For EH, ModelarDB$_{v2}$−G is 1.77 times slower than ModelarDB$_{v1}$, as expected, while ModelarDB$_{v2}$+G is 3.92 times slower as the groups in EH are larger. M-AGG results on EP are shown in Figures C.27–C.28. For M-AGG-One, queries `GROUP BY` *category* which match the groups created by ModelarDB$_{v2}$+G when ingesting. As having a direct match between the groups and the executed aggregate queries is the optimal use-case for MMGC, since each query only reads time series required for the query, ModelarDB$_{v2}$+G is 1.93–54.70 times faster than the other formats. For M-AGG-Two, queries `GROUP BY` *concrete* to drill-down one level below the grouping level. However, contrary to pre-computed aggregates, ModelarDB$_{v2}$+G can execute queries on each time series in a group, so it is still the fastest by 2.36–55.21 times. M-AGG results on EH are shown in Figures C.29–C.30. For M-AGG-One, queries `GROUP BY` *park* and ModelarDB$_{v2}$−G is 3.22–253.78 times faster than the other

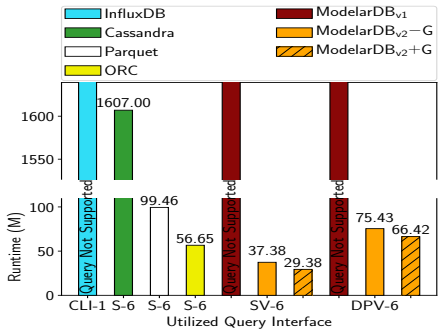

Figure C.27: M-AGG-One, EP

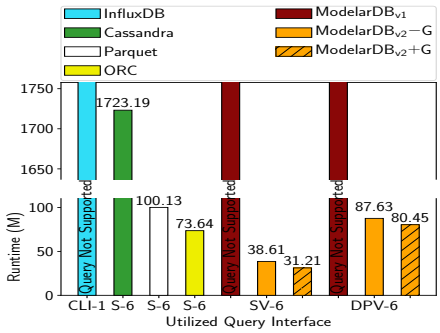

Figure C.28: M-AGG-Two, EP

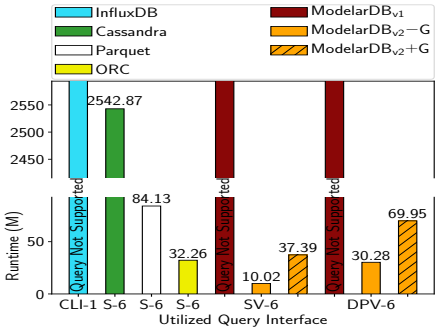

Figure C.29: M-AGG-One, EH

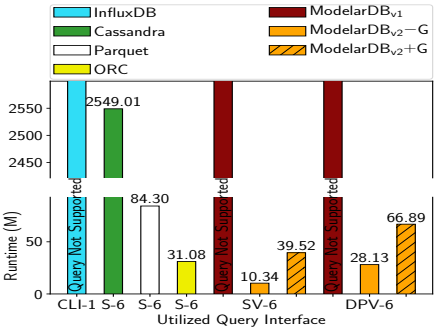

Figure C.30: M-AGG-Two, EH

formats, while for M-AGG-Two, queries `GROUP BY` *entity* and ModelarDB$_{v2}-$G is 3.01–246.52 times faster. ModelarDB$_{v2}+$G is 2.13–2.25 and 64.50–68.01 times faster than Parquet and Cassandra, respectively, with ORC being only 1.16–1.27 times faster. ModelarDB$_{v2}-$G is faster due to the difference in the group size when grouping using the lowest distance (`0.16666667`). Last, the benefit of Algorithm C.6 is clear as SV is up to 3.02 times faster than DPV. In summary, for S-AGG and P/R ModelarDB$_{v2}$ provides performance competitive with the other formats when MMGC is disabled. ModelarDB$_{v2}$ outperforms all other formats by up to 253.78 times for M-AGG due to model-based query processing.

8 Related Work

We summarize papers about model-based time series management and model-based OLAP. There are surveys about model-based time series management [1, 2], use of signal processing in the energy domain [25], Hadoop-based OLAP [26] and TSMSs [27].

Group Compression: Compression of correlated time series is primarily used for efficient data acquisition in sensor networks [28–31]. However, group compression methods designed for TSMSs have been proposed. Gamps [8] approximates each time series using constant functions and then relaxes error bounds before compressing overlapping models together, possibly with scaling. Static grouping is done by an approximate algorithm, with groups re-computed at run-time using dynamically sized windows. MTSC [9] proposes two graph-based methods for partitioning time series into correlated groups MTSC$_{mc}$ and MTSC$_{star}$, with MTSC$_{star}$ sacrificing compression for performance. MTSC also uses constant functions and represents correlated series using a base signal and scaling. Sprintz [31] performs lossless compression of integer time series. It is designed for small devices with no floating-point operations with decompression on large servers. Sprintz combines forecasting, bit packing of errors for time series groups, run-length encoding, and Huffman encoding. As it is designed for integers, floating-point values must be quantized leading to lossy compression without

a guaranteed error bound. Group compression has also been used in other domains. Elgohary et. al. [14, 15] proposed compressing static matrices of features for machine learning using dense dictionary coding, run-length encoding and offset-list encoding. Correlated columns are found by analyzing the static data set and compressed together. For performance, common machine learning operations are performed directly on the compressed matrix and vectors.

Multi-Model Compression: MMC was proposed in [3, 4]. Model types fit models to data points in parallel until they all fail, the model with the best compression is stored. The Adaptive Approximation (AA) algorithm [5] fits models in parallel and creates segments as each model type fails. After all model types fail, segments from the model type with the best compression are stored. In [6] regression models are fitted in sequence with coefficients added as required by the error bound. The model with the best compression is stored after a given number of coefficients.

Model-Based Data Management Systems: DBMSs with explicit support for using mathematical models for data cleaning or compression have also been proposed. MauveDB [32] integrates model use in an RDBMS using views, to support data cleaning without exporting data to an external application. FunctionDB [33] natively supports polynomial function models and evaluate queries directly on models when possible. Plato [34] supports models for cleaning and can add user-defined model types integrating with its optimizer and query processor. Using an in-memory tree index, a distributed key-value store and MapReduce [35] allows segments to be stored and queried in a distributed system. ModelarDB$_{v1}$ [7] is a distributed model-based TSMS using MMC with user-defined model types by integrating its portable core with Spark and Cassandra.

Model-Based OLAP: Another use of model-based time series compression is approximate materialization of data cubes. Perera et. al. [36] propose offline algorithms for finding similarities between time series aggregates. In an OLAP cube, similar aggregates are then materialized as a model (perhaps with an offset) to reduce the materialized cube size. A similar method for online data cubes was proposed by Shaikh et. al. [37]. Using models an approximate data cube is materialized in memory. As data points are ingested, the in-memory data cube is updated and data points persisted to disk. Models representing old data are persisted to preserve memory.

ModelarDB$_{v2}$: In contrast to existing model-based compression algorithms [3–6, 8, 9] and systems [7, 32–35], ModelarDB$_{v2}$ utilizes multiple model types for compression and unifies MMC and MGC to create the novel MMGC method for efficient compression of time series online. In addition, ModelarDB$_{v2}$ groups correlated time series using only metadata and user hints based on domain knowledge. Analyzing time series correlation to gain domain knowledge is an orthogonal problem [8, 9, 14, 15]. As correlation is determined only by metadata, data points can be sent directly to the worker assigned to each time series group. A simple API allows users to add user-defined model types without recompiling ModelarDB$_{v2}$. Compared to other OLAP systems [38–45], ModelarDB$_{v2}$ executes multi-dimensional aggregate queries on models. While existing model-based approaches for OLAP [36, 37] store both data

points and models, ModelarDB$_{v1}$ stores only highly compressed models. In summary, ModelarDB$_{v2}$ provides state-of-the-art compression and query performance for time series by compressing correlated series as a sequence of models and executing OLAP queries on it.

9 Conclusion & Future Work

Motivated by the need to efficiently store and perform multidimensional analysis on big time series from reliable sensors, we presented a new version of our distributed model-based TSMS *ModelarDB*, ModelarDB$_{v2}$. It achieves state-of-the-art compression and query performance by exploiting time series correlation using multiple model types (optionally user-defined). We presented several novel contributions: (i) the concept of *Multi-model Group Compression* and extensions to model types to support it, (ii) a set of primitives simplifying describing time series correlation without requiring historical data, and (iii) query processing algorithms for efficiently evaluating multidimensional aggregate queries directly on models. ModelarDB$_{v2}$ uses stock versions of Apache Spark and Cassandra. A comprehensive evaluation shows that ModelarDB$_{v2}$ provides faster ingestion, a significantly reduced storage need by adaptively selecting appropriate model types for dynamically sized segments, and provides much faster query performance for multidimensional aggregates.

In future work, we will simplify ModelarDB use and increase its query performance by: (i) Developing indexing techniques exploiting that data is stored as user-defined model types. (ii) Supporting high-level analytical queries, e.g., similarity search, directly on models. (iii) Removing or automatically inferring parameter arguments.

10 Acknowledgments

This research was supported by the DiCyPS center funded by Innovation Fund Denmark [46], the GOFLEX project EU grant agreement No 731232 [47], and Microsoft Azure for Research [48].

References

[1] S. Sathe, T. G. Papaioannou, H. Jeung, and K. Aberer, "A Survey of Model-based Sensor Data Acquisition and Management," in *Managing and Mining Sensor Data*. Springer, 2013, pp. 9–50.

[2] N. Q. V. Hung, H. Jeung, and K. Aberer, "An Evaluation of Model-Based Approaches to Sensor Data Compression," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 25, no. 11, pp. 2434–2447, November 2013.

[3] T. G. Papaioannou, M. Riahi, and K. Aberer, "Towards Online Multi-Model Approximation of Time Series," in *Proceedings of the 12th International Conference on Mobile Data Management (MDM)*, vol. 1. IEEE, 2011, pp. 33–38.

[4] ——, "Towards Online Multi-Model Approximation of Time Series," https://infoscience.epfl.ch/record/164651/files/tech_report.pdf, EPFL LSIR, Tech. Rep., 2011.

[5] J. Qi, R. Zhang, K. Ramamohanarao, H. Wang, Z. Wen, and D. Wu, "Indexable online time series segmentation with error bound guarantee," *World Wide Web (WWW)*, vol. 18, no. 2, pp. 359–401, March 2015.

[6] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm, "A time-series compression technique and its application to the smart grid," *The VLDB Journal (VLDBJ)*, vol. 24, no. 2, pp. 193–218, April 2015.

[7] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "ModelarDB: Modular Model-based Time Series Management with Spark and Cassandra," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 11, pp. 1688–1701, July 2018.

[8] S. Gandhi, S. Nath, S. Suri, and J. Liu, "GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2009, pp. 771–784.

[9] N. Pan, P. Wang, J. Wu, and W. Wang, "MTSC: An Effective Multiple Time Series Compressing Approach," in *Proceedings of the 29th International Conference on Database and Expert Systems Applications (DEXA)*. Springer, 2018, pp. 267–282.

[10] H. Elmeleegy, A. K. Elmagarmid, E. Cecchet, W. G. Aref, and W. Zwaenepoel, "Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 145–156, August 2009.

[11] R. E. Korf, "Multi-Way Number Partitioning," in *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2009, pp. 538–543.

[12] I. Lazaridis and S. Mehrotra, "Capturing Sensor-Generated Time Series with Quality Guarantees," in *Proceedings of the 19th International Conference on Data Engineering (ICDE)*. IEEE, 2003, pp. 429–440.

[13] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A Fast, Scalable, In-Memory Time Series Database," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1816–1827, August 2015.

[14] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed Linear Algebra for Large-Scale Machine Learning," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 12, pp. 960–971, August 2016.

[15] ——, "Compressed linear algebra for large-scale machine learning," *The VLDB Journal (VLDBJ)*, vol. 27, no. 5, pp. 719–744, October 2018.

[16] N. T. T. Ho, H. Vo, M. Vu, and T. B. Pedersen, "AMIC: An Adaptive Information Theoretic Method to Identify Multi-Scale Temporal Correlations in Big Time Series Data," *IEEE Transactions on Big Data*, 2019, Early Access.

[17] N. T. T. Ho, T. B. Pedersen, M. Vu, H. L. Van, and C. A. N. Biscio, "Efficient Bottom-Up Discovery of Multi-Scale Time Series Correlations Using Mutual Information," in *Proceedings of the 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1734–1737.

[18] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," in *Proceedings of the 12th International Conference on Data Engineering (ICDE)*. IEEE, 1996, pp. 152–159.

[19] "Azure Databricks," https://azure.microsoft.com/en-us/pricing/details/databricks/, Viewed: 2019-08-14.

[20] "Apache Cassandra - Hardware Choices," http://cassandra.apache.org/doc/latest/operating/hardware.html, Viewed: 2019-08-14.

[21] "Apache Spark - Hardware Provisioning," https://spark.apache.org/docs/2.1.0/hardware-provisioning.html, Viewed: 2019-08-14.

[22] "InfluxQL Reference - Durations," https://docs.influxdata.com/influxdb/v1.4/query_language/spec/#durations, Viewed: 2019-08-14.

[23] "InfluxDB - Issue 3991," https://github.com/influxdata/influxdb/issues/3991, Viewed: 2019-08-14.

[24] "InfluxDB - Issue 6723," https://github.com/influxdata/influxdb/issues/6723, Viewed: 2019-08-14.

[25] Z. Uddin, A. Ahmad, A. Qamar, and M. Altaf, "Recent advances of the signal processing techniques in future smart grids," *Human-centric Computing and Information Sciences (HCIS)*, vol. 8, no. 1, 2018.

[26] M. Ptiček and B. Vrdoljak, "MapReduce Research on Warehousing of Big Data," in *40th International Convention on Information and Communication Technology, Electronics and Microelectronics*, 2017, pp. 1361–1366.

[27] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time Series Management Systems: A Survey," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 29, no. 11, pp. 2581–2600, November 2017.

[28] T. Dang, N. Bulusu, and W.-c. Feng, "RIDA: A Robust Information-Driven Data Compression Architecture for Irregular Wireless Sensor Networks," in *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN)*. Springer, 2007, pp. 133–149.

[29] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos, "Compressing Historical Information in Sensor Networks," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2004, pp. 527–538.

[30] B. Wang, Y. Song, Y. Sun, and J. Liu, "Improvements to Online Distributed Monitoring Systems," in *Trustcom/BigDataSE/ISPA*. IEEE, 2016, pp. 1093–1100.

[31] D. Blalock, S. Madden, and J. Guttag, "Sprintz: Time Series Compression for the Internet of Things," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, vol. 2, no. 3, September 2018.

[32] A. Deshpande and S. Madden, "MauveDB: Supporting Model-based User Views in Database Systems," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2006, pp. 73–84.

[33] A. Thiagarajan and S. Madden, "Querying Continuous Functions in a Database System," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2008, pp. 791–804.

[34] Y. Katsis, Y. Freund, and Y. Papakonstantinou, "Combining Databases and Signal Processing in Plato," in *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR)*, 2015.

[35] T. Guo, T. G. Papaioannou, and K. Aberer, "Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud," *Big Data Research*, vol. 1, pp. 52–65, August 2014.

[36] K. S. Perera, M. Hahmann, W. Lehner, T. B. Pedersen, and C. Thomsen, "Modeling Large Time Series for Efficient Approximate Query Processing," in *Database Systems for Advanced Applications - DASFAA International Workshops, SeCoP, BDMS, and Posters*. Springer, 2015, pp. 190–204.

[37] S. A. Shaikh and H. Kitagawa, "Approximate OLAP on Sustained Data Streams," in *Proceedings of the 22nd International Conference on Database Systems for Advanced Applications (DASFAA)*. Springer, 2017, pp. 102–118.

[38] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive – A Petabyte Scale Data Warehouse Using Hadoop," in *Proceedings of the 26th International Conference on Data Engineering (ICDE)*. IEEE, 2010, pp. 996–1005.

[39] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major Technical Advancements in Apache Hive," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 1235–1246.

[40] S. Chen, "Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 1459–1468, 2010.

[41] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A Real-time Analytical Data Store," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 157–168.

[42] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in Spark," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2015, pp. 1383–1394.

[43] S. Vitthal Ranawade, S. Navale, A. Dhamal, K. Deshpande, and C. Ghuge, "Online Analytical Processing on Hadoop using Apache Kylin," *International Journal of Applied Information Systems (IJAIS)*, vol. 12, pp. 1–5, May 2017.

[44] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon Redshift and the Case for Simpler Data Warehouses," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2015, pp. 1917–1923.

[45] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang *et al.*, "The Snowflake Elastic Data Warehouse," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2016, pp. 215–226.

[46] "DiCyPS - Center for Data-Intensive Cyber-Physical Systems," http://www.dicyps.dk/dicyps-in-english/, Viewed: 2019-08-14.

[47] "GOFLEX," https://goflex-project.eu/, Viewed: 2019-08-14.

[48] "Microsoft Azure for Research," https://www.microsoft.com/en-us/research/academic-program/microsoft-azure-for-research/, Viewed: 2019-08-14.

Paper D

Demonstration of ModelarDB: Model-Based Management of Dimensional Time Series

Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen

The paper has been published in the
Proceedings of the International Conference on Management of Data (SIGMOD),
Pages 1933–1936, 2019. DOI: 10.1145/3299869.3320216

Abstract

Due to the big amounts of sensor data produced, it is infeasible to store all of the data points collected and practitioners currently hide outliers by storing simple aggregates instead. As a remedy, we demonstrate ModelarDB*, a model-based TSMS for* time series with dimensions *and* possibly gaps*. In this demonstration, participants can ingest data sets from multiple domains and experience how ModelarDB provides fast ingestion and a high compression ratio by adaptively compressing time series using a set of* models *to accommodate changes in the structure of each time series over time. Models approximate time series within a user-defined error bound (possibly zero). Participants can also experience how the compression ratio can be improved by ingesting correlated time series in* groups *created by ModelarDB from user-hints. Participants provide these using primitives for describing correlation. Last, participants can execute SQL queries on the ingested data sets and see how the system optimizes queries directly on models.*

1 Introduction

ModelarDB [1, 2] is a model-based distributed TSMS for management of *dimensional time series* possibly with gaps. *Dimensions* are like in a data warehouse with each dimension consisting of hierarchically organized *members* describing the time series. A *gap* is a sub-sequence from a time series where no values exist, and a *model* is any representation of a time series from which the original values can be reconstructed within a *user-defined error bound* (possibly zero).

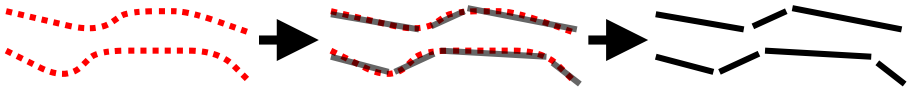

Figure D.1: Model-based compression of time series

Figure D.1 shows an example where linear functions $y = ax + b$ are used to represent increasing and decreasing sub-sequences. The values of these time series can be represented using just seven linear models, with the models clearly preserving the structure of the time series using only two parameters each. If we assume each time series represent five minutes and a data point is sampled each second, $(300 \times 4)/1024 = 1.17$ KiB are required to store the values using `floats`, while only $7 \times (4+4) = 56$ bytes are required for the models. As time series often change over time the compression ratio can be increased significantly by compressing each time series using multiple models [1, 2]. In addition, as time series often are correlated, e.g., from temperature sensors in close proximity, additional compression can be achieved by compressing correlated time series together [2]. ModelarDB combines both of these approaches using a novel technique we named MMGC [2]. The system includes three types of models: a constant function, a linear function, and lossless compression of floating-point values [2]. In addition, because each model is a black-box, user-defined models can optionally be added without recompiling the TSMS. ModelarDB accepts queries as SQL and executes simple and multi-dimensional aggregate queries directly on models instead of reconstructed data points for increased query performance. As an example, for a linear function, `SUM` can be computed in constant time using the model compared to linear time using data points. In summary: ModelarDB provides state-of-art model-based compression with very fast ingestion, efficient query processing using a familiar interface, and is extensible for the end user without recompiling the TSMS. This make ModelarDB unique among TSMSs as the few existing model-based systems focus on data cleaning, compress time series using only one type of model, or do not exploit correlation [3]. The rest of the paper is structured as follows. In Section 2 we provide an overview of ModelarDB, while Section 3 describes the demonstration scenarios.

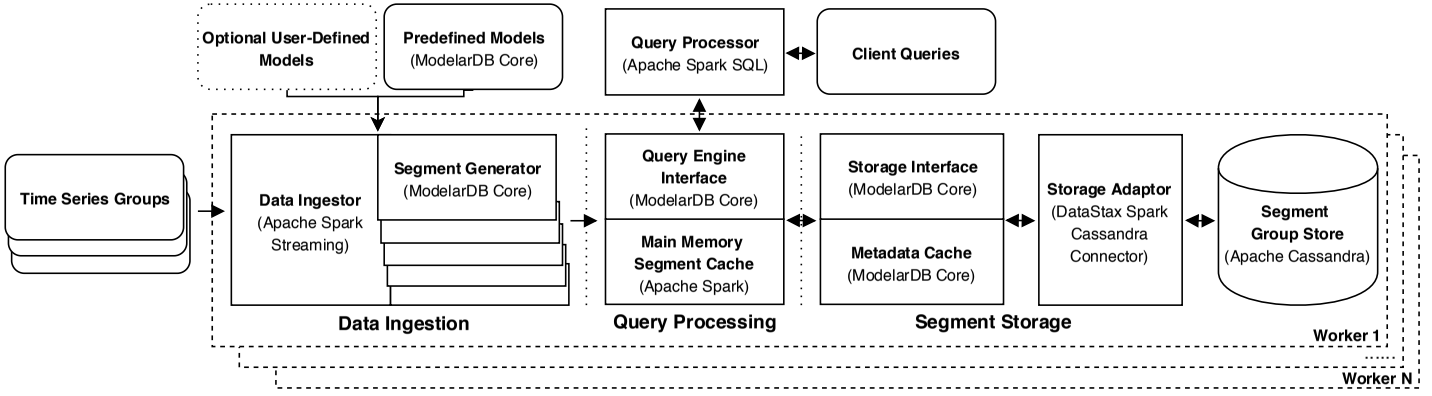

Figure D.2: The distributed architecture of ModelarDB with data ingestion, query processing, and segment storage

2 ModelarDB

Architecture: ModelarDB is implemented (`https://github.com/skejserjensen/ModelarDB`) as the portable Java library ModelarDB Core which can be interfaced with query processing systems and storage systems. This modular architecture allows ModelarDB to be optimized for multiple domains and reuse infrastructure by interfacing ModelarDB Core with other systems. ModelarDB Core also contains the models provided by the system. Users can optionally also implement additional models without recompiling the system. Our current implementation interfaces ModelarDB Core with the stock versions of Apache Spark for query processing, and Apache Cassandra or a JDBC compatible RDBMS for storage. As a result ModelarDB can be deployed on an existing cluster running Spark simply by submitting a JAR file using Spark's `spark-submit` script. The distributed architecture of ModelarDB is shown in Figure D.2. The architecture consists of three sets of components: data ingestion, query processing, and segment storage. Each component is annotated with the software providing that functionality (our components are marked ModelarDB Core). We now describe the user-facing components, data ingestion and query processing (they and segment storage are documented in detail in [1, 2]).

Model-based Data Ingestion: To ensure low latency, models are fitted online and the ingestion process requires only that users specify time series and an error bound. ModelarDB statically assigns each time series to the worker node with the most available resources. Then as shown in Figure D.1 the *Segment Generator* fits models to dynamically sized sub-sequences of each time series. For each, the model with the best compression ratio is emitted to the in-memory cache and persistent storage, see Figure D.2, as part of a segment containing the metadata to reconstruct the time series. When a gap occurs, a segment is emitted and values received after the gap are fitted to a new model. This simplifies implementation of user-defined models as ModelarDB manages gaps [1, 2].

Figure D.3: Model-based compression of grouped series

As many time series are correlated, e.g., wind speed and the energy production of a wind turbine, the compression ratio can be increased further by compressing time series together in a group as shown in Figure D.3. In this example the two time series have a similar structure but different values. Therefore, a scaling constant is applied during ingestion and query processing so the time series have similar structure and values during ingestion with the original values restored during query processing. Compared to compressing the values of each time series separately as in Figure D.1 compressing them together uses only $4 \times (4 + 4) = 32$ bytes instead of 56 bytes. Should a group no longer be correlated, ModelarDB splits the group into sub-groups of correlated time

series, and if they become correlated again, the sub-groups are merged.

```
1 modelardb.correlation 44L80R9a_AirTemperature
2                       44L80R9b_AirTemperature
3 modelardb.correlation Measure 1 Temperature
4 modelardb.correlation Location 0, Measure 3
5 modelardb.correlation 0.25
6 modelardb.correlation Location 0, Measure 3 *
7                       Measure 1 Temperature 0.5
```

Listing D.1: User-hints specifying correlated time series

As ModelarDB is a distributed system, time series compressed together should be ingested by the same node to prevent network traffic from limiting the scalability of the system. As historical data might not be available or be too large (just $50,000$ time series create $\binom{50,000}{2} \approx 1.25 \times 10^9$ pairs) computing correlation from historical data quickly becomes infeasible. To remedy this, ModelarDB groups dimensional time series using their dimensions and user-hints. Users can specify correlation in ModelarDB's configuration file as specific time series, groups of time series with specific members, or time series with similar members in their dimensional hierarchy. Examples of the supported primitives for specifying correlation are shown in Listing D.1. Line 1–2 specify that two time series are correlated, and Line 3 that time series with the member *Temperature* at the first level of the dimension *Measure* are correlated. Line 4 specifies that time series are correlated if they share members in all levels of the *Location* dimension as well as all members until and including the third level of the *Measure* dimension. Line 5 specifies that time series are correlated if the distance between their dimensional hierarchies is 0.25 or below. The distance between hierarchies is in the range $[0, 1]$ and computed based on the number of shared ancestors and the size of each dimension. An example of setting a scaling constant is shown on Line 6–7, where the suffix * *Dimension Level Member ScalingFactor* scales time series with the member `Temperature` at the first level in the `Measure` dimension by `0.5` [2].

The performance of ModelarDB's model-based ingestion is shown in Figure D.4. ModelarDB provides up to 11 time faster ingestion when Bulk Loading on one node (BL-1) compared to other formats and scales when five more nodes are added despite performing Online Analytics (OA-6) in parallel [2].

Model-based Query Processing: ModelarDB uses SQL as its query language to give users a well-known interface and simplify interfacing it with other applications. Queries are executed either directly on segments using the *Segment View* or on reconstructed data points using the *Data Point View* [1, 2]. Dimensions are denormalized and joined to the rows of each view. We have implemented the Segment View by means of Spark's UDAFs with support for simple aggregates like `AVG` and multi-dimensional aggregates using a combination of `GROUP BY` on the denormalized dimensions and UDAFs suffixed by the time interval they aggregate by, e.g., `CUBE_COUNT_HOUR` and `CUBE_AVG_YEAR`. ModelarDB requires no explicit time dimension as aggregate queries in the time dimension can be computed efficiently using the start and end time of segments. To allow queries on the Segment View to be executed at data point granularity, functions are defined to restrict either the start time (`START`), end time

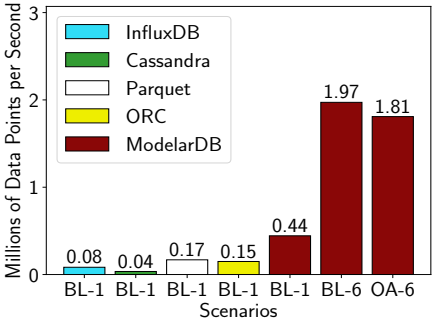

Figure D.4: Ingestion [2]

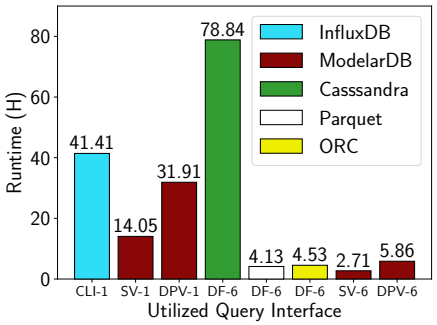

Figure D.5: Aggregation [1]

(`END`), or both (`INTERVAL`) of segments. Efficient execution of queries directly on user-defined models is supported if users optionally implement optimized methods for computing `MIN`, `MAX` and `SUM`.

Examples of queries supported by ModelarDB can be seen in Listing D.2. In Line 1–2 the average of the values from the time series identified by `Tid = 3` is computed using both views. The operator `#` is a specialized version of $\star$ that selects only the columns from the Segment View required for a UDAF. In Line 4–5 the query computes the minimum value ingested after the timestamp *2019–02–06 18:22* using the Segment View. As the `WHERE` clause filters at the segment level the function `START` is used to disregard data older than the timestamp provided. The UDAF `MIN_SS` is a version of `MIN_S` that operates on the struct returned by `START` as Spark does not support functions returning multiple columns. Last in Line 7–8 the average temperature per month is computed for each time series with the member *Temperature* at the *Category* level of the *Measure* dimension.

```
1 SELECT AVG_S(#) FROM Segment WHERE Tid = 3
2 SELECT AVG(Value) FROM DataPoint WHERE Tid = 3
3
4 SELECT MIN_SS( START(#, '2019-02-06 18:22') )
5 FROM Segment WHERE EndTime >= '2019-02-06 18:22'
6
7 SELECT CUBE_AVG_MONTH(#) FROM Segment
8 WHERE Category = 'Temperature' GROUP BY Tid
```

Listing D.2: ModelarDB query examples

The query performance of ModelarDB for large scale aggregate queries can be seen in Figure D.5, where ModelarDB provides 1.52–29.09 times faster query performance using the Segment View on one or six nodes compared to existing formats when executing queries through their Command-Line Interface (CLI) or a Spark DataFrame (DF) [1].

3 Demonstration Scenario

Graphical User Interface: In the demonstration participants can operate ModelarDB through the graphical user interface shown in Figure D.6. The interface is split into data ingestion and query processing. Ingestion allows the user to choose between multiple data sets, specify correlation between the time series, and change the error bound and models used. Correlation can be specified with the primitives using either text or a graphical tool. For each ingested data set the participants can view the size of the data set in comparison to the raw data set, the distribution of models used per time series and the distribution of models used in total. This allows participants to get insights into how ModelarDB adapts to different combinations of data sets and parameters. The query processing component as shown in Figure D.6 allows users to select an ingested data set, either local or remote, and view the storage and models used by ModelarDB. For remote pre-ingested data sets we also provide a comparison to existing formats as shown in the left part of the interface. Using the editor in the right part of the interface, participants can execute any query against the Segment View and the Data Point View. The query results are printed as well as the columns required, how the `WHERE` clause was rewritten to translate time series to time series groups, how predicate push-down was performed, and the query time to illustrate the capabilities of ModelarDB.

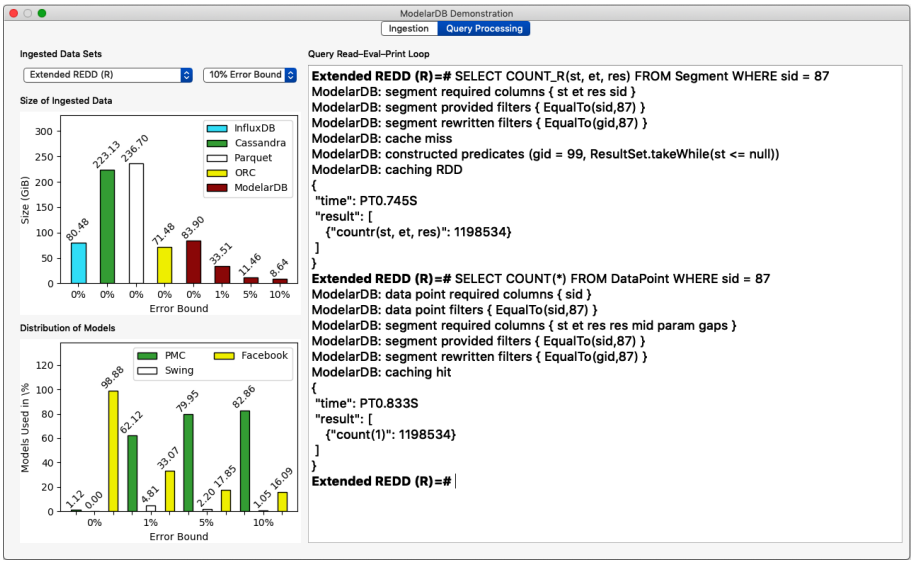

Figure D.6: Executing queries on the Segment View and Data Point View using the interface with a data set from [1]

Data Sets: ModelarDB is designed for clusters, but as the demonstration mainly uses a laptop we use smaller real life data sets from multiple domains. For this small scale demonstration we use the data set REDD which contains measurements of

electricity usage from houses, a data set from BackBlaze containing measurements of hard drive health with metadata regarding the hard disk type, time series from the KNMI Climate Explorer with temperature, wind speed and snow fall with metadata about the location, and last time series from the UCR Time Series Classification Archive. Data sets too small for the demonstration are enlarged by duplicating the time series and applying a scaling constant so duplicate values do not change the results. In addition, we have pre-ingested larger real life data sets about energy production on a remote cluster so participants can experience executing queries at scale. These data sets contain metadata describing both the location and type of measurement performed, but both the dimensions and values have been anonymized due to NDAs with the data providers.

Scenarios: The demonstration allows participants to experience the entire workflow of using ModelarDB. First the participants use the user interface to select a data set and specify correlations before starting the quick ingestion procedure. The participants can then compare the size of the ingested data and the models used with the results from other participants. After the data has been ingested the participants can execute queries on the ingested data sets using the Segment View and Data Point View to see the difference in performance. In addition to the small data sets, participants can execute queries on a cluster to experience operating ModelarDB at scale.

4 Acknowledgments

This research was supported by the DiCyPS (Innovation Fund DK) and GOFLEX (EU grant 731232) projects.

References

[1] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "ModelarDB: Modular Model-based Time Series Management with Spark and Cassandra," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 11, pp. 1688–1701, July 2018.

[2] ——, "Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB," *CoRR*, March 2019, arXiv:1903.10269v1.

[3] ——, "Time Series Management Systems: A Survey," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 29, no. 11, pp. 2581–2600, November 2017.

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-473-7

AALBORG UNIVERSITY PRESS