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

Søren Kejser Jensen
Aalborg University, Denmark
skj@cs.aau.dk

Torben Bach Pedersen
Aalborg University, Denmark
tbp@cs.aau.dk

Christian Thomsen
Aalborg University, Denmark
chr@cs.aau.dk

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 Time Series Management System (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.

PVLDB Reference Format:
Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen. ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra. *PVLDB*, 11(11): 1688-1701, 2018.
DOI: https://doi.org/10.14778/3236187.3236215

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 [29].

Table 1: Comparison of common storage solutions

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

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 select the systems based on DB-Engines Ranking [11], discussions with companies in the energy sector, and our survey [28]. Two Relational Database Management Systems (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 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

Proceedings of the VLDB Endowment, Vol. 11, No. 11
Copyright 2018 VLDB Endowment 2150-8097/18/07... $ 10.00.
DOI: https://doi.org/10.14778/3236187.3236215

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) *Approximate Query Processing (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 [22,37,40], we found no TSMS using multi-model compression for our survey [28]. Also, the existing methods do not provide all the properties listed above. They either provide no latency guarantees [22,40], require a trade-off between latency and compression [37], or limit the supported model types [22, 40]. 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 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), $1 \le i$, 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 $t_i \le 300$, $1 \le i$.

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 $1 \le i$ 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 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 $1 \le i$.*

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

DEFINITION 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, $1 \le i$, 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 $m_{est} = -0.0024t_i + 29.5$, $1 \le i \le 5$, and an error function using the uniform error norm so $m_{err} = max(|v_i - m_{est}(t_i)|)$, $1 \le i \le 5$. 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 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 $t_e = t_s + m \times SI$, $m \in \mathbb{N}_{\ge 2}$, and where no data points exist between t_s and t_e.*

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

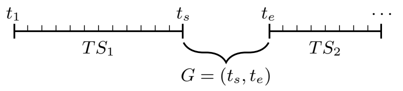


Figure 1: Illustration of a gap G between t_s and t_e

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 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 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.*

In the example for Definition 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 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 3. Data flow between components is shown as arrows, while the sets of components are separated by dashed lines. Our implementation of

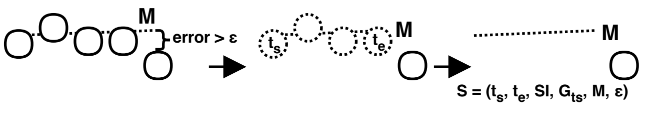


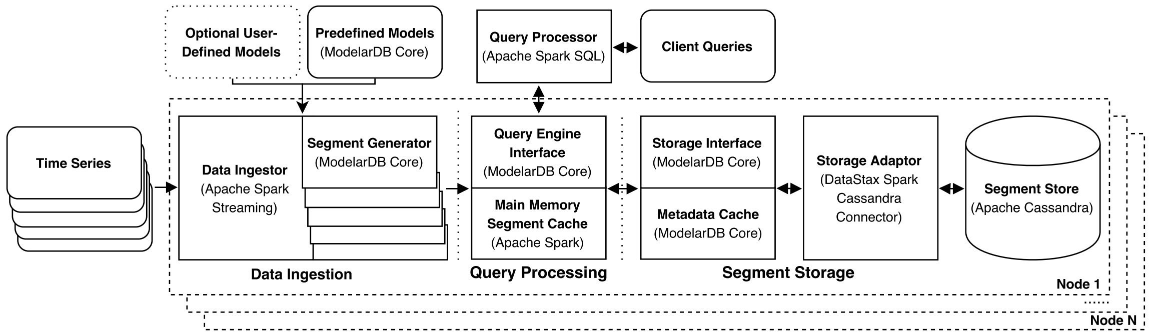
Figure 2: Model-based representation of data points

the architecture integrates ModelarDB Core with the stock versions of Apache Spark [8, 17, 45, 46, 47] for distributed query processing, while Apache Cassandra [2, 32] 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 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 [3, 18], can be added to ModelarDB by implementing a new engine class. Support for other storage systems, e.g. Apache HBase [4] or MongoDB [5], 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 [44]. 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 [39].

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



Optional User-Defined Models
Predefined Models (ModelarDB Core)
Query Processor (Apache Spark SQL)
Client Queries
Time Series
Data Ingestor (Apache Spark Streaming)
Segment Generator (ModelarDB Core)
Query Engine Interface (ModelarDB Core)
Main Memory Segment Cache (Apache Spark)
Storage Interface (ModelarDB Core)
Metadata Cache (ModelarDB Core)
Storage Adaptor (DataStax Spark Cassandra Connector)
Segment Store (Apache Cassandra)
Data Ingestion
Query Processing
Segment Storage
Node 1
Node N

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

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 [31], 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 4 and uses a list of models from which one model is active at a time as proposed by [22]. 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 1. First variables are initialized in Line 8-11, this corresponds to t_0 in Figure 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

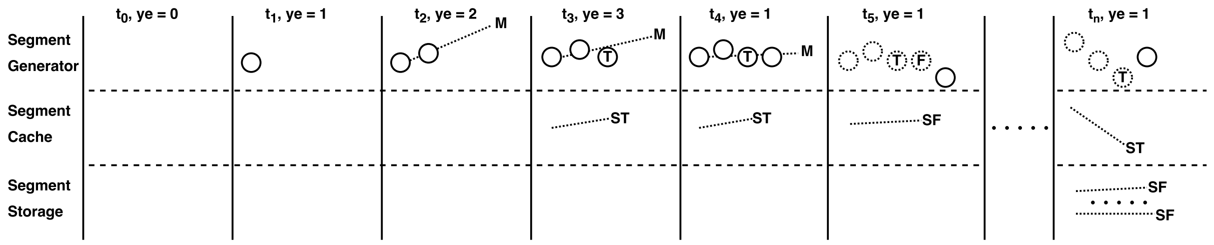


Figure 4: The multi-model compression algorithm, maximum for yet to be emitted (ye) data points is three

Algorithm 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: $model \leftarrow head(models)$
 9: $buffer \leftarrow create\_list()$
10: $yet\_emitted \leftarrow 0$
11: $previous \leftarrow nil$
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:     $previous \leftarrow data\_point$
19:     if $append\_data\_point\_to\_model($
        $data\_point, model, error, limit)$ then
20:         $yet\_emitted \leftarrow yet\_emitted + 1$
21:         if $yet\_emitted = latency$ then
22:             $emit\_temporary\_segment(model, buffer)$
23:             $yet\_emitted \leftarrow 0$
24:         end if
25:     else if $has\_next(models)$ then
26:         $model \leftarrow next(models)$
27:         $initialize(model, buffer)$
28:     else
29:         $emit\_finalized\_segment(models, buffer)$
30:         $model \leftarrow head(models)$
31:         $initialize(model, buffer)$
32:         $yet\_emitted \leftarrow min(yet\_emitted, lengh(buffer))$
33:     end if
34: end while
35: $flush\_buffer(buffer)$
```

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 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 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 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 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 [37, 38] 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 [37, 38]. 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 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 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 2. `Tid` is a unique id assigned to each time series. By having a segment class, a model object can store 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 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

Table 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 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 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 [39], 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 [22], 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 [22] but simply a parameter.

For evaluation we implement a set of general-purpose models from the literature. We base our selection of models on [27] 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 [22, 38]. 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 [34]. 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 [33], the linear Swing model [23], 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 [39] 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 [21, 29, 43].

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 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.

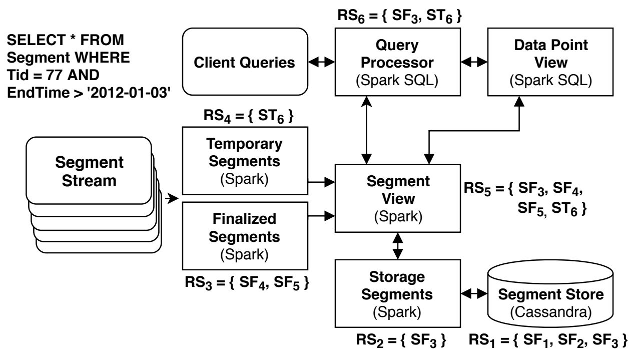
5.3 Execution of Queries on Views

Examples of queries on the views are shown in Listing 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 2: Query examples supported in ModelarDB

To lower query latency, the main memory segment cache, see Figure 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 when a SF with the same `Tid` is emitted so the time intervals do not overlap.



SELECT * FROM Segment WHERE Tid = 77 AND EndTime > '2012-01-03'
Client Queries
RS_6 = { SF_3, ST_6 }
Query Processor (Spark SQL)
Data Point View (Spark SQL)
RS_4 = { ST_6 }
Segment Stream
Temporary Segments (Spark)
Finalized Segments (Spark)
Segment View (Spark)
RS_5 = { SF_3, SF_4, SF_5, ST_6 }
RS_3 = { SF_4, SF_5 }
Storage Segments (Spark)
Segment Store (Cassandra)
RS_2 = { SF_3 }
RS_1 = { SF_1, SF_2, SF_3 }

Figure 5: Query processing in ModelarDB

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 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.

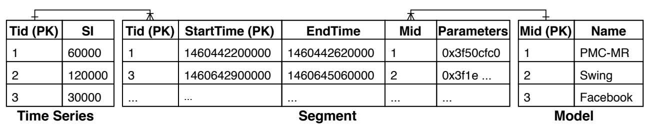
```
1  def getDataPointGridFunction
2    (columns: Array[String]): (DataPoint => Row) = {
3    val target = getTarget(columns, dataPointView)
4    (target: @switch) match {
5      //Permutations of ('tid')
6      case 1 => (dp: DataPoint) => Row(dp.tid)
7      ...
8      //Permutations of ('tid', 'ts',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

Listing 3: Selection of method for a projection

The method generated for the data point view is shown in Listing 3. On Line 3, the list of requested columns is converted to column indexes and concatenated in the 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` [15].

6. SEGMENT STORAGE

Figure 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 [22,24,38], 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.



Tid (PK)	SI
1	60000
2	120000
3	30000

Time Series

Tid (PK)	StartTime (PK)	EndTime	Mid	Parameters
1	1460442200000	1460442620000	1	0x3f50cfc0
3	1460642900000	1460645060000	2	0x3f1e ...
...	...	...	...	...

Segment

Mid (PK)	Name
1	PMC-MR
2	Swing
3	Facebook

Model

Figure 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 [24], secondary indexes can be implemented in ModelarDB as part of the storage interface shown in Figure 3.

6.2 Predicate Push-Down

The columns for which predicate push-down is supported in our implementation are shown in Figure 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 [6,26] files stored in HDFS [42], Apache Parquet [7] files stored in HDFS, InfluxDB [13], and Apache Cassandra [32]. 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 [22,37, 38]. 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 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.

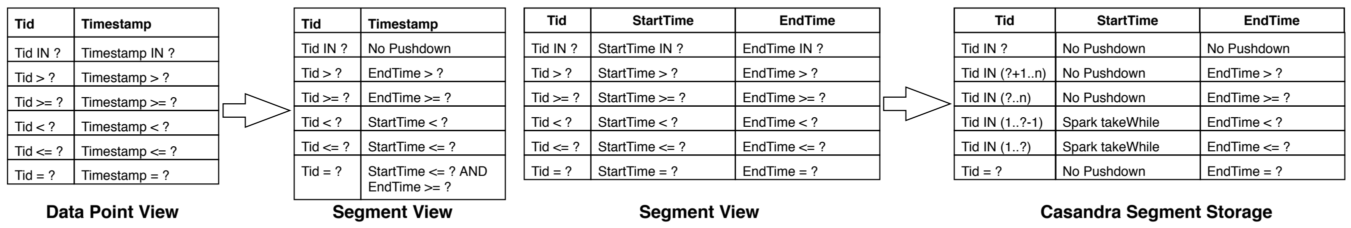
The configuration of each system is, in general, left with its default values. However, the memory available for Spark and either Cassandra or HDFS, is statically allocated to prevent crashes. To divide memory between query processing and data storage, we limit the amount of memory Spark can allocate 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 3, with ModelarDB

Table 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 [33]	Constant Function	Lossy Compression
Swing [23]	Linear Function	Lossy Compression
Facebook [39]	Array of Delta Values	Lossless Compression
Uncompressed Values	Array of Values	No Compression



Data Point View

Tid	Timestamp
Tid IN ?	Timestamp IN ?
Tid > ?	Timestamp > ?
Tid >= ?	Timestamp >= ?
Tid < ?	Timestamp < ?
Tid <= ?	Timestamp <= ?
Tid = ?	Timestamp = ?

Segment View

Tid	Timestamp
Tid IN ?	No Pushdown
Tid > ?	EndTime > ?
Tid >= ?	EndTime >= ?
Tid < ?	StartTime < ?
Tid <= ?	StartTime <= ?
Tid = ?	StartTime <= ? AND EndTime >= ?

Segment View

Tid	StartTime	EndTime
Tid IN ?	StartTime IN ?	EndTime IN ?
Tid > ?	StartTime > ?	EndTime > ?
Tid >= ?	StartTime >= ?	EndTime >= ?
Tid < ?	StartTime < ?	EndTime < ?
Tid <= ?	StartTime <= ?	EndTime <= ?
Tid = ?	StartTime = ?	EndTime = ?

Casandra Segment Storage

Tid	StartTime	EndTime
Tid IN ?	No Pushdown	No Pushdown
Tid IN (?+1..n)	No Pushdown	EndTime > ?
Tid IN (?..n)	No Pushdown	EndTime >= ?
Tid IN (1..?-1)	Spark takeWhile	EndTime < ?
Tid IN (1..?)	Spark takeWhile	EndTime <= ?
Tid = ?	No Pushdown	EndTime = ?

Figure 7: The two-step methods for predicate push-down utilized by 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) [30] 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 [14]. 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.

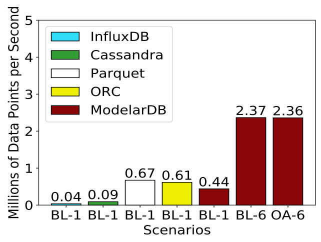


Figure 8: Ingestion, ER

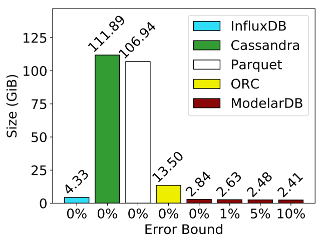


Figure 9: Storage, EH

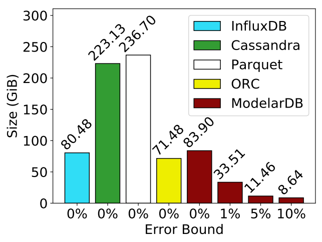


Figure 10: Storage, ER

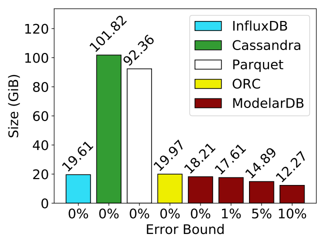


Figure 11: Storage, EP

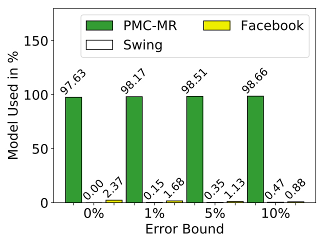


Figure 12: Models, EH

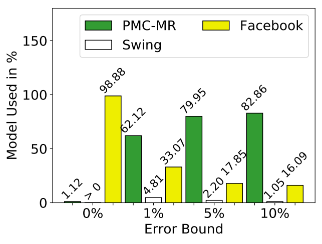


Figure 13: Models, ER

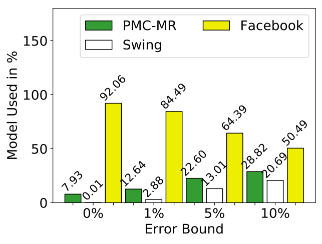


Figure 14: Models, EP

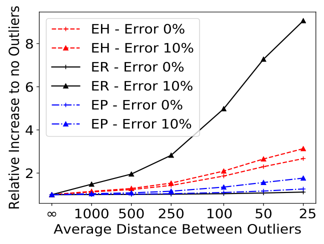


Figure 15: Outlier Effect

The results are shown in Figure 8. As expected InfluxDB and Cassandra had the 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 $(Value\ of\ Data\ Point\ to\ Replace + 1) * 2$.

The storage used for EH is seen in Figure 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 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, 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 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 12—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 15. As expected the storage used 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

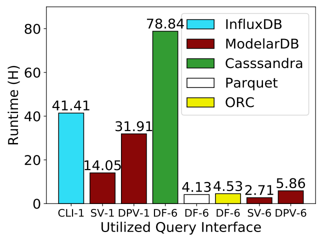


Figure 16: L-AGG, ER

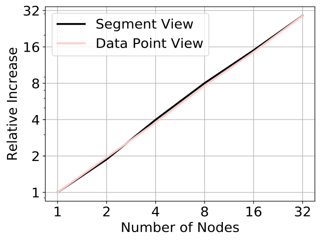


Figure 17: Scale-out

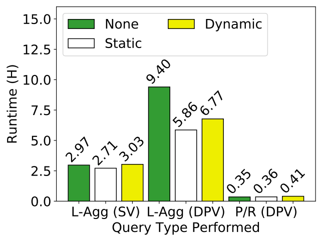


Figure 18: Projection, ER

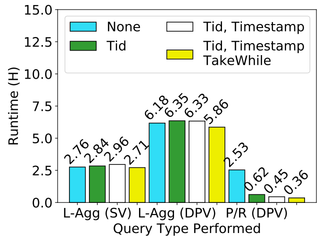


Figure 19: Predicate, ER

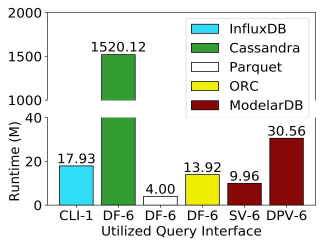


Figure 20: S-AGG, EH

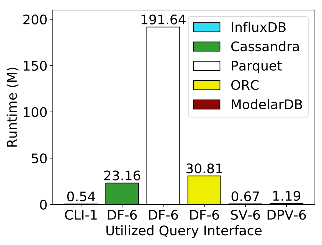


Figure 21: S-AGG, ER

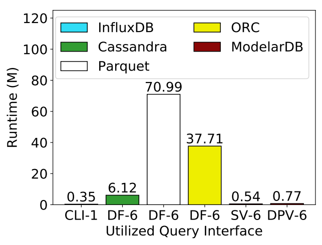


Figure 22: S-AGG, EP

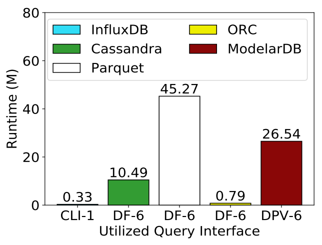


Figure 23: P/R, EH

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 [1,9,10]. 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 [19]. 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 (DF). We evaluate the query performance using a DF and a cached Data Frame (DFC) as shown in Figure 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 16—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 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 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 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 20—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 remains competitive with an optimized single node system.

The results for P/R are shown in Figures 23—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

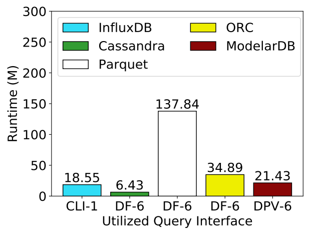


Figure 24: P/R, ER

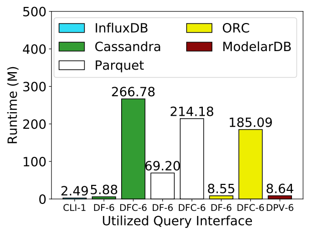


Figure 25: P/R, EP

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 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 [28].

Methods have been proposed for online construction of approximations with minimal error [25], or maximal segment length for better compression [34]. As the optimal model can change over time, methods using multiple mathematical models have been developed. In [37] 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 [38]. The Adaptive Approximation (AA) algorithm [40] 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 [22]. 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 [21] 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 [43] 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 [29] 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 [36], based on the MiSTRAL architecture [35], 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 [20] was proposed by [24]. 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 [21, 29, 43], support for both fixed and dynamically sized user-defined models that can be fitted online without requiring offline training of any kind [36], and automated selection of the most appropriate model for each part of a time series while also storing each segment only once [24].

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 [12], and Microsoft Azure for Research [16].

11. REFERENCES

[1] Apache Cassanddra - Hardware Choices. http://cassandra.apache.org/doc/latest/operating/hardware.html. Viewed: 2018-07-15.

[2] Apache Cassandra. http://cassandra.apache.org/. Viewed: 2018-07-15.

[3] Apache Flink. https://flink.apache.org/. Viewed: 2018-07-15.

[4] Apache HBase. https://hbase.apache.org/. Viewed: 2018-07-15.

[5] Apache MongoDB. https://www.mongodb.com/. Viewed: 2018-07-15.

[6] Apache ORC. https://orc.apache.org/. Viewed: 2018-07-15.

[7] Apache Parquet. https://parquet.apache.org/. Viewed: 2018-07-15.

[8] Apache Spark. https://spark.apache.org/. Viewed: 2018-07-15.

[9] Apache Spark - Hardware Provisioning. https://spark.apache.org/docs/2.1.0/hardware-provisioning.html. Viewed: 2018-07-15.

[10] Azure Databricks. https://azure.microsoft.com/en-us/pricing/details/databricks/. Viewed: 2018-07-15.

[11] DB-Engines Ranking. https://db-engines.com/en/ranking. Viewed: 2018-07-15.

[12] DiCyPS - Center for Data-Intensive Cyber-Physical Systems. http://www.dicyps.dk/dicyps-in-english/. Viewed: 2018-07-15.

[13] InfluxData InfluxDB. https://www.influxdata.com/. Viewed: 2018-07-15.

[14] InfluxDB API Client Libraries. https://docs.influxdata.com/influxdb/v1.4/tools/api_client_libraries/. Viewed: 2018-07-15.

[15] Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-3.html#jvms-3.10. Viewed: 2018-07-15.

[16] Microsoft Azure for Research. https://www.microsoft.com/en-us/research/academic-program/microsoft-azure-for-research/. Viewed: 2018-07-15.

[17] 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 SIGMOD*, pages 1383–1394. ACM, 2015.

[18] 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*, 38(4):28–38, 2015.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SOCC*, pages 143–154. ACM, 2010.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*, pages 137–150. USENIX, 2004.

[21] A. Deshpande and S. Madden. MauveDB: Supporting Model-based User Views in Database Systems. In *Proceedings of SIGMOD*, pages 73–84. ACM, 2006.

[22] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm. A time-series compression technique and its application to the smart grid. *The VLDB Journal*, 24(2):193–218, 2015.

[23] H. Elmeleegy, A. K. Elmagarmid, E. Cecchet, W. G. Aref, and W. Zwaenepoel. Online piece-wise linear approximation of numerical streams with precision guarantees. *PVLDB*, 2(1):145–156, 2009.

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

[25] T. Guo, Z. Yan, and K. Aberer. An Adaptive Approach for Online Segmentation of Multi-Dimensional Mobile Data. In *Proceedings of MobiDE*, pages 7–14. ACM, 2012.

[26] 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 SIGMOD*, pages 1235–1246. ACM, 2014.

[27] N. Q. V. Hung, H. Jeung, and K. Aberer. An Evaluation of Model-Based Approaches to Sensor Data Compression. *IEEE TKDE*, 25(11):2434–2447, 2013.

[28] S. K. Jensen, T. B. Pedersen, and C. Thomsen. Time Series Management Systems: A Survey. *IEEE TKDE*, 29(11):2581–2600, 2017.

[29] Y. Katsis, Y. Freund, and Y. Papakonstantinou. Combining Databases and Signal Processing in Plato. In *Proceedings of CIDR*, 2015.

[30] J. Z. Kolter and M. J. Johnson. REDD: A Public Data Set for Energy Disaggregation Research. In *Proceedings of SustKDD*, 2011.

[31] R. E. Korf. Multi-Way Number Partitioning. In *Proceedings of IJCAI*, pages 538–543, 2009.

[32] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[33] I. Lazaridis and S. Mehrotra. Capturing Sensor-Generated Time Series with Quality Guarantees. In *Proceedings of ICDE*, pages 429–440. IEEE, 2003.

[34] 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 ICDE*, pages 173–184. IEEE, 2015.

[35] 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 Big Data*, pages 15–21. IEEE, 2013.

[36] 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 Big Data*, pages 291–300. IEEE, 2014.

[37] T. G. Papaioannou, M. Riahi, and K. Aberer. Towards Online Multi-Model Approximation of Time Series. In *Proceedings of MDM*, volume 1, pages 33–38. IEEE, 2011.

[38] T. G. Papaioannou, M. Riahi, and K. Aberer. Towards Online Multi-Model Approximation of Time Series. Technical report, EPFL LSIR, 2011.

[39] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A Fast, Scalable,

In-Memory Time Series Database. *PVLDB*, 8(12):1816–1827, 2015.

[40] 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*, 18(2):359–401, 2015.

[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*, pages 9–50. Springer, 2013.

[42] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of MSST*, pages 1–10. IEEE, 2010.

[43] A. Thiagarajan and S. Madden. Querying Continuous Functions in a Database System. In *Proceedings of SIGMOD*, pages 791–804. ACM, 2008.

[44] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient In-Memory Spatial Analytics. In *Proceedings of SIGMOD*, pages 1071–1085. ACM, 2016.

[45] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of HotCloud*. USENIX, 2010.

[46] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of SIGOPS*, pages 423–438. ACM, 2013.

[47] 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 HotCloud*. USENIX, 2012.