Time Series Management Systems: A 2022 Survey

Jensen, Søren Kejser; Pedersen, Torben Bach; Thomsen, Christian

Published in:
Data Series Management and Analytics

Creative Commons License
Unspecified

Publication date:
2022

Document Version
Accepted author manuscript, peer reviewed version

Citation for published version (APA):
Jensen, S. K., Pedersen, T. B., & Thomsen, C. (in press). Time Series Management Systems: A 2022 Survey. In T. Palpanas, & K. Zoumpatianos (Eds.), *Data Series Management and Analytics* Association for Computing Machinery (ACM).

1 Time Series Management Systems: A 2022 Survey

Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen
Department of Computer Science, Aalborg University, Denmark
skj@cs.aau.dk, tbp@cs.aau.dk, chr@cs.aau.dk

1.1 Abstract

Enormous amounts of time series are being collected in many different domains. These include, but are not limited to, aviation, computing, energy, finance, logistics, and medicine. However, general-purpose Database Management Systems (DBMSs) are not optimized for times series management and thus significantly limit the amount of time series that can be efficiently stored and analyzed. As a remedy, specialized *Time Series Management Systems (TSMSs)* have been developed. This chapter[1], provides a thorough survey and classification of TSMSs that are developed through academic or industrial research and documented through peer-reviewed papers. To document their design and novel contributions, a summary of each system is provided. The systems are primarily classified based on their architecture. In addition, the systems are classified based on: when and why each system was developed, how it can be deployed, how mature its implementation is, how scalable it is, how it processes time series, what interfaces it provides, the type of approximation it supports, how low latency it can achieve, how it stores time series, and the types of queries it supports. The chapter concludes with a collection of open research problems based on the limitations of the surveyed systems.

[1] This chapter is a heavily revised and updated version of [Jensen et al. 2017]. © 2017 IEEE. Reprinted, with permission, from S. K. Jensen, T. B. Pedersen, and C. Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Trans. Knowl. and Data Eng.*, 29(11): 2581–2600. DOI: 10.1109/TKDE.2017.2740932.

1.2 Introduction

Sensors are increasingly used to monitor large industrial systems, and the ability to efficiently analyze sensor data enables automation, fault detection, remote management, and optimization at an unprecedented scale [Sharma et al. 2014]. For example, the sensors in a Boeing 787 can produce half a terabyte of data per flight [Ronkainen and Iivari 2015]. A similar trend can be observed in the consumer market with the deployment of smart appliances [Statista 2022]. Sensor data can inherently be represented as *time series*, bounded 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. Due to the volume of time series being produced, specialized methods and systems for efficient collection, transfer, storage, and analysis of time series have become a necessity [Bagnall et al. 2019, Echihabi et al. 2021, Palpanas 2015, 2016a,b, Palpanas and Beckmann 2019, Shafer et al. 2013, Sharma et al. 2014, Zoumpatianos and Palpanas 2018]. An overview of time series data mining are provided by [Esling and Agon 2012, Fakhrazari and Vakilzadian 2017, Fu 2011], while more details about sensor data management and data mining is provided by [Aggarwal 2013, 2015].

General-purpose Relational Database Management Systems (RDBMSs) have been invaluable for applications that require strong transactional guarantees. However, they are unable to handle the velocity and volume of sensor data being produced today [Bagnall et al. 2019, Palpanas 2015, 2016a,b, Palpanas and Beckmann 2019, Shafer et al. 2013, Zoumpatianos and Palpanas 2018]. Separate applications like Python, R, or SPSS must also often be used to analyze time series, as general-purpose RDBMSs lack the necessary optimizations and algorithms to efficiently do so [Bagnall et al. 2019, Palpanas 2015, 2016a,b, Palpanas and Beckmann 2019, Shafer et al. 2013, Zoumpatianos and Palpanas 2018]. As a remedy, *Time Series Management Systems (TSMSs)*[2] have been proposed. These systems are, e.g., optimized for monitoring industrial machinery, analyzing time series collected from scientific experiments, or being embedded into Internet of Things (IoT) devices. In this chapter, a TSMS is defined as any system that is developed or extended *specifically* to store and query time series. TSMSs are not a recent phenomenon and the limitations of RDBMSs, when used for time series management, have been demonstrated in the past. In the 1990s the system SEQ and the SQL-like query language SEQUIN were developed [Seshadri et al. 1996]. SEQ was designed specifically to manage sequential data using a data model [Seshadri et al. 1995] and a query optimizer that utilizes that the data is ordered sequences and not a set of tuples [Seshadri et al. 1994]. The system was implemented as an extension to the object-relational Database Management System (DBMS) PREDATOR and supported storing and querying relational and sequential data together [Seshadri et al. 1996]. While additional support for sequences was added to the SQL standard, e.g., through window queries, the development of query languages and

[2] Time Series Management Systems is one among multiple names for these systems that are commonly used in the literature, another common name is Time Series Databases.

TSMSs continued throughout the early 2000s. For example, the algebra and query language AQuery [Lerner and Shasha 2003] was proposed in which data is represented as ordered sequences that can be nested to represent structures similar to tables. Utilizing the AQuery data model and information provided as part of the query, e.g., sort order, novel methods for query optimization were implemented. In contrast to AQuery which uses a non-relational data model, SQL-TS is an extension to SQL specifically designed for querying sequences in the form of time series [Sadri et al. 2001a]. 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 a sequence can then be expressed as predicates in a WHERE clause. The Optimized Pattern Search (OPS) algorithm was also proposed, thus making complex pattern matching both simple to express and efficient to execute [Sadri et al. 2001b, 2004].

The decades of research into time series management have led to the development of expressive query languages and efficient data processing engines. However, these past generations of TSMSs do not exploit modern hardware such as many-core processors and the ubiquity of distributed computing. Thus, they are unable to efficiently process the volume of time series data being collected. These TSMSs are also generally not well optimized for specific types of time series or use cases as they were designed to be general-purpose [Stonebraker and Çetintemel 2005]. As a result, a new generation of TSMSs has been developed. This chapter provides an overview of the current 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 their limitations, and propose further research directions. To focus the survey, it only includes systems that: implement methods for both *storage and processing of time series*, are designed to manage *numerous time series*, and are documented through *peer-reviewed papers* that are published *in or after 2010*. As a consequence, the following types of systems are not included: systems that simply reuse a TSMS [Kamina and Aotani 2019, Kamina et al. 2021]; systems designed for spatiotemporal data management that focus on the spatial component of the data such as UltraMan [Ding et al. 2018] and MobilityDB [Bakli et al. 2019, Zimányi et al. 2020]; TSMSs that simply combine general-purpose systems [Hamadou et al. 2020, Huang et al. 2021]; TSMSs designed for low powered sensor nodes such as Antelope [Tsiftes and Dunkels 2011], HybridStore [Wang and Baras 2013], LittleD [Douglas and Lawrence 2014], StreamOp [Cuzzocrea et al. 2014], and WearDrive [Huang et al. 2015, Li et al. 2014]; TSMSs for which only pre-print papers have been published such as TritanDB [Siow et al. 2018]; proprietary TSMSs for which no papers have been published such as Amazon Timestream, Azure Time Series Insight, kdb+, Shapelets, and VMware Tanzu; and open-source TSMSs for which no papers have been published such as InfluxDB, Prometheus, Graphite, TimescaleDB, and Warp10. An overview of proprietary and open-source TSMSs is provided by [DB-Engines Ranking of Time Series DBMS], and a survey of open-source TSMSs is provided in [Bader et al. 2017].

The following method was used for collecting the papers: an unstructured search using Google Scholar was performed to get an overview of the research performed in the area of TSMSs and to determine the relevant conferences, terms specific to the research area, and relevant researchers. Using these papers, iterations of structured search were performed. Relevant papers found in each iteration were used as input for the next iteration. In each iteration relevant papers were located by going through the following sources for each paper:

- The references included in the paper.
- Newer papers citing the paper. The citing papers were found using Google Scholar.
- All conference proceedings or journal issues published in or after 2010 for the conference or journal in which the paper was presented or published. The most commonly used data management outlets were ACM SIGMOD, CIDR, IEEE Big Data, and PVLDB, while the USENIX conferences were the most commonly used system outlets.
- The publication history for each paper's author, which was found using a combination of DBLP and Google Scholar.

The rest of the chapter is organized as follows. Section 1.3 provides an overview of the TSMSs included in the survey and a description of the criteria the systems are classified by. Section 1.4 describes the systems that use an internal data store, Section 1.5 describes the systems that use an external data store, and Section 1.6 describes the systems that extend an existing DBMS. Section 1.7 proposes further research directions based on the limitations of the current TSMSs. Last, a summary and conclusion are presented in Section 1.8.

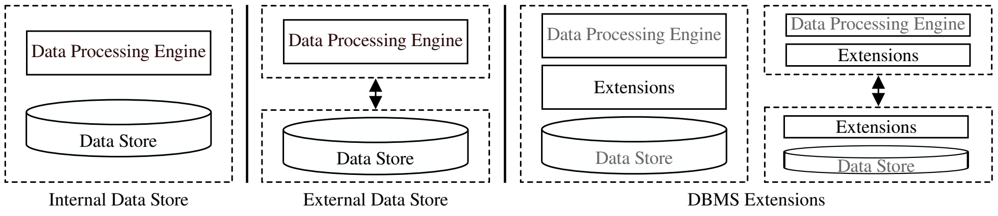


Figure 1.1: The three different types of architectures for TSMSs

1.3 Classification Criteria

An overview of the TSMSs included in this survey is shown in Table 1.1. It uses the classification criteria described below. The name of each system and the papers describing it are also included. The term *Unnamed* is used for systems without a name. Systems for which source code is available at the time of writing are listed with a star (☆). The information in the survey is primarily based on the referenced papers. The systems' source code and documentation are used as secondary sources when specific information is not in the papers. The TSMSs are grouped into three categories based on how the data processing engine and the data store are connected due to the large impact this architectural decision has on the implementation of the system. For example, using a mature external data store allows reuse of existing infrastructure, but the system is also limited to the data store's existing API, data model, and storage layout. The remaining classification criteria were selected based on how the systems differed. The full set of classification criteria are:

Architecture: The architecture of a TSMS is primarily based on how the data processing engine and the data store are connected as shown in Figure 1.1. For some systems, the data processing engine and the data store are integrated into the same application. For example, if a TSMS operates directly on files or uses an embeddable DBMS. Thus, these systems use an *internal data store*. Other systems use an *external data store*, e.g., a DBMS, a Distributed File System (DFS), or cloud storage. Finally, some TSMSs are implemented as *DBMS extensions* that add new functionality for storing and processing time series to existing DBMSs. Systems that use multiple data stores for different types of data are classified based on the data store they use for time series. Also, if a TSMS can use multiple different data stores for time series it is classified based on the system's primary use case. In Table 1.1 the architecture used by each group of TSMSs is written in italics as the heading for that section of the table.

Year: The year the latest paper documenting the TSMS was published. This is included to simplify the comparison of systems developed around the same time. The year of the latest paper is used as new papers indicate that functionality is still being added to the system.

Table 1.1: Overview of the surveyed systems in terms of the classification criteria

	Year	Primary Purpose	Motivational Use Case	Deployment	Maturity	Scale Shown	Data Processing Engine	API	Approximation	Latency	Data Store	Storage Layout
Internal Data Store												
tsdb☆ [Deri et al. 2012]	2012	Monitoring	Network monitoring at scale. Created for monitoring the Italian .it DNS registry.	Centralized	Mature	6.3 GB 1 Node	Proprietary (C)	Client Library (C)	Not Supported	Near Real-Time	BerkeleyDB	Fixed-size arrays of values compressed using QuickLZ and stored in BerkeleyDB by time.
FAQ [Khurana et al. 2014]	2014	Evaluation	Efficient approximate queries on time series with histograms as values.	Centralized	Proof-of-Concept	1.9 GB 1 Node	Proprietary (Java)	Unknown	Approximate Query Processing	Batch	KyotoCabinet	Sketches and histograms organized in a range tree.
NilmDB☆ [Paris et al. 2014]	2014	Data Analytics	Efficient management and analytics of time series in the energy domain.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Python)	Web Interface, HTTP, Python	Not Supported	Batch	Proprietary (Python)	Multivariate time series stored as fixed-size data points in binary files and metadata in SQLite.
Unnamed [Perera et al. 2015]	2015	Data Analytics	Fast approximate queries for decision support systems.	Centralized	Proof-of-Concept	Unknown 1 Node	Proprietary (R)	Extended SQL	Approximate Query Processing	Batch	Proprietary (R)	Time series stored separately as data points and different models.
RINSE [Zoumpatianos et al. 2015a]	2015	Evaluation	Efficient time series similarity search without creating a full index first.	Centralized	Demonstration	1 TB 1 Node	Proprietary (C)	Web Interface, Telnet	Approximate Query Processing	Batch	Proprietary (C)	Files with time series in an ASCII format that is indexed by ADS+.
RoundRobinson [Serra et al. 2016]	2016	Evaluation	Reference implementation of formalisms for TSMSs.	Centralized	Proof-of-Concept	Unknown 1 Node	Proprietary (Python)	Client Library (Python)	Lossy Compression	Batch	Proprietary (Python)	Python objects pickled to a file or written to CSV.
PhilDB☆ [MacDonald 2016]	2016	Data Analytics	Storage and analysis of versioned time series.	Centralized	Demonstration	119.12 MB 1 Node	Proprietary (Python), Pandas	Client Library (Python)	Not Supported	Batch	Proprietary (Python)	Data points stored as triples in binary files and updates in HDF5.
Chronos☆ [Chardin et al. 2016]	2016	Monitoring	Monitor hydroelectric plants using industrial PCs with CompactFlash for storage.	Centralized	Demonstration	11.5 GB 1 Node	Proprietary (C++)	Client Library (C++)	Not Supported	Near Real-Time	Proprietary (C++)	Data points stored quasi-sequentially and indexed by a B-Tree.
LittleTable [Rhea et al. 2017]	2017	Monitoring	Manage time series collected by remotely monitoring IoT devices.	Centralized	Mature	6.7 TB 1 Node	Proprietary (C++), SQLite	SQL	Not Supported	Near Real-Time	Proprietary (C++)	SSTables grouped by recency and compressed using LZO1X-1.
SummaryStore☆ [Agrawal and Vulimiri 2017]	2017	Data Analytics	Fast approximate analytics of time series on a single node using approximate representations.	Centralized	Demonstration	10 TB 1 Node	Proprietary (Java)	Client Library (Java)	Lossy Compression	Near Real-Time	Proprietary (Java), RocksDB	Approximated segments stored in RocksDB grouped by time series and ordered by time.
MDDS [Colmenares et al. 2017]	2017	Monitoring	Efficient and fast ingestion of multivariate time series on a single node.	Centralized	Demonstration	22.8 GB 1 Node	Proprietary (C++), SQLite	SQL	Not Supported	Near Real-Time	Proprietary (C++)	Data Records stored in row order and indexed by KD-Trees and an R^*-Tree.

Table 1.1: (continued)

	Year	**Primary Purpose**	**Motivational Use Case**	**Deployment**	**Maturity**	**Scale Shown**	**Data Processing Engine**	**API**	**Approximation**	**Latency**	**Data Store**	**Storage Layout**
TSMMDB [Lan et al. 2019]	2019	Data Analytics	In-memory analytics for IoT without the overhead of data transfer by embedding the TSMS in the application.	Centralized	Proof-of-Concept	92.16 GB 1 Node	Proprietary (C++)	Client Library (C++)	Not Supported	Near Real-Time	Proprietary (C++)	Memory-mapped files containing data points and indexing information for application-specific time intervals.
MetricQ⭐ [Ilsche et al. 2019]	2019	Monitoring	Provide efficient monitoring, simple extensibility, and efficient aggregate queries.	Distributed	Mature	13 TB 2 Nodes	Proprietary (C++)	Client Library (C++, Python, JavaScript), AMQP	Not Supported	Near Real-Time	Proprietary (C++)	Data points and multiple levels of aggregates in flat files ordered by time.
ChronicleDB [Seidemann and Seeger 2017, Seidemann et al. 2019]	2019	Monitoring	Provide both a very high ingestion rate, indexes that make ad-hoc queries efficient, and a recovery time within milliseconds.	Centralized	Demonstration	2.9 GB 1 Node	Proprietary (Java)	Client Library (Java), SQL-like	Not Supported	Near Real-Time	Proprietary (Java)	Log file storing blocks of data points organized in a column-based format compressed by LZ4.
Plato [Katsis et al. 2015, Lin et al. 2020]	2020	Data Analytics	Efficient model-based analytics of spatiotemporal sensor data using methods from signal processing.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Unknown)	ModelQL, InfinityQL	Approximate Query Processing	Batch	Proprietary (Unknown)	Tables storing data points and model coefficients using a built-in model data type.
AtriumDB [Goodwin et al. 2020]	2020	Data Analytics	Long-term storage of exact time series collected from medical equipment.	Centralized	Mature	820 GB 1 Node	Proprietary (PHP)	Client Library (Julia), REST API	Not Supported	Batch	Proprietary (PHP)	Data points stored in a column-based layout and compressed using delta compression and bzip2.
Timon [Cao et al. 2020]	2020	Monitoring	Scalable near real-time analytics of time series from a monitoring system despite out-of-order data points.	Distributed	Mature	Unknown 97 Node	Proprietary (C++)	TQL	Approximate Query Processing	Near Real-Time	Proprietary (C++)	Compressed SSTables sorted by time and a separate SSTable for out-of-order data points.
Apache IoTDB⭐ [Wang et al. 2020]	2020	Data Analytics	High ingestion rate, low latency queries, and advanced analytics on both the edge and in the cloud.	Distributed	Mature	Unknown 1 Node	Proprietary (Java)	Client Library (Go, C++, Python, Java), SQL-like, REST API	Not Supported	Near Real-Time	Proprietary (Java), HDFS	TsFiles with losslessly compressed data points in a columnar format and lightweight statistics.
TubeDB⭐ [Wöllauer et al. 2021]	2021	Data Analytics	Simplify management of climate sensor data for researchers that lack experience with computer science or the used sensors.	Centralized	Mature	664 MB 1 Node	Proprietary (Java)	Client Library (R), Web Interface	Approximate Query Processing	Batch	MapD	Compressed timestamp and value columns that are partitioned by station, sensor, and year and stored in MapD.
VergeDB [Paparrizos et al. 2021]	2021	Data Analytics	Adaptive compression of time series on the edge based on downstream requirement.	Centralized	Proof-of-Concept	Unknown 1 Node	Proprietary (Rust)	Unknown	Lossy Compression	Near Real-Time	RocksDB	Fixed-size arrays compressed by lossless or lossy compression

Table 1.1: (continued)

	Year	Primary Purpose	Motivational Use Case	Deployment	Maturity	Scale Shown	Data Processing Engine	API	Approximation	Latency	Data Store	Storage Layout
TS-NSM [Cai et al. 2021]	2021	Monitoring	Efficient management of time series from IoT using both solid-state drives and persistent memory.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Unknown)	Client Library (Unknown)	Not Supported	Near Real-Time	Proprietary (Unknown)	4K blocks storing regular segments as a timestamp and values compressed using a duplicate values counter and deltas.
Mach [Solleza et al. 2022]	2022	Monitoring	Ingesting from billions of data sources while also executing queries with low latency, especially for the most recent data points.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Rust)	Client Library (Rust)	Lossy Compression	Near Real-Time	Proprietary (Rust), ClickHouse, Amazon S3 Glacier	Fixed-size segments with a column-based layout stored compressed in blocks that match the OS's page-size.

Table 1.1: (continued)

	Year	Primary Purpose	Motivational Use Case	Deployment	Maturity	Scale Shown	Data Processing Engine	API	Approximation	Latency	Data Store	Storage Layout
External Data Store												
TSDS☆ [Weigel et al. 2010]	2010	Data Analytics	Querying time series in multiple formats from a single unified end-point.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Java)	REST API	Approximate Query Processing	Batch	Proprietary (Java)	Binary files with 64-bit floating-point values and HDF5 for time series, and NcML for metadata.
SensorGrid [Cuzzocrea and Saccà 2013]	2013	Data Analytics	Efficient online analytics of sensor data through AQP and grid computing.	Distributed	Demonstration	Unknown 5 Nodes	Proprietary (Unknown)	Web Interface, SQL	Approximate Query Processing	Real-Time	Microsoft SQL Server 2000	Data points and two-dimensional aggregates.
Respawn [Buevich et al. 2013]	2013	Monitoring	Executing low latency queries across both edge nodes and cloud nodes.	Distributed	Demonstration	21 GB 1 Nodes	Proprietary (Unknown), Bodytrack Datastore	HTTP	Not Supported	Near Real-Time	Proprietary (Unknown), Bodytrack Datastore	Bodytrack Datastore's binary format stored with multiple different levels of lossless compression.
Bolt☆ [Gupta et al. 2014]	2014	Monitoring	Simplify management and sharing of IoT data from connected smart homes.	Centralized	Demonstration	37.89 MB 1 Nodes	Proprietary (C#)	Client Library (C#)	Approximate Query Processing	Near Real-Time	Proprietary (C#), Amazon S3, Microsoft Azure	Data points on disk with in-memory indexes that map tags to timestamps and file offsets.
Druid☆ [Yang et al. 2014]	2014	Data Analytics	Scalable sub-second analytics of multivariate time series with metadata.	Distributed	Mature	10 TB 6 Nodes	Proprietary (Java)	HTTP	Approximate Query Processing	Near Real-Time	Proprietary (Java), DFS	Immutable segments stored as columns that are compressed using a method appropriate for each column's datatype.
Unnamed [Guo et al. 2013, 2014a,b]	2014	Evaluation	Indexing time series stored as models in a scalable distributed key-value store by both time and value.	Distributed	Proof-of-Concept	15 GB 9 Nodes	Proprietary (Unknown), Apache Hadoop	Unknown	Lossy Compression	Batch	Apache HBase	In-memory binary trees indexing models stored in Apache HBase.
Unnamed [Williams et al. 2014]	2014	Monitoring	Combining monitoring and analytics of time series from industrial installations.	Distributed	Mature	5 TB 46 Nodes	GE's Stream Processing Engine, Pivotal GemFire	Client Library (Java), OQL	Not Supported	Real-Time	Pivotal Gemfire	Fixed-size segments that are sorted by time and stored as linked lists.
Unnamed [Mickulicz et al. 2015]	2015	Data Analytics	Efficiently execute aggregate queries using hierarchies of pre-computed aggregates.	Distributed	Mature	3 TB Unknown	Proprietary (Java), MySQL	SQL	Approximate Query Processing	Near Real-Time	MySQL	Data points, simple aggregates, and sketches stored in MySQL tables.
servIoTicy☆ [Pérez and Carrera 2015]	2015	Monitoring	Integration of stream processing and data storage for processing IoT data.	Distributed	Demonstration	Unknown 16 Nodes	Proprietary (Java), Couchbase, Elasticsearch, Apache Storm	REST API	Not Supported	Real-Time	Couchbase	JSON documents that are stored in Couchbase and indexed by Elasticsearch.
Gorilla☆ [Pelkonen et al. 2015]	2015	Monitoring	Reducing query response time for data point recently collected from a real-time monitoring system.	Distributed	Mature	1.3 TB 80 Nodes	Proprietary (C++)	Client Library (Unknown)	Not Supported	Near Real-Time	Proprietary (C++), GlusterFS, Apache HBase	Segments containing timestamp and value deltas encoded using a variable number of bits.

Table 1.1: (continued)

	Year	**Primary Purpose**	**Motivational Use Case**	**Deployment**	**Maturity**	**Scale Shown**	**Data Processing Engine**	**API**	**Approximation**	**Latency**	**Data Store**	**Storage Layout**
Storacle [Cejka et al. 2015]	2015	Monitoring	Efficient monitoring of smart grids using edge nodes with limited hardware and scalable cloud storage.	Centralized	Demonstration	541 MB 1 Node	Proprietary (Java)	Client Library (Java)	Not Supported	Near Real-Time	Proprietary (Java), Cloud Storage	Java objects as the in-memory format and Protocol Buffers as the on-disk format.
BTrDB⭐ [Andersen and Culler 2016]	2016	Monitoring	Analyzing time series with nanosecond timestamps and out-of-order data points at multiple resolutions.	Distributed	Mature	2.757 TB 2 Nodes	Proprietary (Go)	Client Library (Go, Python)	Not Supported	Near Real-Time	Proprietary (Go), Ceph	Versioned copy-on-write trees with aggregates in the internal nodes and compressed data points in the leaf nodes.
FluteDB [Li et al. 2017, 2018]	2018	Monitoring	Efficient and reliable time series management in domains where very fast ingestion is a requirement.	Distributed	Proof-of-Concept	174.75 MB 1 Node	Proprietary (Unknown), PostgreSQL	SQL	Lossy Compression	Near Real-Time	Proprietary (Unknown), Persistent Storage, PostgreSQL, Redis	Timestamps compressed using delta-of-delta compression and values compressed using lossy XOR compression.
M-DB [Arora et al. 2019]	2019	Monitoring	Storage and periodic processing of data points from unreliable sensors.	Distributed	Demonstration	Unknown 10 Nodes	Apache Storm	Unknown	Approximate Query Processing	Near Real-Time	Apache Kafka, Apache Cassandra	Key-value pairs with multiple data points stored for each key.
UPS [Kosen et al. 2020]	2020	Data Analytics	Support both real-time monitoring and data analytics simultaneously.	Distributed	Demonstration	Unknown 11 Nodes	Proprietary (Go)	Client Library (Go)	Not Supported	Near Real-Time	Proprietary (Go), BTrDB, Ceph	Cache with three levels that store time series as data points or k-ary trees, and an on-disk format indexed by k-ary trees.
TimeCrypt⭐ [Burkhalter et al. 2020]	2020	Data Analytics	Support storage and sharing of private time series using an untrusted data store.	Centralized	Demonstration	Unknown 1 Nodes	Proprietary (Java)	Client Library (Java)	Approximate Query Processing	Batch	Apache Cassandra	Encrypted segments of data points indexed by a k-ary tree with statistics in the internal nodes.
Peregreen [Visheratin et al. 2020]	2020	Data Analytics	Efficient cloud-based time series management in terms of performance and cost.	Distributed	Mature	172 GB 7 Nodes	Proprietary (Go)	REST API	Approximate Query Processing	Batch	Proprietary (Go), HDFS, Amazon S3	Segments of data points stored in a row-based layout and compressed using delta compression followed by ZStandard.
Db2 Event Store [Garcia-Arellano et al. 2020]	2020	Data Analytics	Efficient ingestion, storage, and complex real-time analytics of time series stored in an open storage format in the cloud.	Distributed	Mature	Unknown 9 Nodes	Proprietary (C++), Db2 BLU	SQL	Not Supported	Near Real-Time	Proprietary (C++), Shared Storage	Apache Parquet files in shared storage accessible from the compute nodes, e.g., cloud storage, a DFS, or a NAS device.
Monarch [Adams et al. 2020]	2020	Monitoring	Unify monitoring across Google's internal systems.	Distributed	Mature	750 TB Unknown	Proprietary (Unknown)	Monarch's Novel Query Language.	Not Supported	Near Real-Time	Proprietary (Unknown), Colossus, Long-term Repository	Timestamps that are stored once for multiple time series and values compressed by RLE and delta compression.

Table 1.1: (continued)

	Year	**Primary Purpose**	**Motivational Use Case**	**Deployment**	**Maturity**	**Scale Shown**	**Data Processing Engine**	**API**	**Approximation**	**Latency**	**Data Store**	**Storage Layout**
TorqueDB [Garg et al. 2020]	2020	Monitoring	Reliable storage and efficient query processing on a combination of large edge nodes and small edge nodes.	Distributed	Demonstration	Unknown 15 Nodes	Proprietary (Java), InfluxDB	Flux	Not Supported	Near Real-Time	InfluxDB, ElfStore	Row-ordered byte-arrays with metadata and lightweight statistics.
ByteSeries [Shi et al. 2020]	2020	Monitoring	Efficient storage of many short time series with metadata collected by monitoring short tasks.	Distributed	Mature	12.5 GB 1 Node	Proprietary (C++)	SQL	Not Supported	Near Real-Time	Proprietary (C++), HDFS	Compressed data points ordered by their key and indexed by a trie and two compressed arrays.
BitemporalDB [Sedighi et al. 2020]	2020	Evaluation	Versioning of time series by creating new data points when updates are performed.	Centralized	Proof-of-Concept	Unknown 1 Node	Proprietary (Python)	Unknown	Not Supported	Near Real-Time	Azure Cosmos DB	Lists of As-Of and As-At data points in Azure Cosmos DB columns.
ModelarDB☆ [Jensen et al. 2018, 2019, 2021, 2023]	2023	Data Analytics	Efficient ingestion, storage, and analytics of groups of very high-frequent regular time series across the edge and the cloud using models.	Distributed	Demonstration	83.90 GB 33 Nodes	Proprietary (Java, Scala), H2, Apache Spark	SQL	Lossy Compression	Near Real-Time	Proprietary (Java, Scala), RDBMS, Apache Cassandra, HDFS	Segments with metadata and models of regular time series groups stored in a manner optimized for each data store.

Table 1.1: (continued)

	Year	Primary Purpose	Motivational Use Case	Deployment	Maturity	Scale Shown	Data Processing Engine	API	Approximation	Latency	Data Store	Storage Layout
DBMS Extensions												
F^{2}DB [Fischer et al. 2012b]	2012	Data Analytics	Effective and efficient forecasting of time series stored in an RDBMS.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Unknown), PostgreSQL	Extended SQL	Not Supported	Near Real-Time	PostgreSQL	Data points stored in tables and models stored in a model pool with a model index on top.
TimeTravel [Khalefa et al. 2012]	2012	Data Analytics	Forecasting of power consumption in smart grids using a uniform interface for queries on historical and forecasted data points.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Unknown), PostgreSQL	SQL	Approximate Query Processing	Near Real-Time	PostgreSQL	Data points stored in sorted arrays with models organized in a hierarchical index on top.
Unnamed [Huang et al. 2014]	2014	Monitoring	Efficient support for time series data, relational data, and complex SQL queries.	Distributed	Mature	53.05 GB 1 Node	Proprietary (Unknown), IBM Informix	SQL	Lossy Compression	Near Real-Time	IBM Informix	Segments stored as values; timestamps and values; or device ids, timestamps, and values.
TRISTAN [Marascu et al. 2014]	2014	Data Analytics	Efficient low-latency analytics of noisy, irregular, and misaligned sensor data.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Unknown), HYRISE	Unknown	Approximate Query Processing	Near Real-Time	Proprietary (Unknown), HYRISE	Sparse representation of fixed-size segments created through online dictionary compression.
Unnamed [Bakkalian et al. 2016]	2016	Evaluation	Analysing time series as ordered sequences of events using linear functions.	Centralized	Proof-of-Concept	23 MB 1 Node	Proprietary (PL/SQL), Oracle Database	SQL	Approximate Query Processing	Batch	Oracle Database	Data points and model parameters in tables.
Chronix⭐ [Lautenschlager et al. 2015, 2017]	2017	Data Analytics	Efficient storage and analysis of time series collected by monitoring distributed systems.	Distributed	Demonstration	8.7 GB 1 Node	Proprietary (Java), Apache Solr	HTTP	Lossy Compression	Batch	Apache Solr	Segments stored using lossy compression for timestamps and lossless compression for values.
EdgeDB [Yang et al. 2019]	2019	Monitoring	Efficient management of time series on the edge by storing them in groups.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Go), BTrDB	Unknown	Lossy Compression	Near Real-Time	Proprietary (Go), BTrDB	Fixed-size segments that are compressed, grouped, and indexed by trees.
tspDB⭐ [Agarwal et al. 2020]	2020	Data Analytics	Incremental imputation and forecasting using matrix factorization in an RDBMS.	Centralized	Demonstration	Unknown 1 Node	Proprietary (Python), PostgreSQL	Extended SQL	Approximate Query Processing	Near Real-Time	PostgreSQL	Data points and model parameters in tables.
Heracles⭐ [Wang et al. 2021]	2021	Monitoring	Efficient storage of multivariate time series as a timestamp column and multiple value columns.	Centralized	Demonstration	Unknown 1 Node	Prometheus	PromQL	Not Supported	Near Real-Time	Proprietary (Go), Prometheus	Groups with a sequence of timestamp deltas and sequences of XORed values in aligned blocks.
TVStore⭐ [An et al. 2022]	2022	Data Analytics	Limit the amount of storage used to a user-defined upper bound while also preserving the accuracy of the time series as much as possible.	Distributed	Demonstration	Unknown 1 Node	Apache IoTDB	Client Library (Go, C++, Python, Java), SQL-like, REST API	Lossy Compression	Near Real-Time	Proprietary (Java), Apache IoTDB	Files consisting of pages with data points that are compressed using lossy and lossless compression, metadata, and statistics.

Primary Purpose: The primary type of workload the TSMS was designed and optimized for. A system can be designed for *monitoring* to assist with detection and correction of faulty behavior, extraction of value from time series through advanced *data analytics*, or exclusively as an *evaluation* of research into novel formalisms, architectures, and methods for TSMSs.

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

Deployment: The type of deployment the system is designed for. A system can be designed for a *centralized* deployment on a single node or to scale out through both *distributed* computing and storage. It is included due to the effect this decision has on the architecture of the system and the constraints a centralized system has in terms of scalability.

Maturity: The maturity of the system using a three-level scale: *proof-of-concept* systems implement only the functionality necessary to evaluate a new technique. *Demonstration* systems also implement the 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 be supported through either an open-source community or commercial support.

Scale Shown: Scale shown is used as a concrete measure of scalability and is defined as the *size of the largest data set* and the *highest number of physical nodes* a TSMS can manage as documented by the referenced papers. The size of the largest data set and the highest number of nodes need not be from the same experiment. The size of the data set is the amount of storage used by the system when storing the data set, thus, the use of compression affects the size of the data set. Results where the data set's size is documented as a number of data points are not included. This is because the size of data points differs significantly depending on the number of values they contain, the data types used, and the amount of metadata they contain.

Data Processing Engine: The data processing engine used by the system for querying, and if supported, analyzing the time series. Also included as it often provides the system's external interface. The *name of the system* is written if an existing system is used. For TSMSs where a new data processing engine has been developed, it is listed as *proprietary* with the implementation language added in parentheses. Multiple data processing engines are listed if a system supports more than one. Both the term proprietary and the name of a system are also listed if a significant amount of functionality for data processing has been implemented on top of or as extensions to an existing data processing engine.

API: The primary interface provided by the TSMS. It can be *unknown*, an *existing query language*, an *extended version of an existing query language*, a *new query language*, a *client* *library* with the implementation language added in parentheses, a *web service*, or a *web interface*. The interface of embeddable systems is listed as a client library.

Approximation: Describes the primary method for approximation that each system supports if any. Approximation can reduce the amount of storage required and the query processing time. It can be implemented either with or without error bound guarantees. Common approaches for approximation include: reducing the precision of timestamps and values, representing time series using models such as polynomial functions or sketches, and using only a sample of the data points in a time series to answer a query. Answering queries using pre-computed aggregates may also produce an approximate result, e.g., if the time interval the aggregates are pre-computed for and the time interval the aggregate query is restricted to do not match exactly. Approximation can be performed during ingestion by using *lossy compression* to approximate time series or during query processing by using *Approximate Query Processing (AQP)* to approximate query results. Systems that do not implement any functionality for approximation are listed as *not supported*. A survey of model-based management of time series is provided by [Sathe et al. 2013], while information about sampling and integration of AQP into a DBMS can be found in [Mozafari and Niu 2015].

Latency The degree to which a system can ingest and process time series online with low latency. A system is categorized as *batch* if a data set must be fully ingested offline or in large batches before it can be queried, as *near real-time* if it supports executing ad-hoc queries on a data set while ingesting data points of it, and *real-time* if it supports executing user-defined continuous queries using stream processing. For an overview of how stream processing differs from traditional query processing see [Babcock et al. 2002], while an in-depth introduction to stream processing is provided by [Garofalakis et al. 2016].

Data Store: The data store used by the system for storing time series. The *name of the system* is written if an existing system is used. For TSMSs where a new data store has been developed, it is listed as *proprietary* with the implementation language added in parentheses. Multiple data stores are listed if a system supports more than one. Both the term proprietary and the name of a system are also listed if a significant amount of functionality for data storage has been implemented on top of or as extensions to an existing data store.

Storage Layout The internal representation used by the TSMS for storing time series. Included due to the impact the internal representation has on the system's query processing capabilities and the amount of storage it uses.

An overview of the functionality provided by each TSMSs is shown in Table 1.2. It is split into three sections like Table 1.1 and the information in the table is also based on the referenced papers. To describe the functionality with a uniform set of well-known terms, SQL keywords are primarily used. The table uses black circles to show if a system implements functionality to: *append* data points to the end of a time series; *insert* data points into a time series at any location; *update* data points already part of a time series; *delete* data points or time series; restrict queries to *where* data points has specific timestamps, values, or metadata; compute *aggregates* from a time series; *join* multiple time series on timestamps, values, or metadata; *group by* timestamps, values, metadata, or a number of data points; and perform *data analytics*. A black circle is only shown in a cell if the referenced papers explicitly state that the TSMS implements that functionality, otherwise the cell is left empty. For *data analytics*, only the methods the referenced papers state as implemented in the corresponding systems are listed.

The following three sections describe the primary contributions of each system in relation to its motivational use case and the classification criteria. There is one section for each of the three architectures. To make the systems simpler to compare despite the papers using different terms, the following terms are used throughout the chapter: a *data point* is a timestamp and one or more values, a *time series* is a sequence of data points ordered by time, a *segment* is a sub-sequence of a time series, and a *univariate time series* has one value per data point while a *multivariate time series* has multiple values per data point. From a user's perspective, most TSMSs only support univariate time series, so it is only explicitly stated if a TSMS supports multivariate time series. Figures redrawn from the referenced papers are also used to describe the systems. Two types of figures are used depending on the contribution of each system: *architecture diagrams* and *method illustrations*. Architecture diagrams show system components as *boxes with full lines*, undefined connections as *lines* between components, data as *boxes with rounded corners*, data flow between components as *arrows*, logically related components inside *boxes with dotted lines* or as components separated by *dotted lines*, data storage as *cylinders*, nodes as components surrounded by *boxes with dashed lines*, and annotations as *text labels*. The definition of a component depends on the level of detail in the original figure. For an embedded system it may be a buffer pool while it may be a set of nodes for a large distributed system. Method illustrations are not as regular because they show different aspects of each system. The layout of data in memory is drawn as *boxes with full lines* while *squares with rounded corners* are nodes in trees. Constructs only used for a single figure are described as part of the figure.

Table 1.2: Overview of the surveyed systems in terms of their functionality

	Append	**Insert**	**Update**	**Delete**	**Where**	**Aggregates**	**Join**	**Group By**	**Data Analytics**
Internal Data Store									
tsdb☆ [Deri et al. 2012]	●	●	●	●	●				
FAQ [Khurana et al. 2014]					●	●			Cosine Similarity, Distinct Count, Empirical Entropy, Frequency Estimation, Frequency Moments, Histogram Distances, Jaccard Coefficient, p-Norms, Rank Correlation, Set Membership, Top-K
NilmDB☆ [Paris et al. 2014]	●			●	●	●		●	Transient Event Identification, User-Defined, Visualization
Unnamed [Perera et al. 2015]					●	●		●	Histogram
RINSE [Zoumpatianos et al. 2015a]	●			●	●				Similarity Search, Visualization
RoundRobinson [Serra et al. 2016]	●			●	●	●		●	
PhilDB☆ [MacDonald 2016]	●	●	●		●		●		Pandas
Chronos☆ [Chardin et al. 2016]	●	●	●	●	●				
LittleTable [Rhea et al. 2017]	●	●		●	●	●		●	
SummaryStore☆ [Agrawal and Vulimiri 2017]	●			●	●	●			Distinct Count, Frequency Estimation, Set Membership
MDDS [Colmenares et al. 2017]	●				●	●			
TSMMDB [Lan et al. 2019]	●			●	●				
MetricQ☆ [Ilsche et al. 2019]	●				●	●		●	Visualization
ChronicleDB [Seidemann and Seeger 2017, Seidemann et al. 2019]	●	●		●	●	●	●	●	Pattern Matching
Plato [Katsis et al. 2015, Lin et al. 2020]	●				●	●	●	●	Correlation, Interpolation, Standard Deviation
AtriumDB [Goodwin et al. 2020]	●				●		●		Query for Data Quality
Timon [Cao et al. 2020]	●	●			●	●	●	●	Anomaly Detection, Cauchy Distribution, Distinct Count, Mean Absolute Deviation, Median, Pearson Correlation, Percentiles, Standard Deviation
Apache IoTDB☆ [Wang et al. 2020]	●	●	●	●	●	●	●	●	Similarity Search

Table 1.2: (continued)

	Append	**Insert**	**Update**	**Delete**	**Where**	**Aggregates**	**Join**	**Group By**	**Data Analytics**
TubeDB☆ [Wöllauer et al. 2021]	●				●	●		●	Interpolation, Quality Control, User-Defined, Visualization
VergeDB [Paparrizos et al. 2021]	●				●	●			Clustering, Outlier Detection, Sampling
TS-NSM [Cai et al. 2021]	●				●				
Mach [Solleza et al. 2022]	●	●			●				
External Data Store									
TSDS☆ [Weigel et al. 2010]					●	●			Sampling
SensorGrid [Cuzzocrea and Saccà 2013]	●				●	●		●	Visualization
Respawn [Buevich et al. 2013]	●				●	●		●	
Bolt☆ [Gupta et al. 2014]	●			●	●				Sampling
Druid☆ [Yang et al. 2014]	●			●	●	●		●	Cardinality Estimation, Quantile Estimation
Unnamed [Guo et al. 2013, 2014a,b]	●				●				Interpolation
Unnamed [Williams et al. 2014]	●			●	●	●		●	Standard Deviation
Unnamed [Mickulicz et al. 2015]	●	●			●	●		●	Distinct Count, Frequency Estimation
servIoTicy☆ [Pérez and Carrera 2015]	●	●	●	●	●				
Gorilla☆ [Pelkonen et al. 2015]	●				●				Pearson Correlation
Storacle [Cejka et al. 2015]	●				●	●			
BTrDB☆ [Andersen and Culler 2016]	●	●		●	●	●		●	Versioning
FluteDB [Li et al. 2017, 2018]	●				●				
M-DB [Arora et al. 2019]	●	●		●	●	●	●	●	Event Detection, Prediction, User-Defined
UPS [Kosen et al. 2020]	●	●	●	●	●				

Table 1.2: (continued)

	Append	**Insert**	**Update**	**Delete**	**Where**	**Aggregates**	**Join**	**Group By**	**Data Analytics**
TimeCrypt☆ [Burkhalter et al. 2020]	●			●	●	●		●	Approximate Quantiles, Histogram, Prediction, Standard Deviation, Trend Detection, Variance
Peregreen [Visheratin et al. 2020]	●			●	●	●		●	Difference, Moving Average, Percentiles, Sampling, Standard Deviation
Db2 Event Store [Garcia-Arellano et al. 2020]	●			●	●	●	●	●	
Monarch [Adams et al. 2020]	●			●	●	●	●	●	Histogram
TorqueDB [Garg et al. 2020]	●				●	●		●	
ByteSeries [Shi et al. 2020]	●				●	●		●	
BitemporalDB [Sedighi et al. 2020]	●	●	●		●				Versioning
ModelarDB☆ [Jensen et al. 2018, 2019, 2021, 2023]	●				●	●	●	●	
DBMS Extensions									
F^{2}DB [Fischer et al. 2012b]	●				●	●	●	●	Forecasting
TimeTravel [Khalefa et al. 2012]	●				●	●	●		Forecasting
Unnamed [Huang et al. 2014]	●	●			●		●		
TRISTAN [Marascu et al. 2014]	●				●	●		●	Interpolation
Unnamed [Bakkalian et al. 2016]					●	●	●	●	Interpolation
Chronix☆ [Lautenschlager et al. 2015, 2017]	●			●	●	●		●	Derivative, Difference, FastDTW, Integral, Moving Average, Outlier Detection, Percentiles, SAX, Scaling, Standard Deviation, Trend Detection, User-Defined
EdgeDB [Yang et al. 2019]	●				●	●	●	●	
tspDB☆ [Agarwal et al. 2020]	●	●			●	●		●	Imputation, Forecasting
Heracles☆ [Wang et al. 2021]	●				●	●	●	●	
TVStore☆ [An et al. 2022]	●	●	●	●	●	●	●	●	Similarity Search

1.4 Internal Data Store

1.4.1 Overview

Implementing a TSMS with an internal data store allows the data processing engine to be tightly integrated with the data store. As the data store is only accessible from the TSMS, the API and storage layout of a new data store can be optimized exclusively for the data processing engine. However, if a new data store is developed, no prior knowledge exists about how to best use it. On the other hand, if an existing data store is used, existing knowledge about how to use it can be reused and development time may be reduced, but the TSMS may be restricted by the data store's API and storage layout. The use of an internal data store may also simplify deployment due to the reduction of external dependencies, but the TSMS cannot reuse existing infrastructure such as a distributed DBMS or a DFS. Also, systems with an internal data store generally do not allow the data processing engine and the data store to be scaled separately.

1.4.2 Systems

tsdb presented by **[Deri et al. 2012]** is a centralized open-source TSMS designed for monitoring the large quantity of requests the .it DNS registry receives. It is implemented as a C library that can be embedded and uses the embeddable key-value store BerkeleyDB as its data store. The use of BerkeleyDB provides a uniform mechanism for storing the time series and the metadata used by the system. tsdb requires that all time series must start at the same time, have the same sampling interval, and contain the same number of data points. Thus, the system essentially stores a single multivariate time series. These restrictions make it simpler to perform analyzes that use multiple time series, but it also means that tsdb is not applicable for domains where the time series are sampled at different or irregular intervals. When ingesting, tsdb first sets the timestamp to use for the next batch of data points. Then an in-memory array is allocated, and the values of the data points are written to it. Afterward, the in-memory array is split into chunks, compressed using QuickLZ, and stored in BerkeleyDB with the timestamp and chunk id as the key. tsdb does not implement a query language, so data points must be manually inserted and retrieved using the library's C API. It also does not support approximation.

The centralized TSMS *FAQ* proposed by **[Khurana et al. 2014]** uses sketches organized in a range tree to efficiently execute approximate queries on time series with histograms as values. The system consists of multiple components as shown in Figure 1.2. The Index contains multiple types of sketches and histograms. Multiple representations are used to support different types of queries and different trade-offs between the accuracy of the query result and the query response time. The current prototype requires that users manually tune the index as automatic selection of parameters is listed as future work. The Index Manager and Error Estimator are used by the Query Planner to select the best representation of the data for a given query. The representation that provides the lowest query response time and a result within the required

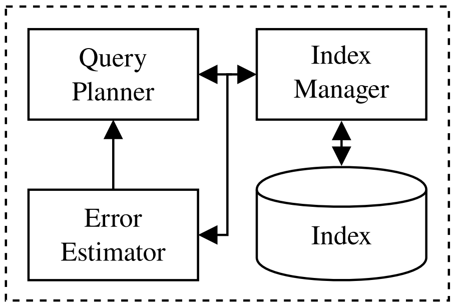


Figure 1.2: The architecture of FAQ, redrawn from [Khurana et al. 2014]

error bound is considered the best. Thus, FAQ supports approximation through AQP. However, detailed information about the Query Planner and the system's API is not included in the paper.

[Paris et al. 2014] designed *NilmDB* as a centralized open-source TSMS for managing multivariate time series from the energy domain. The system manages data points as Intervals, metadata that represents which time intervals each time series stores data points for. User-defined and system metadata are stored in SQLite, while the data points are managed by a novel append-only data store named BulkData. It stores each time series in a separate directory containing binary files of approximately 128 MB in size. During ingestion, the data points received for a specific time series are simply appended to the current file using a fixed number of bytes per data point. Specific data points from a time series can be retrieved from NilmDB using HTTP requests. User-defined Python scripts can also be submitted to process large data sets on the server before returning the result. NilmDB locates the data points by first retrieving all Intervals for the queried time series from SQLite. Then, the relevant Intervals are located using a red-black interval tree and the matching data points are retrieved from BulkData. When retrieving data points, BulkData exploits the files fixed layout to compute their location. Data points are deleted by storing their ids in a separate file and marking the space they occupy in the data file as unused. Both files are deleted when all data points in the data file have been deleted. A separate NILM Manager provides additional functionality, e.g., support for configuring NilmDB instances, transient event identification, and real-time visualization of time series from NilmDB. When creating visualizations, NILM Manager tries to retrieve time series with an appropriate level of aggregation. NilmDB does not support approximation.

A centralized TSMS with support for model-based AQP was proposed by **[Perera et al. 2015]**. The system represents each time series using a set of model types so an appropriate type of model can be used for each query. The data points can also be stored and used as a fallback if necessary. The system fits models to time series by splitting each time series into smaller segments and then attempts to fit models to each segment within an error bound. The use of segmentation allows models optimized for the structure of each segment to be used. If a model cannot be fitted to a segment with the necessary precision, the data points are stored for that segment. Multiple extensions to SQL are also proposed. Specifically, support for specifying which model type to use for answering the query, an error bound for the query result, and the maximum query response time. The current R-based implementation supports simple aggregate queries and performing histogram analysis. Approximation is supported for both query types, so the system supports AQP. Also, a Model Advisor is planned that will collect query statistics and use these to decide how a specific query should be executed within the constraints specified in the query. The authors also proposed using error bounded model-based approximation of time series to reduce the storage requirement of materialized OLAP cubes [Perera et al. 2016].

RINSE is a centralized TSMS proposed by **[Zoumpatianos et al. 2015a]**. It supports efficient time series similarity search using the adaptive tree-based ADS+ index [Zoumpatianos et al. 2014, 2016]. The system is split into two sets of components: a Frontend and a Backend. The Backend consists of NodeJS and the ADS+ Server that stores time series in an unspecified on-disk ASCII format indexed by ADS+. The Frontend provides a web interface that allows users to submit similarity search queries to the Backend by drawing the time series to search for using a mouse or a touch screen. The Frontend also provides a command-line interface based on telnet. As ADS+ is adaptive, it does not add the time series being indexed to its leaf nodes until they are requested by a query, thus reducing the amount of time required to create the index. Also, it dynamically adjusts the size of its leaf nodes to improve query performance. ADS+ supports executing exact and approximate similarity search queries using the same index. Thus, RINSE supports approximation though AQP. Following from the work on RINSE and ADS+, multiple new indexes for time series similarity search have been proposed [Palpanas 2020] and preliminary results have been published for NESTOR, a TSMS that automatically optimizes its storage layout based on the received queries [Zoumpatianos et al. 2019].

Pytsms and *RoundRobinson* were proposed by **[Serra et al. 2016]** as proof-of-concept implementations of two formalisms [Llusà-Serra et al. 2013, Serra et al. 2016]. Thus, no attempt was made to make them efficient or comparable in functionality to existing TSMSs. Pytsms is a simple centralized TSMS that provides a small set of operations that can be performed on time series. For example, computing the union of two time series and verifying that a time series is regular. RoundRobinson provides a multiresolution abstraction on top of Pytsms. This is implemented as Buffers and Discs. Buffers store data points for a time interval, e.g., five hours. Data points are added to the Buffers and aggregates are computed when data points outside the time interval are ingested. The aggregates are then added to Discs that function as fixed-size round-robin data stores, thus they discard old aggregates when new ones are added. The use of multiple Discs allows the system to provide a multiresolution view of a time series. For example, the most recent data can be stored using high-resolution aggregates, while older data are stored at much lower resolutions. An example of such a configuration is shown in Figure 1.3. It shows a schema for time series represented at different resolutions where the resolution used depends on the age of the data. The lowest resolution aggregates

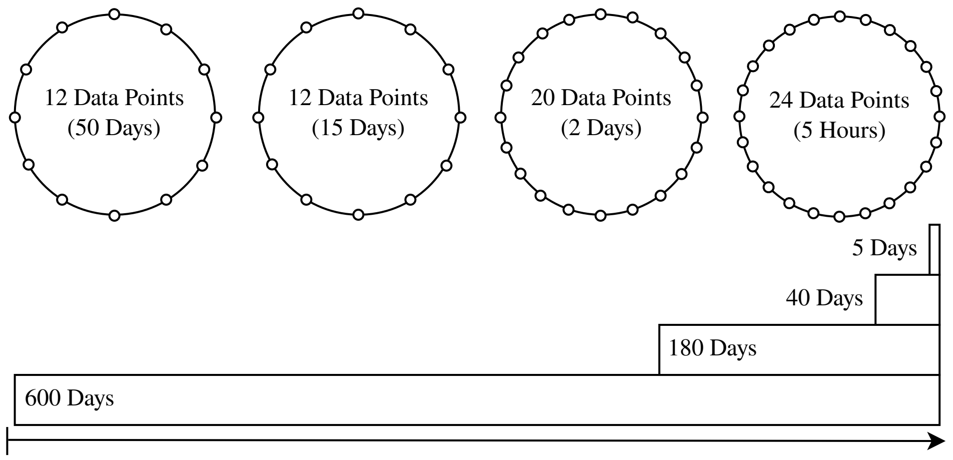


Figure 1.3: A time series stored at multiple resolutions, redrawn from [Serra et al. 2016]

are for fifty days so only twelve are needed for the entire six hundred days of the time series. However, the data for the latest five days are stored as twenty-four aggregates, one for each five-hour interval. Thus, RoundRobinson supports approximation through lossy compression.

PhilDB proposed by **[MacDonald 2016]** is a centralized open-source TSMS designed for data analytics that preserves the existing data points when updates are performed. It uses different data stores for each type of data. Metadata is stored in SQLite, e.g., time series are identified by user-defined strings and they can be associated with additional attributes such as their data source. Data points are stored sequentially in binary files where each data point is stored as a timestamp, a value, and an integer flag. The flag contains metadata about the data point, e.g., if the value is missing. Updates are stored in HDF5 files collocated with the binary files storing the data points. The HDF5 files store updated data points as two timestamps, a value, and an integer flag. The additional timestamp indicates when the update was performed. As the original data points are unchanged, queries can request a specific version of a time series based on a timestamp. Support for reading and writing time series to disk is implemented as part of PhilDB, however, the system uses Pandas as its in-memory representation. The use of Pandas allows PhilDB to interface with Python's data science ecosystem without converting the time series to another format first. However, the system does not support approximation.

The centralized open-source system *Chronos* by **[Chardin et al. 2016]** is a TSMS for monitoring industrial systems in hydroelectric power stations. In the power stations, the persistent storage is flash-based due to its endurance. However, it has substantially lower write performance than hard disk drives. To remedy this, Chronos implements an abstraction layer [Chardin et al. 2011] so that writes to the file system or the raw flash-based storage are performed close to each other. The addition of the abstraction layer makes the TSMS' write pattern close to sequential. This is preferable as the performance of a write to the flash-based storage decreases the further from the previous write it is performed. Chronos indexes the stored data through B-Trees which use a novel splitting algorithm that is optimized for time series. The system supports efficient out-of-order inserts within a limited window by buffering and sorting ingested data points before persisting them. The size of the buffers is a trade-off between the time interval for which out-of-order inserts are assumed to occur and the possibility of data loss as their data are only stored in memory. Chronos does not support approximation.

The TSMS *LittleTable* was developed at Cisco Meraki by **[Rhea et al. 2017]** to efficiently manage multivariate time series collected from customer IoT devices. It is designed as a centralized relational system for time series and requires that each table defines a composite primary key with a timestamp column as the last column in the key. However, the system lacks functionality commonly provided by RDBMSs, e.g., NULL values and support for updates. LittleTable's client is implemented using SQLite's Virtual Table Interface. When ingesting, SQLite batches data points before transferring them to LittleTable which inserts them into MemTables implemented as binary trees. Each MemTable stores data points for a predefined time interval so data points with very different timestamps are not stored together. The MemTables become read-only at a configurable size or age and are asynchronously written to disk as Sorted Strings Tables (SSTables). An SSTable contains key-value pairs ordered by the keys and was popularized by BigTable [Chang et al. 2006, 2008]. If LittleTable terminates abnormally, the clients try to re-insert the lost data points. Each SSTable also contains an index of the keys and time intervals it contains. The SSTables are grouped using the same time intervals as used for the MemTables and SSTables within each group are periodically merged. Queries are processed by a combination of the SQLite client and LittleTable. The client transfers the query to LittleTable which returns the requested data points in sorted order. The client then computes the final result if necessary. Query processing and ingestion can be performed in parallel, but LittleTable provides no guarantees about which of the data points currently being ingested will be in the query result. The system does not support approximation.

[Agrawal and Vulimiri 2017] designed the centralized open-source TSMS *SummaryStore* to provide efficient approximate analytics by storing approximate representations of time series. Thus, SummaryStore primarily supports approximation through lossy compression. The system is specifically designed for analytics in domains where the most recent data points are the most relevant. Important data points, such as outliers, can manually be stored separately to ensure they are preserved without any error. During ingestion, SummaryStore splits time series into segments and approximates each using multiple representations, e.g., lightweight statistics, histograms, Count-Min Sketches, and samples. This allows different types of queries to be answered efficiently. The representations to use for each time series can be configured, and additional representations can be added through an API. For a representation to be used with SummaryStore, it is a requirement that two instances of the representation can be combined. Segments and the representations they contain are combined as they age, so recent data is

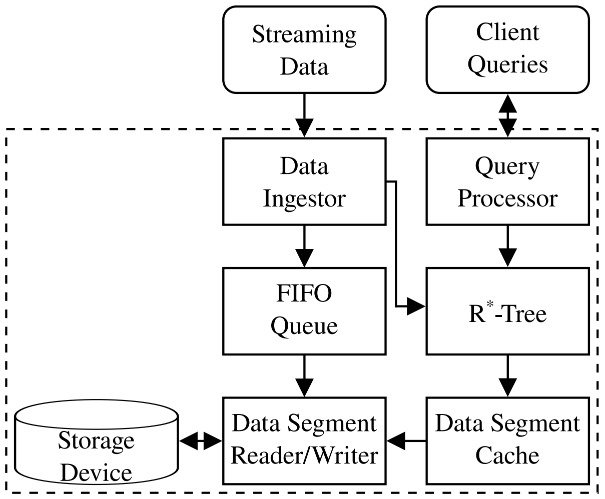


Figure 1.4: The architecture of MDDS, redrawn from [Colmenares et al. 2017]

stored in short segments while older data is stored in longer segments. As the same amount of storage is allocated for each segment, the data stored in the older segments is inherently less accurate. When answering queries, SummaryStore may approximate the result depending on the representation used. For example, when computing a sum over a time interval that spans half a segment, it is assumed that each part of the segment contributes equally to it. The error of each query result is estimated using the mean and variance of the deltas between adjacent timestamps and the mean and variance of the values for each time series. An in-memory tree mapping from time intervals to segments is used to make retrieving segments efficient.

MDDS was proposed by **[Colmenares et al. 2017]** as a centralized TSMS for managing multivariate time series with metadata. It is purposely optimized for fast ingestion on a single node instead of horizontal scalability. The system is designed for sensor data but generalizes data points to Data Records which are tuples containing fields. All Data Records in a time series must contain the same fields, and at least two of the fields must be numeric for indexing. The architecture of MDDS is shown in Figure 1.4 and consists of a separate ingestion and data processing pipeline with a shared R*-Tree and storage. MDDS ingests Data Records in micro-batches and divides these micro-batches into Data Segments. A Data Segment contains Data Records, lightweight statistics, and a KD-Tree that indexes the Data Records. Each Data Segment is added to the R*-Tree and asynchronously written to storage. The Query Processor is implemented using SQLite's Virtual Table Interface and executes queries by determining which Data Segments are relevant using the R*-Tree and then retrieving them from the cache or storage. MDDS requires that the user specifies the Data Records structure, the fields to index,

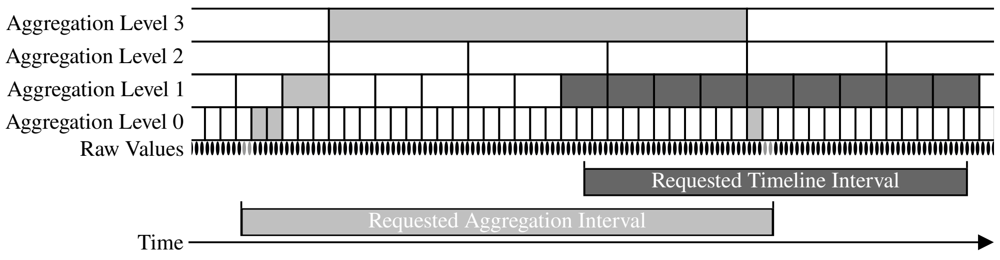


Figure 1.5: MetricQ's Hierarchical Timeline Aggregation, redrawn from [Ilsche et al. 2019]

the maximum micro-batch size, and the maximum Data Segment size. It stores Data Records in row order instead of column order for simplicity, and does not support approximation.

[Lan et al. 2019] designed *TSMMDB* to be a centralized in-memory TSMS for analyzing time series with metadata without transferring the data points from the TSMS to other applications. Instead, an application embeds TSMMDB and interacts with its Reader and Writer components directly. Configuration is performed using a Manager component. The system stores ingested data points in files. Each file contains both data points for a specific time interval and indexing information. These files are modified by memory mapping them and then modifying the corresponding memory. While new data points are flushed asynchronously to the disk, they can immediately be read from the mapped memory. TSMMDB maintains an in-memory mapping of metadata to files using red-black trees. A data retention policy can also be set to move older files out of the database. The system does not support approximation.

MetricQ is a distributed open-source TSMS designed by **[Ilsche et al. 2019]** for monitoring the LZR data center at Dresden University of Technology. It consists of multiple separate components that communicate over the RabbitMQ message-broker using AMQP messages and through JSON RPC. MetricQ stores metadata and configuration in CouchDB and provides access to it through a Manager. Data Sources ingest data points, encode them using Protocol Buffers, and add them to RabbitMQ. Sinks consume data points, e.g., for a dashboard or persistent storage. The message-broker guarantees that the messages will be received in order. Transformers operate as both a Data Source and a Sink as they apply operations to data points and then write back the result. The Sinks writing data points to persistent storage are specialized as they also process queries. MetricQ stores data points as timestamp and value pairs in files. The files are ordered by time, so new data points can be appended and data points for specific time intervals can be located using binary search. Multiple levels of pre-computed aggregates are also stored so MetricQ can answer aggregate queries efficiently as shown in Figure 1.5. The aggregates are computed for fixed time intervals at each level, so the location of an aggregate can be computed directly during query processing. Aggregate queries are answered using the highest levels that provide an aggregate within the requested time interval (light grey). Queries can also include the minimum resolution required, e.g, for visualizations, which also enables MetricQ to return aggregated values (dark grey). The system does not support approximation.

ChronicleDB is a centralized TSMS designed by **[Seidemann and Seeger 2017, Seidemann et al. 2019]** to provide a high ingestion rate for multivariate time series, fast ad-hoc queries, and fast recovery. The system can be embedded or used as a standalone server. To minimize random reads and writes, the log is used as the database and it is assumed that the time series the system ingests, generally, are ordered. The data points for each time series are ingested into a separate queue. A set of worker threads managed by a Load Scheduler writes the data points to disk in blocks. The data points are added to fixed-size L-Blocks which are compressed using LZ4 to produce C-Blocks. These are densely packed into fixed-size Macro Blocks and split if necessary. However, some storage in each Macro Block is reserved for out-of-order data points. As the C-Blocks are of different sizes, mappings from the L-Block ids to the corresponding C-Blocks' physical location are stored interleaved with the Macro Blocks as Address Translation Tree Blocks (ATT-Blocks). These are named TLB-Blocks in the original paper [Seidemann and Seeger 2017]. This allows the mappings and the data to be read as part of one sequential scan. For fast recovery, each ATT-Block keeps a reference to the previous ATT-Block. The data points are indexed by time using a novel Temporal Aggregated B^+-Tree (TAB^+). This is an extended B^+-Tree with timestamps as the keys, nodes the size of an L-block, and where each node contains references to both the next and previous nodes. ChronicleDB also stores user-extensible statistics in the TAB^+ nodes as a lightweight index on values. Log-structured indexes and Bloom Filters are also supported as secondary indexes. Depending on the workload, ChronicleDB can increase its ingestion rate by temporarily not updating the secondary indexes and by indexing the data points by the system's current time. If queries are commonly executed over time intervals of a specific size, e.g., a month, users can configure the system to create TAB^+s that match this time interval. ChronicleDB can also replay continuous queries from stream processing systems. However, the system does not support approximation. ChronicleDB was also used in a case study that evaluated multiple different methods for improving the performance of TSMSs through the use of persistent memory [Glombiewski et al. 2020].

Plato is a centralized TSMS proposed by **[Katsis et al. 2015]** that implements model-based AQP. The system provides similar functionality to an RDBMS and combines these features with methods from signal processing. Plato is designed for analytics and thus removes the need for exporting the time series to other tools such as R or SPSS. Plato's architecture consists of three layers as shown in Figure 1.6. Users can add new model types through the Extensibility Layer, fit models to time series through the Storage Layer, and query tables containing data points or models through the Query Layer. By providing an interface for implementing new model types, Plato can be extended for new domains. However, a user must manually select the model type to use for a time series as automated model selection is future work. Plato supports two SQL-like query languages: ModelQL and InfinityQL. ModelQL is designed for data analysts familiar with R or SPSS, while InfinityQL is designed for users familiar with

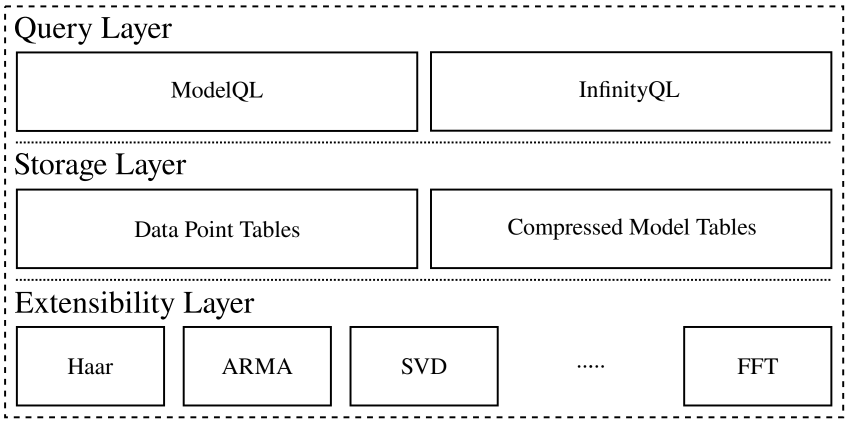


Figure 1.6: The architecture of Plato, redrawn from [Katsis et al. 2015]

SQL. Approximation is supported through AQP by executing queries using models. Queries are evaluated directly on models if they implement the functionality necessary for the query. Otherwise, data points are reconstructed from the models at the resolution necessary for the query. As future work, the authors propose that models are not used only as a method for representing time series, but also returned as the query result to provide information about the structure of the data. In [Lin et al. 2020], methods were proposed that enable Plato to provide results within deterministic error bounds for some queries. For example, error bounds can be provided for arithmetic operations, product, and inner product. It does so by segmenting time series into fixed-size or variable-size segments depending on a parameter, compressing each segment using a user-configurable model type that minimizes the Euclidean distance between the actual and approximated values, and associating one to three error measures with the segment. These measures are then used to provide the error bounds for the query results.

[Goodwin et al. 2020] proposed *AtriumDB* as a centralized TSMS for long-term storage and analytics of time series collected from medical equipment. It is designed to be device agnostic, provide lossless compression of high-frequency time series, and support efficient retrieval of data for parallel and distributed data processing. Existing software collects data points from the medical equipment, converts them to JSON, and stores them in RabbitMQ. AtriumDB retrieves the JSON messages and stores them in a separate file for each combination of bed, time series, and hour. At the end of each hour, the data points are compressed and the JSON files are archived on another server as a backup. The compression method splits the time series into segments with a user-configurable length and compresses the timestamps and values separately. The values are scaled to integers by a suitable power of ten. Then the timestamps and values are compressed using a set of unspecified pre-processing procedures, delta compression, and bzip2. AtriumDB uses the novel TSC file format as its on-disk format.

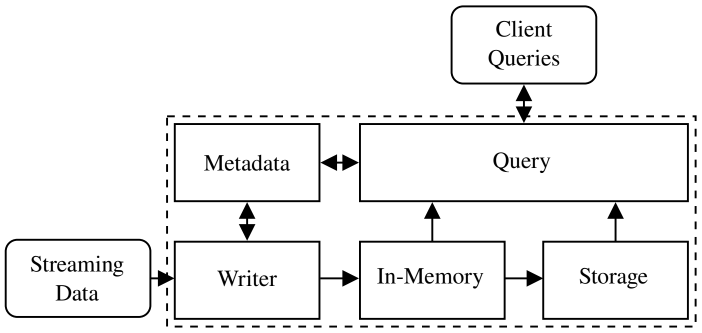


Figure 1.7: The architecture of Timon, redrawn from [Cao et al. 2020]

It consists of a header with information about the segments in the file followed by the segments. Each segment consists of a header followed by the compressed timestamps and values. The header contains information about the compression used and lightweight statistics about the data. The compressed TSC files are stored in a folder hierarchy organized by time. Information about the files and the time series, e.g., the scaling factors used and lightweight statistics about the values, are stored in MariaDB. Queries are given through a REST API from which JSON is returned, and they can only be executed on data points that have been written to TSC files. They are executed by mapping from metadata to files using the metadata in MariaDB, decompressing the required segments from the TSC files, removing irrelevant data points, and transferring the result to the user as JSON. Alternatively, a library written in Julia can be used to query the data points more efficiently as it operates directly on their binary representation without serializing the data points to JSON. As some use cases require time series that only contain a few or no gaps, users can query AtriumDB for time intervals where a time series contains gaps of at most a specified size. Users can also query AtriumDB for time intervals where data points have been collected for a specific set of time series. However, the system does not support approximation.

[Cao et al. 2020] implemented the distributed TSMS *Timon* as the data store for the real-time monitoring infrastructure at Alibaba Cloud [Cao et al. 2018]. The data points are initially added to a message queue before a stream processing system forwards them to Timon. The system supports multivariate time series with metadata as the monitored entities usually expose multiple metrics. It leverages that many queries can be answered by aggregates computed using associative and commutative operators to remove the need for a read-modify-write sequence when ingesting out-of-order data points. The architecture of Timon consists of five components as shown in Figure 1.7. The Writer component ingests data points in batches, maps their metadata to UUIDs using the Metadata component, and forwards the data points and UUIDs to the In-Memory component. As the message queue uses incrementing ids, duplicate data points can be detected by storing the highest observed ids. The In-Memory component performs

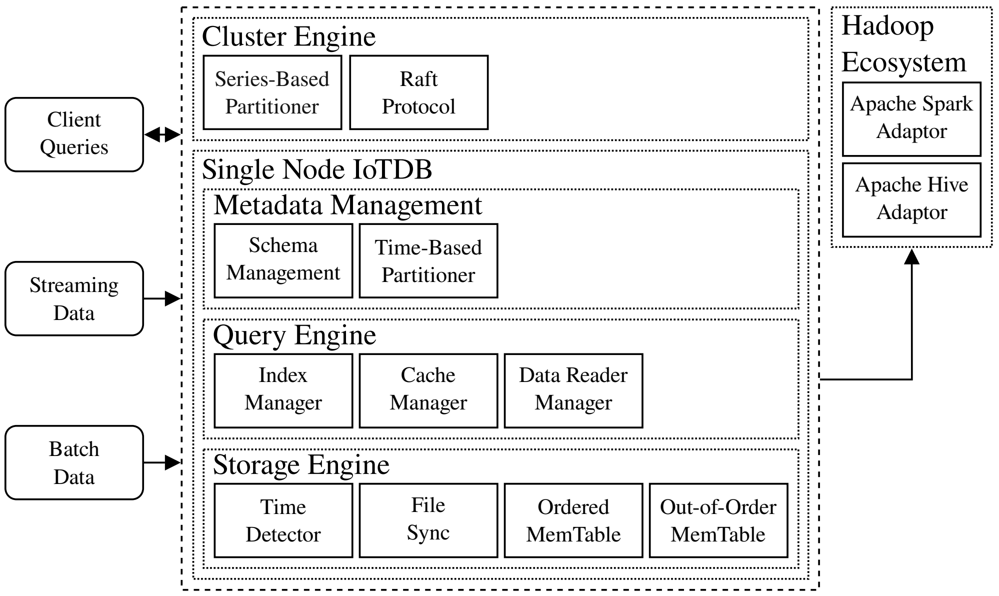


Figure 1.8: The architecture of Apache IoTDB, redrawn from [Wang et al. 2020]

write-ahead-logging and inserts the data points into MemTables with a row-based layout. Out-of-order data points are stored in separate MemTables. Initially, the MemTables store the data points in unordered pages that each contain data points for a specific time series. When a page is full, it is sorted and data points with equivalent timestamps are aggregated. The Storage component writes full MemTables to disk as SSTables with a column-based layout. Periodically, the SSTables are merged to reduce the number of small files on disk. The data points in the SSTables are indexed by time-partitioning k-ary trees, while the MemTables and SSTables are indexed by novel Time-Segment Log-Structured Merge-Trees. Queries are specified using TQL, an SQL-like language with pipes, and executed by the Query component together with the In-Memory and Storage components. Timon determines the resolution at which data points should be returned based on the requested time interval. Materialized views can be created and are assigned derived UUIDs. Thus, the Writer component will also update the relevant materialized views when forwarding data points to the In-Memory component. Timon supports approximation through AQP, e.g., by calculating distinct values using HyperLogLog sketches.

Apache IoTDB was proposed by **[Wang et al. 2020]** as a distributed open-source TSMS for the edge and cloud as shown by the system's architecture in Figure 1.8. Specifically, it was designed to be used as an embedded TSMS on small edge nodes, a standalone TSMS on large edge nodes, and a distributed TSMS in the cloud. The time series are uniquely identified by metadata organized in a tree structure. When ingesting, Apache IoTDB batches data points using MemTables before flushing them to disk as TsFiles. A TsFile is a novel column-based file format similar to Apache Parquet. It stores time series in sorted chunks containing multiple pages with a timestamp and a value column. The columns are compressed using lossless compression methods such as Gorilla's compression method and RLE. Snappy is also used for historical data. Lightweight statistics are stored for each file, chunk, and page to improve query performance. Out-of-order data points are stored in separate MemTables which are flushed to separate TsFiles and asynchronously merged with the other TsFiles. Similarly, updates and deletes are first written to differential files before being merged with the TsFiles. The TsFiles can be transferred to other Apache IoTDB instances using the File Sync component. Queries can be submitted through a REST API, over JDBC using an SQL-like query language, or using Apache IoT's native API. To improve the performance of analytical queries, Apache IoTDB uses the KV-match index [Wu et al. 2019] for sub-sequence similarity search and the PISA index [Huang et al. 2016] for aggregation. It can reuse existing infrastructure, e.g., TsFiles can be written to local storage or HDFS. Also, data processing engines like Apache Hive and Apache Spark can read TsFiles through libraries. The system does not support approximation.

[Wöllauer et al. 2021] designed the centralized open-source *TubeDB* to simplify management of sensor data from climate stations for researchers that lack experience with computer science or the used sensors. It is designed to have low system requirements and can be installed by simply extracting a ZIP archive. TubeDB can ingest the file formats exported by a variety of climate stations. Data points can also be pushed to TubeDB's HTTP interface. The ingested time series are stored in an ordered key-value store backed by MapDB. A collection is created for each climate station and sensor pair, and in each collection, the data points are partitioned by year. The timestamps and the values are compressed separately using quantization, delta-of-delta compression, sign-magnitude representation, and fastPFOR. Queries are specified through a web interface that dynamically visualizes the result when the query is modified. Alternatively, TubeDB can be queried through R using the rTubeDB package. The queries are executed by operators that are connected as specified in the web interface or through rTubeDB. The system implements data processing and analytics methods commonly used for climate research. TubeDB can, e.g., create regular time series from irregular time series using aggregation and approximate missing values using interpolation. Thus, it supports approximation through AQP. Many aspects of the system can also be extended through Java and a custom text format.

VergeDB is a centralized TSMS designed by **[Paparrizos et al. 2021]** for ingesting, storing, and analyzing time series on the edge. The architecture of VergeDB can be seen in Figure 1.9. Ingested data points are added to fixed-size arrays. When an array is full it is pushed to the Uncompressed Buffer, compressed using one or more compression methods, and added to the Compressed Buffer. The methods used depend on the configuration. The system supports different lossless and lossy compression methods. Thus, it supports approximation through lossy compression. A component is in development that will automatically select the compression methods to use based on the available resources, the ingestion rate, and the

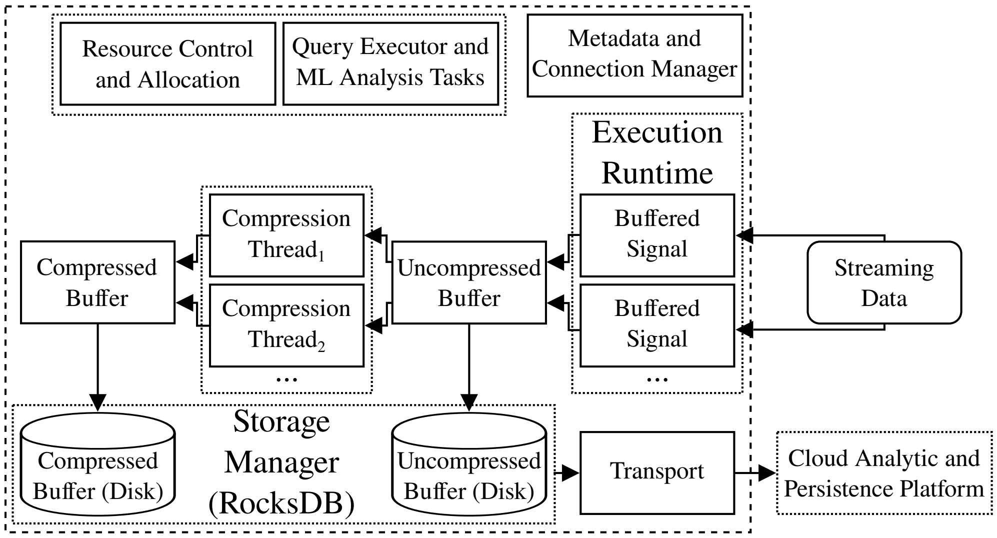

Resource Control and Allocation
Query Executor and ML Analysis Tasks
Metadata and Connection Manager
Execution Runtime
Compression Thread$_1$
Compression Thread$_2$
Compressed Buffer
Uncompressed Buffer
Buffered Signal
Buffered Signal
Streaming Data
Storage Manager (RocksDB)
Compressed Buffer (Disk)
Uncompressed Buffer (Disk)
Transport
Cloud Analytic and Persistence Platform

Figure 1.9: The architecture of VergeDB, redrawn from [Paparrizos et al. 2021]

analytics to be performed. RocksDB is used for storage and both the Uncompressed and Compressed Buffer can be written to disk. For query processing, VergeDB uses an unspecified query interface and supports multiple types of analytics, e.g., clustering and outlier detection.

[Cai et al. 2021] proposed *TS-NSM*, a centralized TSMS for IoT that is designed to efficiently use solid-state drives and persistent memory. As file systems add significant overhead to IO operations, the system is implemented as extensions to the NVMe and PMEM Linux drivers, thus bypassing the file system. The system's interface is a set of system calls added to the Linux kernel as shown in Figure 1.10. TS-NSM uses persistent memory as fast storage and stores data points in 4 KiB blocks. The data points of each time series are stored in separate blocks. Each block contains a header and scaled delta values as shown in Figure 1.11. The header contains the timestamp from the first data point, the value from the first data point, the number of duplicate instances of the value from the first data point, a reference to the previous block, and a reference to the next block. Only the first timestamp is stored as the time series are assumed to be regular. TS-NSM's compression component operates in two different modes. While the ingested data points contain the same value as is stored in the header, the values are stored by incrementing the duplicate counter. When a data point with another value is received, the compression component switches to storing the values as scaled deltas until the current block is full. The most recent data points are stored in persistent memory while older data points are transferred to solid-state drives by the Hierarchical Management component. Metadata, indexes, and the location of migrated blocks are also stored in persistent memory. The blocks in persistent memory are stored in a reverse linked list for each time series. A

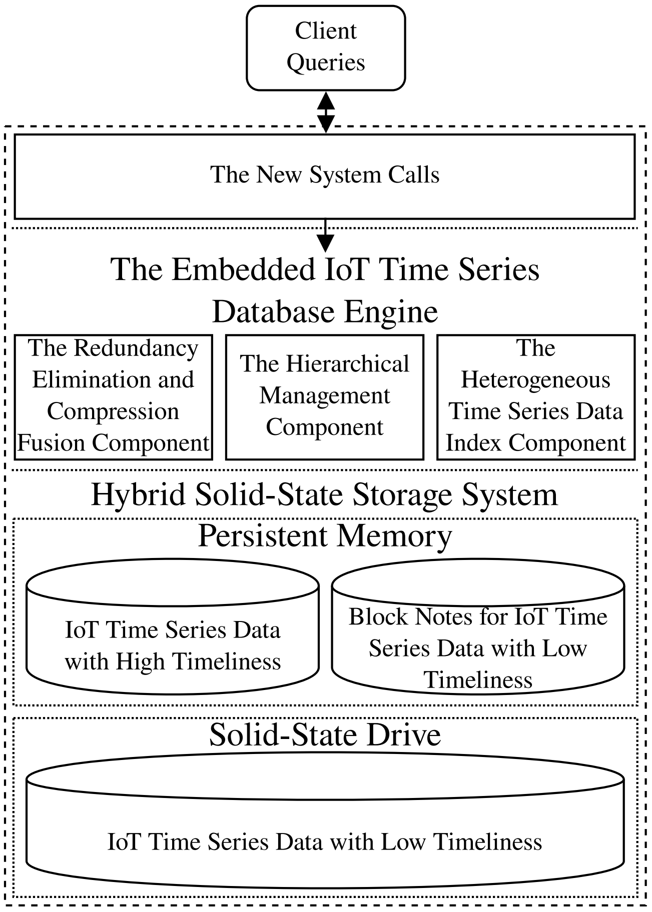

Client Queries
The New System Calls
The Embedded IoT Time Series Database Engine
The Redundancy Elimination and Compression Fusion Component
The Hierarchical Management Component
The Heterogeneous Time Series Data Index Component
Hybrid Solid-State Storage System
Persistent Memory
IoT Time Series Data with High Timeliness
Block Notes for IoT Time Series Data with Low Timeliness
Solid-State Drive
IoT Time Series Data with Low Timeliness

Figure 1.10: The architecture of TS-NSM, redrawn from [Cai et al. 2021]

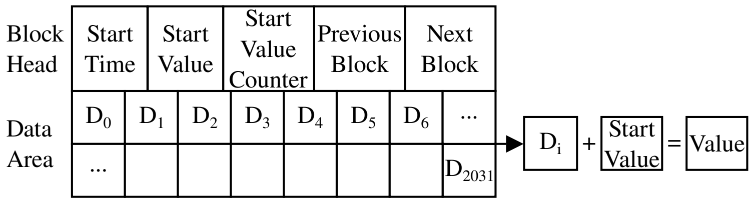


Figure 1.11: Structure of a TS-NSM block, redrawn from [Cai et al. 2021]

hash-based index maintains a mapping from time series ids to the matching lists. The locations of the migrated blocks are indexed using a B-Tree. TS-NSM does not support approximation. Both the throughput and latency of InfluxDB, OpenTSDB, and TS-NSM were improved by TS-PMEM, a version of the PMEM Linux driver optimized for TSMSs [Cai et al. 2022].

Mach is a centralized and embeddable TSMS implemented by **[Solleza et al. 2022]** to manage multivariate time series with metadata collected by monitoring systems. The system is designed based on experience with monitoring at Slack Technologies [Karumuri et al. 2020]. Mach is optimized for both managing large volumes of data points and ingesting from a large number of data sources simultaneously. Queries are assumed to predominately read recent data points. Segments that are rarely queried can be migrated to an external data store for analytics, e.g., ClickHouse, or archival, e.g., Amazon S3 Glacier. Mach is designed with a minimum of shared state so threads can operate independently. Thus, Mach can efficiently scale by simply increasing the number of threads. For ingestion, each data source is assigned to a writer thread. Each thread maintains a fixed-size Active Segment per data source and appends data points to it. Active Segments use a column-based layout. Out-of-order data points are assumed to rarely occur. If an out-of-order data point is received it is added to a separate buffer which is periodically merged with the other data points. When the Active Segment is full, it is compressed and stored in an Active Block that matches the operating system's page size. Mach allows a different compression method to be used for each column. Thus, methods optimized for a specific type of data can be used. Mach also supports approximation through lossy compression. When the Active Block is full, it is flushed to a thread-local file and its location is added to a global Block Index. If the compressed Active Segment is too large for the current Active Block, it is split and stored in multiple blocks. New files are created based on a size threshold. Queries are executed by first creating a snapshot representing the data points currently ingested for the queried data source. The snapshot is lightweight as it only consists of a few pointers and metadata. By creating a snapshot, the reader thread need not hold any locks during query execution. Then the relevant data points are retrieved from the Active Segment, Active Block, and the blocks persisted to disk. The locations of the relevant persisted blocks are retrieved from the Block Index. Mach does not cache the blocks retrieved from disk. Instead, it relies on the operating system's page cache to store the recently accessed blocks.

1.4.3 Discussion

A high-level overview of the surveyed TSMSs that use an internal data store is shown in Table 1.3. These systems are predominantly centralized with the only exceptions being MetricQ [Ilsche et al. 2019], Timon [Cao et al. 2020], and Apache IoTDB [Wang et al. 2020]. Instead, distributed TSMSs are generally implemented using an external data store as shown in Section 1.5. In addition, few of the TSMSs reuse existing systems as components. Of the twenty-two systems, nineteen implement an entirely new proprietary data processing engine, while seventeen implement an entirely new proprietary data store. One reason could

Table 1.3: High-level overview of the surveyed systems with an internal data store

	Year	**Primary Purpose**	**Deployment**	**Maturity**	**Approximation**	**Latency**
tsdb☆	2012	Monitoring	Centralized	Mature	Not Supported	Near Real-Time
FAQ	2014	Evaluation	Centralized	Proof-of-Concept	Approximate Query Processing	Batch
NilmDB☆	2014	Data Analytics	Centralized	Demonstration	Not Supported	Batch
Unnamed	2015	Data Analytics	Centralized	Proof-of-Concept	Approximate Query Processing	Batch
RINSE	2015	Evaluation	Centralized	Demonstration	Approximate Query Processing	Batch
RoundRobinson	2016	Evaluation	Centralized	Proof-of-Concept	Lossy Compression	Batch
PhilDB☆	2016	Data Analytics	Centralized	Demonstration	Not Supported	Batch
Chronos☆	2016	Monitoring	Centralized	Demonstration	Not Supported	Near Real-Time
LittleTable	2017	Monitoring	Centralized	Mature	Not Supported	Near Real-Time
SummaryStore☆	2017	Data Analytics	Centralized	Demonstration	Lossy Compression	Near Real-Time
MDDS	2017	Monitoring	Centralized	Demonstration	Not Supported	Near Real-Time
TSMMDB	2019	Data Analytics	Centralized	Proof-of-Concept	Not Supported	Near Real-Time
MetricQ☆	2019	Monitoring	Distributed	Mature	Not Supported	Near Real-Time
ChronicleDB	2019	Monitoring	Centralized	Demonstration	Not Supported	Near Real-Time
Plato	2020	Data Analytics	Centralized	Demonstration	Approximate Query Processing	Batch
AtriumDB	2020	Data Analytics	Centralized	Mature	Not Supported	Batch
Timon	2020	Monitoring	Distributed	Mature	Approximate Query Processing	Near Real-Time
Apache IoTDB☆	2020	Data Analytics	Distributed	Mature	Not Supported	Near Real-Time
TubeDB☆	2021	Data Analytics	Centralized	Mature	Approximate Query Processing	Batch
VergeDB	2021	Data Analytics	Centralized	Proof-of-Concept	Lossy Compression	Near Real-Time
TS-NSM	2021	Monitoring	Centralized	Demonstration	Not Supported	Near Real-Time
Mach	2022	Monitoring	Centralized	Demonstration	Lossy Compression	Near Real-Time

be that TSMSs that use an internal data store are primarily designed in this manner to allow for deep integration between a novel data processing engine and a novel internal data store. While existing systems are generally not reused, only five TSMSs have a proof-of-concept implementation: FAQ [Khurana et al. 2014], the system by [Perera et al. 2015], Pytsms and RoundRobinson [Serra et al. 2016], TSMMDB [Lan et al. 2019], and VergeDB [Paparrizos et al. 2021]. Ten systems are classified as demonstration in terms of maturity. For example, Chronos [Chardin et al. 2016], TS-NSM [Cai et al. 2021], and Mach [Solleza et al. 2022]. Surprisingly, seven of the twenty-two systems have mature implementations. For example, tsdb [Deri et al. 2012], MetricQ [Ilsche et al. 2019], and TubeDB [Wöllauer et al. 2021]. Approximation is supported by ten of the twenty-two systems. Six support AQP, while four support lossy compression. Of the systems with a mature implementation, only Timon [Cao et al. 2020] and TubeDB [Wöllauer et al. 2021] support approximation and both implement AQP. So AQP is, in general, the preferred approach for approximation. None of the systems with an internal data store support real-time data processing. In summary, most of the TSMS that use an internal data store are centralized and developed without reusing existing systems as components. Despite not reusing existing systems, only a few of the TSMSs have simple proof-of-concept implementations. Instead, the majority of the TSMSs with an internal data store have implementations that are robust enough to evaluate new methods and architectures with real-life use cases, and a significant number of them are also ready to be or are already used in production. However, approximation is not even implemented by half of the systems and only two of the TSMSs with a mature implementation support approximation. In addition, none of the systems support real-time data processing.

1.5 External Data Store

1.5.1 Overview

Implementing a TSMS with an external data store allows the data processing engine and the data store to be scaled separately. However, it also requires that multiple systems be deployed and managed. In addition, data must be transferred between these systems. In addition, it may be necessary to configure an external data store sub-optimally for the TSMS if it is also being used by other systems with different requirements. Using an existing external data store simplifies the deployment if existing infrastructure can be reused, significantly reduces development time, and existing knowledge about how to best use it may be reused. However, the TSMS will be restricted to the data store's API and storage layout. If a new data store is developed, its API and storage layout can be specifically optimized for the TSMS. However, no prior knowledge about how to best use it exists and no existing infrastructure can be reused.

1.5.2 Systems

TSDS developed by **[Weigel et al. 2010]** is a centralized open-source system designed to simplify time series analytics. TSDS operates as a query interface and data processing engine for multivariate time series stored in different formats. It can, e.g., access times series stored in ASCII files, RDBMSs, and a TSMS developed as part of TSDS named TSDB. The TSMS TSDB is primarily used for caching to reduce query response time. It stores each time series separately using either two or three different files. The values are always stored in a binary file. If the time series is not regular, then the timestamps are stored in another binary file. Both timestamps and values are stored as 64-bit floating-point values. Metadata, such as start time and end time, are stored in a separate file using the XML-based NcML format. HDF5 is used as an interim file format for long time series. TSDS provides a REST API as its query interface. Transformations, such as filtering and sampling, can be specified so only the time interval and resolution required are retrieved from the system. As a result, the system supports approximation through AQP. The output format can also be specified as part of the query.

SensorGrid is a grid framework proposed by **[Cuzzocrea and Saccà 2013]** that can execute exact and approximate aggregate queries on sensor data efficiently. It consists of sensors and two types of grid nodes: Stream Sources that ingest data points from the sensors, and Stream Servers that store the ingested data points and execute queries. To reduce query response time for aggregate queries, the Stream Servers pre-compute two-dimensional aggregates for the time and sensor dimensions. The time intervals to group by and how the sensors should be grouped must be defined based on the requirements of the domain. Succinct versions of these aggregates are copied to the other Stream Servers. Thus, a Stream Server can answer queries that require time series data managed by another Stream Server using multiple methods. It can redirect the entire query, use its local succinct aggregates, or decompose the query into multiple sub-queries and send them to other Stream Servers. If the pre-computed aggregates are

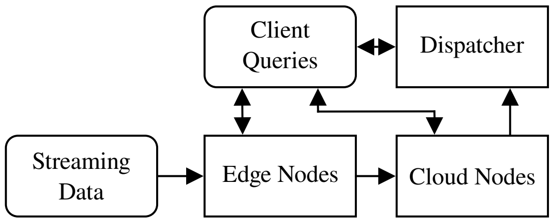


Figure 1.12: The architecture of Respawn, redrawn from [Buevich et al. 2013]

used, the result is approximate if the time interval of the query does not match the aggregates. Thus, approximation is supported through AQP. The framework also supports window and continuous queries. The SensorGrid grid framework was realized as a distributed TSMS and used for hydrogeology risk analysis at IRPI-CNR, by integrating the Stream Sources and Stream Servers with Microsoft SQL Server 2000 and a web interface for visualization.

The distributed TSMS *Respawn* presented by **[Buevich et al. 2013]** is designed to manage time series collected from large sensor networks. The system runs on both edge nodes and cloud nodes as shown in Figure 1.12. The edge nodes are ARM-based devices with a large amount of flash storage. They are placed at the edge of a sensor network and ingest the data points it produces. As its data store, Respawn uses the Bodytrack DataStore extended with lossless compression. It continuously computes aggregates at multiple resolutions to reduce query response time. A subset of the collected data points is continuously migrated to the cloud nodes using two preemptive strategies. Periodic Migration exploits that users usually use low-resolution data to determine which time intervals are of interest before requesting high-resolution data for that specific time interval. Thus, Respawn continuously migrates low-resolution segments to the cloud nodes. The system also continuously migrates segments based on their standard deviation using Proactive Migration. Standard deviation is used as it is efficient to compute and is often a good indicator of statistically important segments. A Dispatcher provides an HTTP query interface and routes queries to the relevant edge nodes and cloud nodes. The result is transferred from the nodes to the user as JSON over HTTP. While Respawn supports pre-aggregation, it does not support approximation. Respawn was utilized as a part of the Mortar.io platform for building automation infrastructure [Palmer et al. 2014].

Bolt was developed by **[Gupta et al. 2014]** and is a centralized open-source TSMS for multivariate time series that is designed to be easily embedded into IoT applications. The system stores time, metadata, and values as data points. Each data point consists of a timestamp and a collection of tag and value pairs. A value can be associated with multiple tags, and the number of values in a time series can change over time. Bolt's interface is based on encrypted streams produced by one writer and consumed by multiple readers. The writer splits the stream into segments with only the latest segment being mutable. Each segment contains a log file

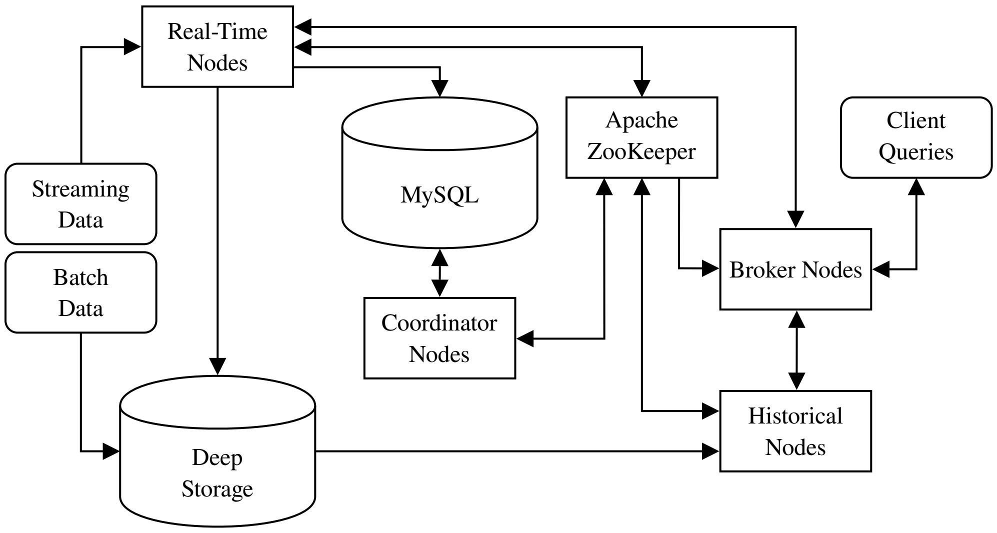
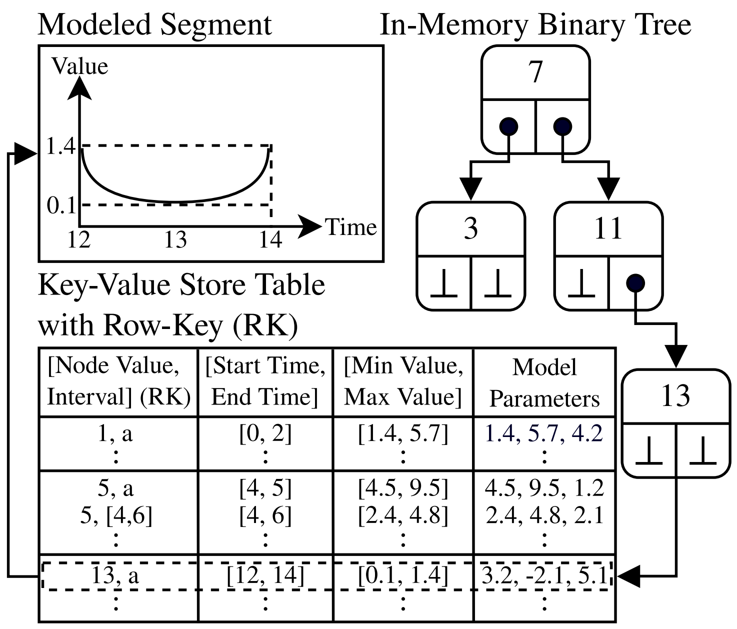


Figure 1.13: The architecture of Druid, redrawn from [Yang et al. 2014]

with the values and an index that maps each tag to a sorted list of timestamps and log file offsets, thus allowing queries for values with a specific tag or a specific tag in a given time interval to be executed efficiently. The entire index for the latest segment is kept in memory, while only a mapping from tags to the time intervals for which values are available is kept in memory for the immutable segments. The current segment is made immutable when the index reaches a pre-specified size. Readers can retrieve the latest values from a stream, values with a specific tag in a given time interval, or the tags in a given tag range. Readers can request that some data points be skipped when retrieving values with a specific tag in a given time interval. Thus, approximation is supported through AQP. Bolt also supports sharing data with other instances through the use of a trusted metadata server and untrusted cloud storage. The location of available streams, their metadata, and the application's keys are exchanged using the metadata server, while the encrypted streams are written to and read from cloud storage.

[Yang et al. 2014] implemented *Druid*, a distributed open-source TSMS that can efficiently ingest, manage, and perform data warehouse-style analytics with sub-second latency on multivariate time series with metadata. Druid is optimized for availability and uses a shared-nothing architecture. Apache ZooKeeper is used for coordination. A Druid cluster consists of multiple specialized node types as shown in Figure 1.13. Real-Time Nodes are designed to efficiently ingest data points and execute queries on recent data. The data points are initially stored in an in-memory row store and later written to an immutable columnar format on disk. Periodically a background task compacts the data point on disk into a large immutable segment and transfers it to Deep Storage which is usually a DFS. Historical Nodes retrieve segments



[Node Value, Interval] (RK)	[Start Time, End Time]	[Min Value, Max Value]	Model Parameters
1, a	[0, 2]	[1.4, 5.7]	1.4, 5.7, 4.2
⋮	⋮	⋮	⋮
5, a	[4, 5]	[4.5, 9.5]	4.5, 9.5, 1.2
5, [4,6]	[4, 6]	[2.4, 4.8]	2.4, 4.8, 2.1
⋮	⋮	⋮	⋮
13, a	[12, 14]	[0.1, 1.4]	3.2, -2.1, 5.1
⋮	⋮	⋮	⋮

Figure 1.14: Model-based query processing using an in-memory tree for indexing the segments as implemented by the TSMS by [Guo et al. 2013, 2014a,b], redrawn from [Guo et al. 2014a]

from Deep Storage and execute queries on the retrieved segments. They also maintain a local persistent cache of segments. Coordinator Nodes manage the cluster's configuration stored in a MySQL database and controls which segments each Historical Node should serve. Last, the Broker Nodes accept queries formatted as JSON through HTTP POST requests, route them to the relevant Real-Time Nodes and Historical Nodes, and merge their partial results. The partial results from the Historical Nodes are also cached and used to serve subsequent queries. Druid supports approximation through AQP, e.g, in the form of approximate quantile estimation.

[Guo et al. 2013, 2014a,b] proposed a distributed TSMS that stores data points as models in a key-value store. The model-based representation is created by splitting the time series into segments and approximating the values of each segment using a model. Thus, it supports approximation through lossy compression. The system consists of three components: a key-value store, an index consisting of two in-memory binary trees, and a MapReduce-based data processing engine. The index consists of two binary trees to efficiently index the segments by both their time and value interval. Similarly, two tables are created in the key-value store as one includes the start time and end time of the segments as part of its row-key, while the other table includes the minimum and maximum value of the segments as part of its row-key. Query processing is shown in Figure 1.14. First, the index is traversed to find relevant segments according to the time and value interval specified in the query. Then, the data processing engine retrieves the relevant segments from the key-value store and creates time series with a regular sampling interval. As each node in the trees can reference multiple segments, the mappers prune the list of candidate segments to remove those that do not match the query before the reducers construct data points with approximate values from the model-based representation.

[Williams et al. 2014] proposed a distributed system that is optimized for monitoring while also providing low-latency analytics for a limited amount of recent data. It is specifically designed for time series produced by sensors monitoring industrial installations. The system is based on Pivotal's Gemfire in-memory data grid to prevent hard disk drives from becoming a bottleneck. Data points are ingested by a stream processing platform that cleans them and performs real-time analytics. Afterward, the data points are partitioned by the entity from which they were collected and inserted into bins in the in-memory data grid. Each bin stores data points collected from a sensor for a fixed-size time interval using a doubly-linked list. Storing multiple data points together makes retrieval faster by reducing the number of bins and allow the data point's metadata to be stored once per bin. However, only data points for a few minutes are stored in each bin to ensure queries never have to retrieve large amounts of irrelevant data. The data points are stored in a doubly-linked list as it provides comparable read and write performance to a statically allocated circular buffer and allows memory usage to scale with the number of data points currently in the bin. As the TSMS is limited by the amount of available memory, integration with a disk-based data store for cold data and a unified query interface across both is future work. The system also does not support approximation. For more information about in-memory big data storage and analytics see [Zhang et al. 2015].

[Mickulicz et al. 2015] proposed a distributed TSMS for data analytics that can efficiently execute approximate aggregate queries over multivariate time series. The system is designed for analyzing events from mobile applications and consists of: a set of Aggregation Servers, multiple MySQL RDBMSs, and a Query Service that executes queries against the MySQL RDBMSs. During data ingestion, the system creates a hierarchy of aggregates based on a set of predefined aggregate queries, the time intervals the aggregates will be computed for, and their error bound. For example, a simple counter will be stored for queries that count specific events, while HyperLogLog can be used to approximately compute the number of distinct values, and Count-Min Sketches can be used for approximate frequency estimation. Thus, the system supports approximation through AQP. Each aggregate is stored with a start time and a duration to specify which part of the time series was aggregated. The hierarchy is organized with aggregates spanning the smallest time interval at the bottom and aggregates spanning the longest time interval at the top. The aggregates in each level are indexed by their start time using a binary search tree. Aggregate queries can be answered using the hierarchy by combining multiple pre-computed aggregates from the different levels. Thus, the hierarchy enables efficient execution of approximate aggregate queries for different time intervals.

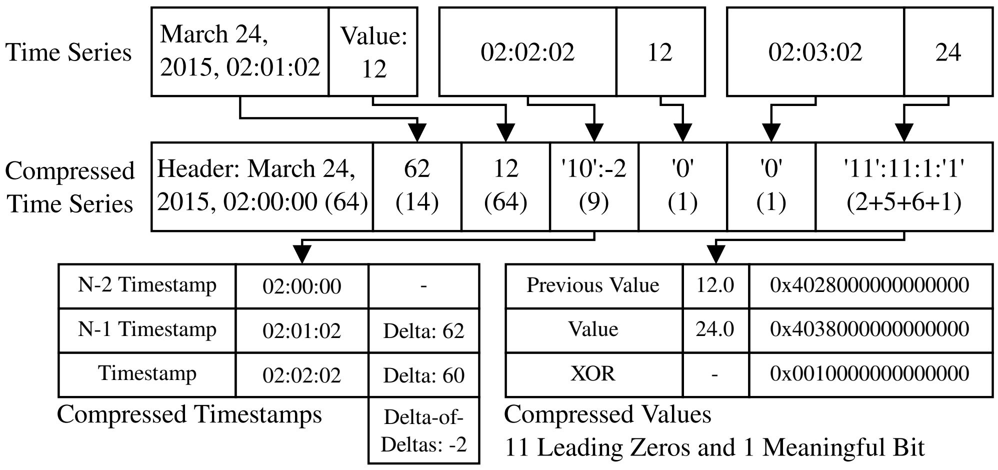


Figure 1.15: Gorilla's lossless compression method with bit patterns written in single quotes and the size of the values in bits written in parentheses, redrawn from [Pelkonen et al. 2015]

servIoTicy is a distributed open-source TSMS implemented by **[Pérez and Carrera 2015]** for storing multivariate time series with metadata from IoT devices. The system is split into a Frontend and a Backend. The Frontend provides the system's interface in the form of a REST API that serves JSON over HTTP. However, to increase the number of devices that can communicate with the system, the REST API is also accessible through STOMP and MQTT. The Backend provides storage and query processing using Couchbase, Elasticsearch, and Apache Storm. The data is stored in Couchbase using two types of JSON documents. The first is used to store metadata about the IoT devices the system is ingesting data from, while the second is used to store the data received from the IoT devices. To reduce query processing time, the documents in Couchbase are indexed by Elasticsearch. The integration of Apache Storm allows the system to provide stream processing using user-defined topologies. servIoTicy also implements support for dynamically modifying the code executed by an Apache Storm Bolt. As a result, an Apache Storm Bolt can execute different versions of its code depending on the data being processed. However, changes to a topology still require that the topology is stopped before it can be modified. servIoTicy does not support approximation, but was integrated with the web service discovery system iServe to augment the stored time series [Villalba et al. 2015].

[Pelkonen et al. 2015] implemented the distributed in-memory TSMS *Gorilla* at Facebook to more efficiently monitor their infrastructure. It is designed to reduce the query response time of an existing TSMS based on Apache HBase by operating as a cache that stores the data points collected in the last 26 hours. This time interval was chosen by analyzing the queries executed by the existing TSMS, and a cache was implemented instead of a replacement for

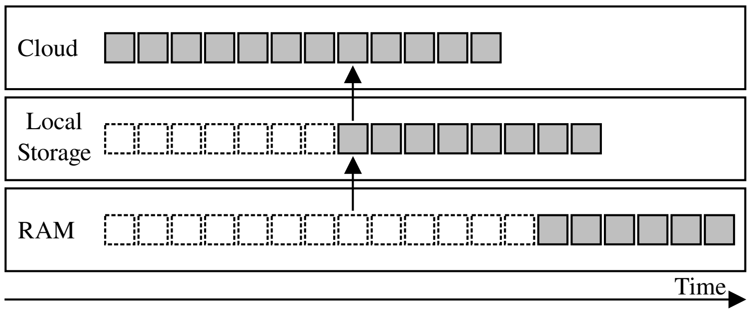


Figure 1.16: Storacle's tiered storage architecture, redrawn from [Cejka et al. 2015]

the existing TSMS as it contained petabytes of data. Data points in Gorilla consist of a key, a timestamp, and a value. The key is unique for each time series and is used for partitioning, thus ensuring each time series is mapped to a single host. Gorilla stores each time series as two-hour segments. Only one segment per time series is being appended to at a time while older segments are immutable. Gorilla uses a novel lossless compression method that stores timestamps and values interleaved as shown in Figure 1.15. Each segment has a header that contains the starting time of its two-hour window. The first data point's timestamp is stored as the delta between the segment's timestamp and the data point's timestamp. The first data point's value is then stored. Subsequent timestamps are compressed by first computing the deltas between the current and previous timestamps and then storing a delta of these deltas using a variable-length binary encoding. Values are compressed by first XOR'ing the current value with the previous value and then storing the result with the zero bits trimmed if possible. Both methods store a zero bit if the computed delta-of-delta or XOR value is zero. For persistence, Gorilla also writes the time series to the DFS GlusterFS. In addition, the data points are written to two separate Gorilla instances located in different data centers to increase their availability if the network is partitioned. However, no consistency guarantees are provided. Queries are executed through a client library that retrieves compressed segments from Gorilla. The low query latency provided by Gorilla has made new methods for time series analytics feasible. For example, tools have been developed that can search for correlated time series, perform real-time plotting, and periodically compute aggregates. Gorilla does not support approximation. Facebook also published an open-source version of Gorilla named Beringei, but it was last updated in 2018.

The centralized TSMS *Storacle* was developed for monitoring smart grids by **[Cejka et al. 2015]**. It is designed to be deployed throughout a smart grid on edge nodes with limited hardware to manage time series with metadata. The system uses Protocol Buffers as its storage format and a three-tiered storage model as shown in Figure 1.16. Thus, it is assumed that data points can be periodically transferred to the cloud for permanent storage and offline processing. The most recent data in each tier is not immediately deleted when it is transferred to the next

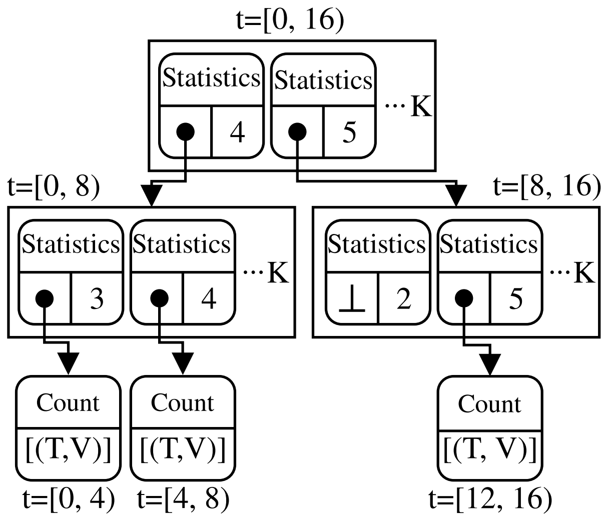


Figure 1.17: BTrDB's copy-on-write k-ary tree that contains data points in the leaf nodes and lightweight statistics in the internal nodes, redrawn from [Andersen and Culler 2016]

tier. Instead, it is temporarily stored in both tiers to provide higher durability without increasing latency when accessing the latest data points. Storacle also supports storing metadata in the form of tags, meta-fields, and lightweight statistics. Both tags and meta-fields are mutable lists of strings, but only tags can be used as part of a query. The lightweight statistics are purposely chosen so they can be updated using only the current data point. For example, the number of data points, their average value, and a histogram of the observed values. The system does not support approximation. Storacle's behavior is controlled through a set of parameters with functionality for dynamically adapting to the currently available resources being future work. Storacle was later integrated with a smart grid management framework [Faschang et al. 2017]

BTrDB was proposed by **[Andersen and Culler 2016]** and is a distributed open-source TSMS designed to manage high-frequency time series with nanosecond timestamps and out-of-order data points collected from high precision power meters. It stores these time series in Time-Partitioning Copy-on-Write Version-Annotated K-ary Trees. An example is shown in Figure 1.17. The leaf nodes contain a configurable number of data points while each internal node contains lightweight statistics for the data points in its sub-trees and version annotated references to its child nodes. The version number associated with each reference specifies when it was added. Each insert operation creates a new root node through which that version of the tree can be accessed. Thus, aggregate queries can efficiently be answered using the multiple levels of statistics, while the use of copy-on-write k-ary trees allows previous versions of the time series to be queried. The complete system consists of three applications. BTrDB provides

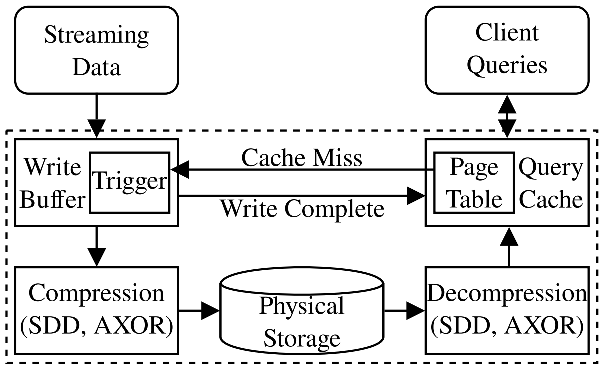


Figure 1.18: Ingestion and query processing in FluteDB, redrawn from [Li et al. 2017]

the query interface, data processing, data ingestion with buffering, and manipulation of the k-ary trees. A DFS in the form of CEPH is used to store the k-ary trees, while MongoDB is used to store metadata and parameters. Both the trees' internal and leaf nodes are compressed before they are stored. Each value is first reduced to a delta, then a delta to the mean of the previous deltas in the sequence, and finally stored using Huffman encoding. This method is only lossless for integer values so floating-point values are split into their mantissa and exponent before they are compressed. Thus, BTrDB does not support approximation. The TSMS was integrated with the DISTIL stream processing framework [Andersen et al. 2015].

[Li et al. 2017, 2018] proposed *FluteDB*, a distributed TSMS for multivariate time series that provides a high ingestion rate. FluteDB's ingestion and query processing pipeline is shown in Figure 1.18. Ingested data points are stored in a Write Buffer before being written to disk in batches. The Write Buffer is flushed if a query encounters a Query Cache miss, if the time interval the Write Buffer stores data for becomes too large, or if the amount of data stored in the Write Buffer becomes too large. The data points are also written to a log for durability. If a query causes the Write Buffer to be flushed, the system serves the query from the Write Buffer before it is flushed if possible. Afterward, it populates the Query Cache with the data. To improve the hit rate of the Query Cache, multiple adjacent data blocks are added to the cache instead of just the one requested. FluteDB stores the ingested data points in a columnar format and uses the novel Sliced Delta of Deltas (SDD) compression algorithm for timestamps and the novel AXOR compression algorithm for floating-point values. SDD compresses timestamps using a predefined delta so the delta-of-deltas can also be computed for the second value. Each of the delta-of-deltas is stored as multiple blocks containing a fixed number of bits. The size of the blocks is based on the average size of the delta-of-deltas. Each block stores one flag bit to indicate if it is the last block, while the remaining bits store the partial or entire delta-of-deltas. AXOR is a lossy compression method that converts each value to a 64-bit

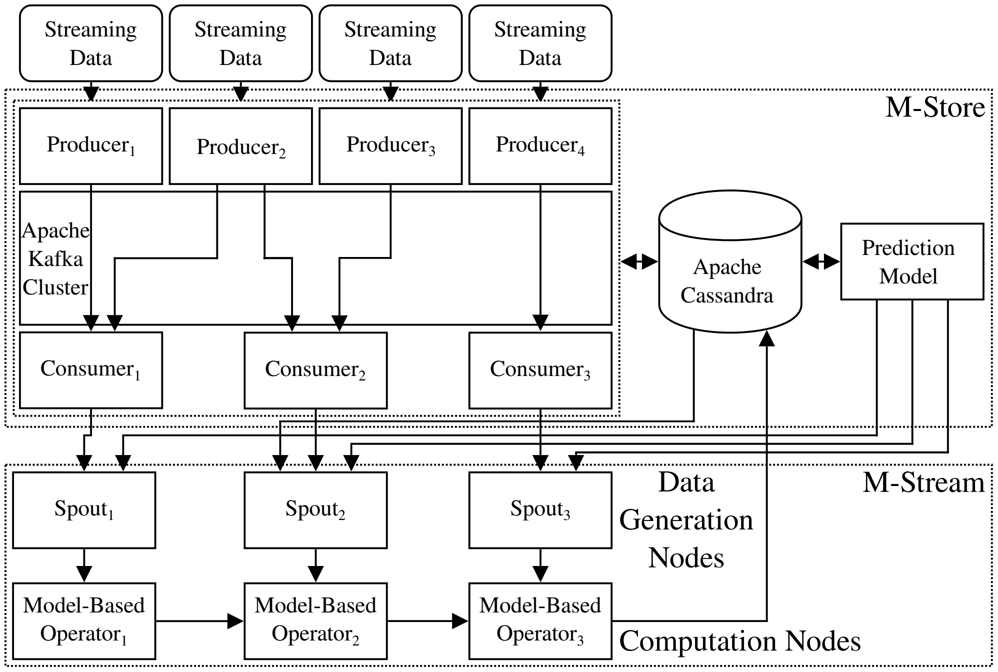


Figure 1.19: The architecture of M-DB, redrawn from [Arora et al. 2019]

floating-point number and sets a number of its trailing bits to zero based on the precision of the original value. Thus, FluteDB supports approximation through lossy compression. The initial value is stored in full, while subsequent values are XOR'd with the previous value and stored as three control bits, the start position of the first one bit, the number of bits from the first to the last one bit, and the sequence including all non-zero bits. The control bits indicate if the result of the XOR operation was zero, if the first one bit is at the same position as the previous value, and if the length of the sequence containing one bits is the same. For indexing, FluteDB uses a novel variant of the B*-Tree named a Triggered Time Series Merge Tree to index the time series by time. The latest data points are stored in in-memory trees, while the remaining are stored in trees on disk. The in-memory trees' are updated by appending data points to the trees' rightmost leaf nodes. FluteDB keeps the latest data points in memory when flushing an in-memory tree by splitting it into a cold and a hot data tree before flushing the cold data tree.

M-DB is a distributed TSMS proposed by **[Arora et al. 2019]** for storage and periodical processing of data points from unreliable sensors using user-defined functions. The system consists of two sets of components as shown in Figure 1.19. M-Store is an ingestion layer and data store implemented using Apache Kafka and Apache Cassandra, respectively. M-DB ingests data through Apache Kafka Producers that write the data points from each sensor to

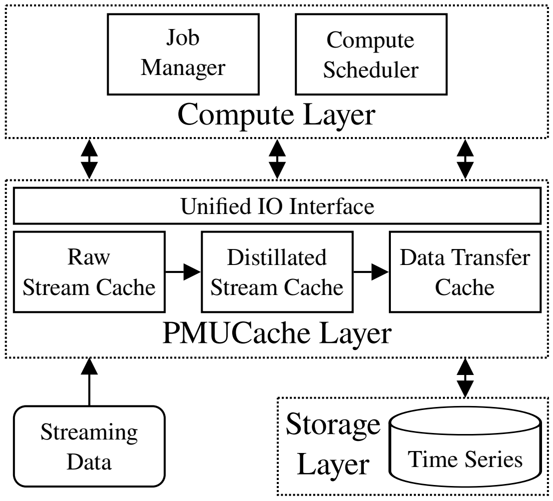


Figure 1.20: The architecture of UPS, redrawn from [Kosen et al. 2020]

different topics, while data requests are served through Apache Kafka Consumers. If missing values are requested from M-Store they can either be ignored or predicted using a user-defined prediction model. By default, a weighted moving average is used. Thus, the system supports approximation using AQP. For long-term persistence, the data points are also written to Apache Cassandra. Data points can automatically be deleted based on a time-to-live parameter. M-Stream is a stream processing component implemented using Apache Storm that periodically requests data points from M-Store and processes them using a pipeline of operators, each implemented as a user-defined Model-based Operator (MBO). M-DB supports four types of MBOs, each designed to transform a set of data points using a user-defined function. T-MBOs process a window of data points from a single sensor, S-MBOs process data points from multiple sensors with approximately the same timestamp, while TS-MBOs and ST-MBOs combine the functionality of T-MBOs and S-MBOs. As the data points collected from the sensors are considered unreliable and their values could have been predicted, each of the MBOs outputs both the result and the result confidence based on a user-defined confidence function.

[Kosen et al. 2020] proposed *UPS* as a distributed TSMS that can be used for both monitoring and data analytics simultaneously. The system consists of three sets of components as shown in Figure 1.20. The Compute Layer manages queries and organizes data transfers between the Storage Layer and the PMUCache Layer. It is implemented as a Job Manager that receives queries and adds them to a queue for scheduling, and a Compute Scheduler that determines when each query should be executed based on its metadata. The queries are primarily scheduled based on their priority classes (e.g., near real-time or offline analytics) and their quality-of-service requirements. Queries that do not define these are scheduled based on a

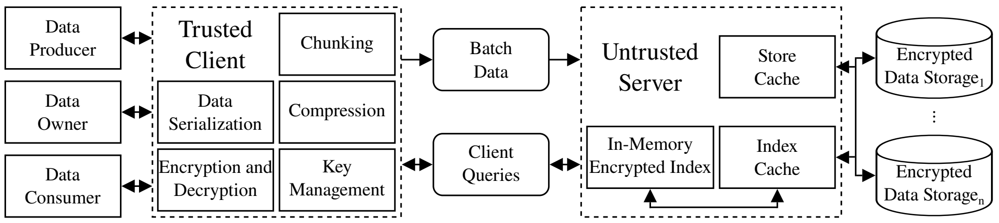


Figure 1.21: The architecture of TimeCrypt, redrawn from [Burkhalter et al. 2020]

best-effort policy. The Compute Scheduler also determines when data should be transferred to the caches based on the queries being executed. The amount of data being transferred depends on the queries' priority classes. For example, UPS transfers data points for twenty-four seconds for near real-time queries, while data points for one day are transferred for offline analytics. The PMUCache layer stores data points using three different caches to efficiently serve queries with different priority classes. The Raw Stream Cache (RSC) stores the ingested data points as key-value pairs, handles out-of-order data points through sorting, and deletes duplicate data points. The Distillated Stream Cache (DSC) receives batches of data points from the RSC and stores them in the copy-on-write k-ary trees proposed for BTrDB. When data is evicted from the DSC, it is moved to the Data Transfer Cache (DTC) and stored in an in-memory format suitable for high-throughput offline analytics. The data stored in the caches are managed by a Metadata Manager and stored on a set of cache nodes. UPS can change the amount of memory allocated for each cache based on the requirements of the queries and migrate data between the caches based on the queries received. Data evicted from DTC is moved to the Storage Layer, where it is stored in CEPH and indexed using the tree-based method proposed for BTrDB. Applications interact with UPS through a client library that uses the Unified IO Interface provided by the PMUCache Layer's Metadata Manager. The API provided consists of a small set of put, get, and delete methods that abstract away that data might be written to or retrieved from the caches or the Storage Layer. The system does not support approximation.

TimeCrypt is a centralized open-source TSMS proposed by **[Burkhalter et al. 2020]** that provides fine-grained access control and data sharing at multiple resolutions. The architecture of TimeCrypt is shown in Figure 1.21. Time series with metadata are ingested from Data Producers. During ingestion, they are split into fixed-size segments and each segment is encrypted with a different key before they are transferred to the data store. Encrypted lightweight statistics about the data points it contains are also computed for each segment. Thus, the segments and the lightweight statistics are stored encrypted and ordered by time in the data store. The keys are not stored in the data store to ensure that the data is not readable even if the data store is compromised. However, TimeCrypt can compute aggregates directly from the encrypted statistics. The segments are indexed by a k-ary tree with lightweight statistics in the internal

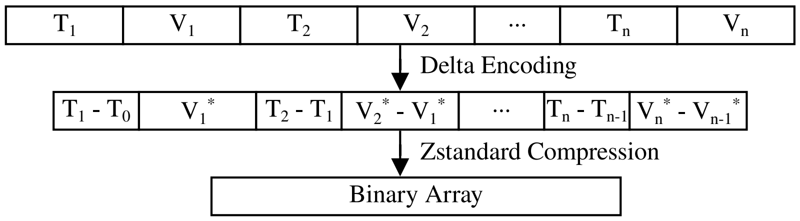


Figure 1.22: The compression method used by Peregreen, redrawn from [Visheratin et al. 2020]

nodes and data points in the leaf nodes. Thus, TimeCrypt can efficiently answer queries at multiple resolutions using the internal nodes. To share data, Data Owners can allow a Data Consumer to access data points for specific time intervals by creating a token encrypted with the Data Consumer's public key and then transfer it to the data store. Access can also be given to an aggregated time series without allowing the Data Consumer to access the individual time series that form the aggregate. It also supports computing approximate quantiles using AQP.

[Visheratin et al. 2020] designed the distributed TSMS *Peregreen* to efficiently manage time series using cloud storage as a data store. It was specifically designed to use Amazon S3. The system consists of three sets of components: Core, Modules, and Cluster. Core implements functionality for ingesting and compressing time series as well as uploading, deleting, and retrieving them from the data store. Modules are interfaces through which Peregreen can be extended with new functionality, e.g., new transformations can be implemented as methods that take values as input. The Cluster components allow nodes to operate together for scalability and fault tolerance. Peregreen is designed to ingest data points in batches to reduce the amount of data transferred to and from the data store. Each batch is split into Data Chunks that contain data points for a configurable time interval. For each Data Chunk, an Index Block is created that contains metadata and lightweight statistics for the Data Chunk. Index Blocks are stored in sorted Index Segments. Each Index Segment contains Index Blocks for a configurable time interval, metadata, and lightweight statistics computed from the Index Blocks. A sorted Index per time series stores the Index Segments together with metadata. Each Index is stored as files in the data store. The Data Chunks use a row-based layout to minimize the number of read requests to the data store. For compression, the timestamps and values are reduced to deltas and then compressed using Zstandard as shown in Figure 1.22. For the first data point, the previous timestamp (T_0) is computed from the metadata in the Index Segment. If the values are integers, the delta is computed using subtraction, while the difference between the binary representations is used for floating-point values. The compressed Data Chunks are appended to Data Segments which are written to the data store as files. Peregreen accepts queries through a REST API and supports two types of queries. Querying by values uses predicates that consist of the following three parts: lightweight statistic, operator, and value. More complex predicates can be created using conjunction and disjunction. The predicates are evaluated against the lightweight statistics stored in each part of the index. Extraction queries allow data points for specific time intervals to be retrieved for a time series. The index is used to efficiently compute the specific bytes to retrieve. Transformation, aggregation, and sampling can also be performed as part of an extraction query. Thus, Peregreen supports approximation through AQP. However, queries on multiple time series and a SQL-like query language are purposely not supported.

[Garcia-Arellano et al. 2020] proposed *Db2 Event Store* as a distributed cloud native TSMS for multivariate time series. The system is designed to provide high availability, high ingestion speed, efficient storage using an open format, and support for complex analytics. It consists of compute nodes with fast local storage backed by shared storage, e.g., a DFS or cloud storage. A hierarchy of caches with a lock-free epoch-based eviction policy is used to hide the latency of the shared storage. The system's catalog is stored in the shared storage and cached on an available node. Each table is split into partitions based on a user-defined key and each node is assigned a set of partitions to manage. If a node fails, the partitions it was responsible for are assigned to other nodes. Apache ZooKeeper is used for coordination. Internally Db2 Event Store stores the data points of each table in three immutable zones: the Log Zone, the Pre-Shared Zone, and the Shared Zone. The Log Zone contains recently ingested data points that are stored in the nodes' logs. To provide durability in the case of a node failure, data points are synchronously replicated to additional nodes during ingestion. The data points are asynchronously transferred to the Pre-Shared Zone in the form of small Apache Parquet files written to the shared storage. By using an open data format, software like Apache Spark, Python, and R can also access the data points. When the amount of data in the Pre-Shared Zone reaches a threshold, it is written to the Shared Zone as a few larger Apache Parquet files. The files in the Shared Zone are usually approximately 64 MB in size. Manual updates and deletes are not supported, but a time-to-live parameter can be set per table after which data points are deleted. To enable efficient pruning during query processing, UMZI indexes [Luo et al. 2019] are created for the Pre-Shared and Shared Zones. In addition, lightweight statistics stored in separate Apache Parquet files are maintained for the Shared Zone. To efficiently handle complex SQL queries, Db2 Event Store uses Db2's query optimizer and the Db2 BLU column-based data processing engine [Raman et al. 2013]. Initially, Db2 Event Store used Apache Spark as its data processing engine, but it was abandoned due to its high query latency and poor performance for complex queries. Db2 Event Store does not support approximation.

The distributed in-memory TSMS *Monarch* was proposed by **[Adams et al. 2020]** to provide monitoring for Google's internal infrastructure. It is designed so each team need not operate their own monitoring infrastructure and provides high availability, relational schemas, support for querying multivariate time series with metadata from multiple entities, and histograms. The architecture of Monarch is shown in Figure 1.23. It consists of monitoring Zones with a geographically replicated global management and data processing layer. Each Zone consists of independent clusters for reliability and can operate independently if necessary. It originally

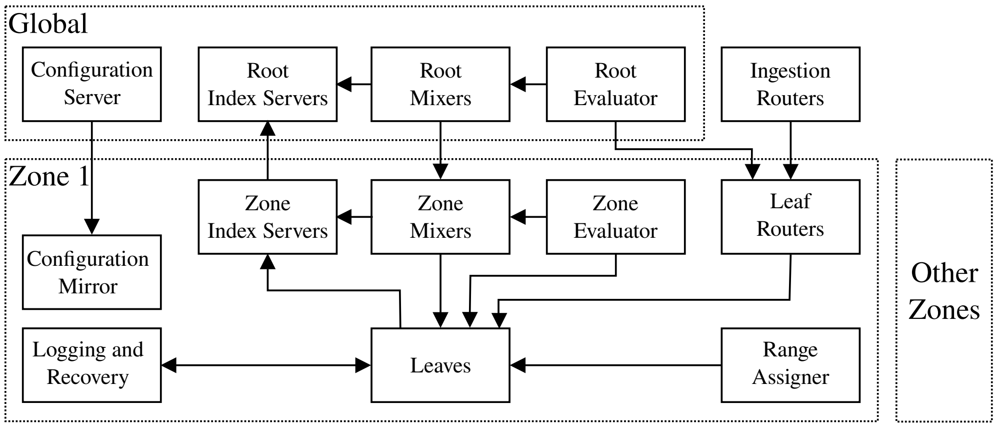


Figure 1.23: The architecture of Monarch, redrawn from [Adams et al. 2020]

used a pull-based approach for ingestion, but switched to a push-based approach to reduce complexity and improve scalability. The received data points are transferred to the correct Zone by Ingestion Routers, while Leaf Routers transfer the data points to the correct Leaves inside the Zone based on the partitioning provided by the Range Assigner. Time series from the same entity are stored on the same Leaf. To ensure high availability, Monarch is designed with as few external dependencies as possible. For example, Leaves write data points to memory and a DFS, but do not wait for the DFS to acknowledge. Thus, the system can operate even if the DFS is unavailable. The data points in the DFS are later moved to an external long-term data store. To reduce the amount of memory required, timestamps are only stored once per entity while delta compression and RLE are used to reduce the size of the values. Only lightweight compression is performed to balance CPU and memory usage. Monarch supports ad-hoc and periodic queries. They are specified as a sequence of operators using Monarch's novel query language. The periodic queries are issued by Evaluators and the results are written to Leaves. Queries are split into sub-queries by Mixers and executed by Leaves. To efficiently prune irrelevant data, a novel Field Hints Index is used to index the time series metadata. A Field Hint is a part of a metadata value, e.g., trigrams are commonly used. The index can efficiently prune based on exact predicates and regular expressions, but may return false positives. They are maintained for each Zone and Leaf and are stored in the Index Servers. Similar indexes are stored in the Leaves. Should network partitioning occur, the system continues to serve queries, but indicates that the data might be incomplete. However, Monarch does not support approximation. Configuration is managed by a global Configuration Server which is mirrored in each Zone. Monarch itself is monitored by an older known stable version of Monarch. While Monarch was designed to unify monitoring at Google, some services have requirements that could not be supported. For example, YouTube required a system that provided both reporting,

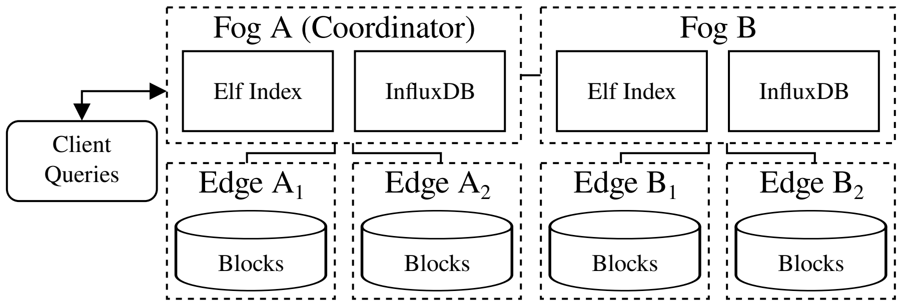


Figure 1.24: The architecture of TorqueDB, redrawn from [Garg et al. 2020]

dashboarding, real-time statistics, monitoring, and complex queries. Thus, they created a data processing engine with support for standard SQL named Procella [Chattopadhyay et al. 2019].

TorqueDB proposed by **[Garg et al. 2020]** is a distributed TSMS that is designed to be deployed on a combination of desktop-class fog nodes and Raspberry Pi-class edge nodes. The edge nodes run ElfStore [Monga et al. 2019] and are used for storage. Data points are ingested from sensors and then stored in ElfStore blocks. Metadata and lightweight statistics are also stored for each block and used to look up specific blocks. For reliability, the blocks are replicated across edge nodes based on a block-specific replication level. Each edge node is associated with a fog node. A fog node and its associated edge nodes form a private network with the fog node serving as a gateway to the other fog nodes and the internet as shown in Figure 1.24. Each fog node maintains an index of the block ids and the metadata for the blocks which are stored on its associated edge nodes. They also maintain an approximate index of the data stored on the edge nodes managed by other fog nodes using bloom filters. For query processing, TorqueDB retrieves relevant blocks from ElfStore and then uses InfluxDB instances deployed on the fog nodes to execute the queries. When a fog node receives a query, it becomes the query's Coordinator and uses the indexes to determine the ids of relevant blocks. The Coordinator can also optionally further filter these block ids based on the actual metadata of each block. The block ids and relevant sub-queries are then assigned to the available fog nodes. These fog nodes then load the blocks into InfluxDB, execute the query, and return the results to the Coordinator. If necessary, e.g., for aggregates, the Coordinator computes the final result. The blocks loaded into InfluxDB are purposely not deleted. Instead, InfluxDB is used as a cache and TorqueDB's query planner takes the location of cached blocks into account using a lazily updated mapping from block ids to fog nodes. TorqueDB does not support approximation.

[Shi et al. 2020] proposed the distributed in-memory TSMS *ByteSeries* as a replacement for the existing in-memory TSMS named tsdc used by ByteDance's infrastructure monitoring system. This monitoring system uses an in-memory TSMS to store the most recent data points while HDFS is used for long-term storage. As the monitored tasks often only run for a short

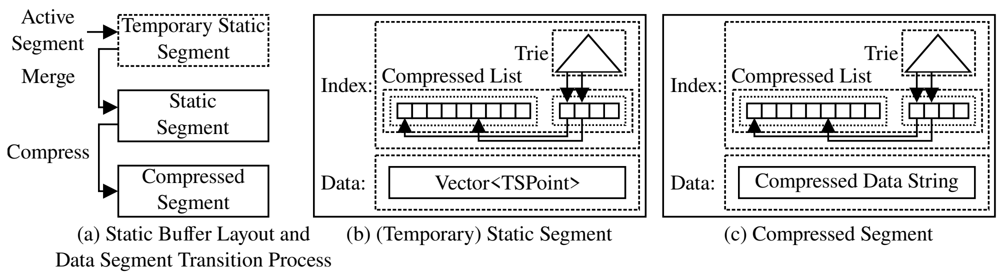


Figure 1.25: The segment types used by ByteSeries, redrawn from [Shi et al. 2020]

amount of time, the collected time series are usually very short. Thus, more than 80% of the storage used by tsdc was for metadata. To provide a high ingestion rate, ByteSeries ingests data points into an Active Buffer which consists of multiple independent uncompressed Active Segments. The Active Segment to use is selected based on the time series key and the data points are both appended to an array and inserted into a local inverted index. Asynchronously, the data points in the Active Segments are compressed and added to a Static Buffer as shown in Figure 1.25. The decision to move data from the Active Buffer to the Static Buffer is made by the Data Conversion Scheduler based on the amount of metadata stored in an Active Segment and the segment's total size. Both of these parameters are user-configurable. The Active Segments are first converted to Temporary Static Segments in parallel and then merged to form Static Segments. The data points in the Static Segment are then sorted by key and compressed using Gorilla's compression method to create Compressed Segments. By compressing larger segments, the cost is amortized. The compression process is also controlled by the Data Conversion Scheduler based on user-configurable parameters. To efficiently retrieve data points based on metadata, a novel Compressed Inverted Index is stored as part of the Static and Compressed Segments. The index consists of a trie and two arrays compressed using p4nzenc64. During query processing, ByteSeries first prunes by time and then by metadata using the Compressed Inverted Index. The system does not support approximation.

BitemporalDB is a centralized TSMS for financial time series designed by **[Sedighi et al. 2020]**. To ensure all changes can be audited, it purposely does not support deleting or updating data points in place. Instead, a new version is created with a timestamp that specifies when the update was performed, thus allowing past versions of a database to be queried. For this, BitemporalDB uses two types of timestamps: As-Of timestamps specify when the data was written to the database, and As-At timestamps specify when a value was updated. The system stores the data in Azure Cosmos DB using three tables. The Value table stores the time series with each column containing a data point with an As-Of timestamp. Additionally, a column may contain one or more data points with an As-At timestamp if updates have been performed. By storing updates as new data points, the previous state of the database can be queried. The

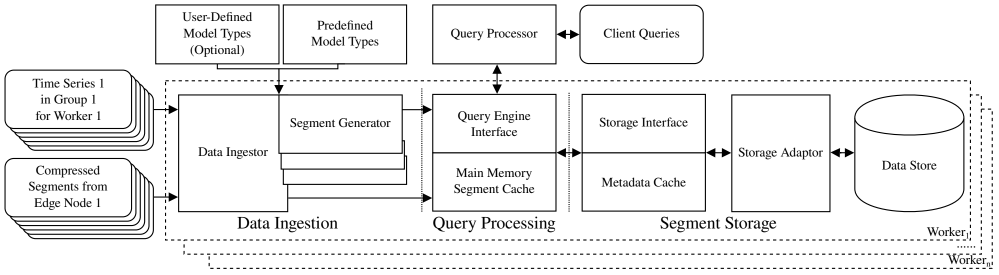


Figure 1.26: The architecture of ModelarDB's cloud worker nodes, redrawn from [Jensen et al. 2023]. The architecture of a single cloud worker node is used when it is deployed on the edge.

Log table stores all inserted data points without any updates and is used to improve query performance. The Index table is used to look up specific time series based on their metadata. It also contains Markers which are references to specific timestamps. Markers are used to efficiently retrieve a specific data point or a range of data points. They can be automatically created by the system or manually by the user. BitemporalDB supports multiple specialized querying methods. Latest Search returns the most recent data point with an As-Of timestamp and any data points with an As-At timestamp associated with it. Marker Search returns data points associated with a specific Marker. Last, Marker to Marker Search returns the range of data points between two Markers. Users can also provide hints as part of a query. These hints are an expanded time interval that BitemporalDB should also include when looking up data points if the initial query produces an empty result. The system does not support approximation.

ModelarDB is a distributed modular open-source TSMS proposed by **[Jensen et al. 2018, 2019, 2021, 2023]** that splits regular time series into variable-size segments and compresses them as models with metadata. The system uses models for both lossy and lossless compression. Thus, it supports approximation through lossy compression. Aggregate queries can also be efficiently answered from the metadata and models. ModelarDB consists of the portable Java library ModelarDB Core which can be interfaced with different data processing engines and data stores depending on the use case. The current implementation of ModelarDB is designed to be deployed on both edge nodes and a cluster of cloud nodes using the same binary. On the edge, it uses H2 as its data processing engine and either an RDBMS or a novel file-based data store backed by the local file system as its data store. In the cloud, it uses Apache Spark as its data processing engine and either Apache Cassandra or the file-based data store backed by HDFS as its data store. None of the existing systems were modified to interface them with ModelarDB, thus existing infrastructure can be reused. ModelarDB supports continuously transferring segments between instances using Apache Arrow Flight. The architecture of ModelarDB's cloud worker nodes is shown in Figure 1.26. ModelarDB uses the same architecture as a single cloud worker node when deployed as a single node system on

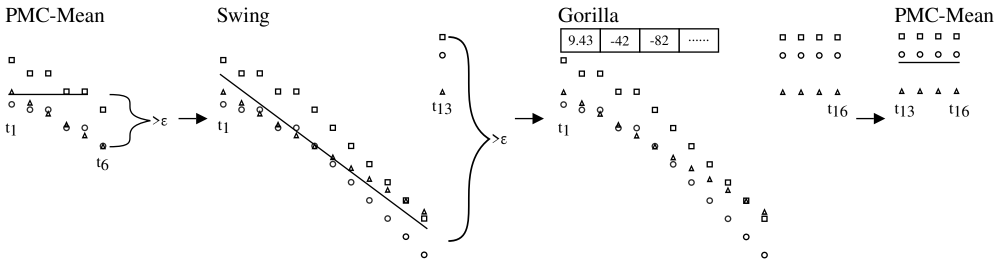


Figure 1.27: ModelarDB's compression method, redrawn from [Jensen et al. 2023]

an edge node. The Data Ingestion components can pull data points from a variety of sources and fit models to these data points using model types. Alternatively, segments from another instance of ModelarDB can be pushed to them. A set of included and, optionally, user-defined model types are used for compressing each time series as their structure generally changes over time. ModelarDB includes PMC-Mean [Lazaridis and Mehrotra 2003], Swing [Elmeleegy et al. 2009], and Gorilla's compression method. Groups of similar time series can be compressed together to further reduce the amount of storage required. These groups can either be created manually by the users or automatically by ModelarDB using the time series metadata. To compress data points from a group of time series within a user-defined error bound, ModelarDB uses a window-based approach as shown in Figure 1.27. The data points are first added to a buffer and a model is then fitted to the data points using the first model type the system is configured to use, which is the constant model type PMC-Mean in Figure 1.27. When data points are received that this model type can no longer fit a model to within the error bound, t_6 in Figure 1.27, the system switches to the next model type and initializes it using the buffer, which is the linear model type Swing in Figure 1.27. For lossless model types like Gorilla's compression method, a length bound is used instead of an error bound. This continues until all model types have been evaluated. A segment containing metadata and the model that provides the best compression ratio is then emitted, a model of type Swing in Figure 1.27. Afterward, the corresponding data points are deleted, and the fitting process restarted by initializing the first model type with the remaining data points in the buffer. The Query Processing components store recently constructed and queried segments in an in-memory cache and use one of the data processing engines to execute SQL queries on reconstructed data points or directly on the segments if applicable. The Segment Storage components cache metadata and mappings between different types of metadata. They also interface ModelarDB with one of the data stores. An optimized storage layout is used for each data store, but they generally store segments so retrieval of segments for specific time series and time intervals is efficient. ModelarDB also supports storing time series that can be derived from another time series as a single dynamically compiled user-defined function, e.g., $\cos(value \times \pi/180)$, instead of the time series segments.

Table 1.4: High-level overview of the surveyed systems with an external data store

	Year	Primary Purpose	Deployment	Maturity	Approximation	Latency
TSDS☆	2010	Data Analytics	Centralized	Demonstration	Approximate Query Processing	Batch
SensorGrid	2013	Data Analytics	Distributed	Demonstration	Approximate Query Processing	Real-Time
Respawn	2013	Monitoring	Distributed	Demonstration	Not Supported	Near Real-Time
Bolt☆	2014	Monitoring	Centralized	Demonstration	Approximate Query Processing	Near Real-Time
Druid☆	2014	Data Analytics	Distributed	Mature	Approximate Query Processing	Near Real-Time
Unnamed	2014	Evaluation	Distributed	Proof-of-Concept	Lossy Compression	Batch
Unnamed	2014	Monitoring	Distributed	Mature	Not Supported	Real-Time
Unnamed	2015	Data Analytics	Distributed	Mature	Approximate Query Processing	Near Real-Time
servIoTicy☆	2015	Monitoring	Distributed	Demonstration	Not Supported	Real-Time
Gorilla☆	2015	Monitoring	Distributed	Mature	Not Supported	Near Real-Time
Storacle	2015	Monitoring	Centralized	Demonstration	Not Supported	Near Real-Time
BTrDB☆	2016	Monitoring	Distributed	Mature	Not Supported	Near Real-Time
FluteDB	2018	Monitoring	Distributed	Proof-of-Concept	Lossy Compression	Near Real-Time
M-DB	2019	Monitoring	Distributed	Demonstration	Approximate Query Processing	Near Real-Time
UPS	2020	Data Analytics	Distributed	Demonstration	Not Supported	Near Real-Time
TimeCrypt☆	2020	Data Analytics	Centralized	Demonstration	Approximate Query Processing	Batch
Peregreen	2020	Data Analytics	Distributed	Mature	Approximate Query Processing	Batch
Db2 Event Store	2020	Data Analytics	Distributed	Mature	Not Supported	Near Real-Time
Monarch	2020	Monitoring	Distributed	Mature	Not Supported	Near Real-Time
TorqueDB	2020	Monitoring	Distributed	Demonstration	Not Supported	Near Real-Time
ByteSeries	2020	Monitoring	Distributed	Mature	Not Supported	Near Real-Time
BitemporalDB	2020	Evaluation	Centralized	Proof-of-Concept	Not Supported	Near Real-Time
ModelarDB☆	2023	Data Analytics	Distributed	Demonstration	Lossy Compression	Near Real-Time

1.5.3 Discussion

A high-level overview of the surveyed TSMSs that use an external data store is shown in Table 1.4. These systems are predominately distributed with the only exceptions being TSDS [Weigel et al. 2010], Bolt [Gupta et al. 2014], Storacle [Cejka et al. 2015], Time-Crypt [Burkhalter et al. 2020], and BitemporalDB [Sedighi et al. 2020]. Instead, centralized TSMSs are generally implemented using an internal data store and as DBMS extensions as shown in Section 1.4 and Section 1.6, respectively. Surprisingly, only TSDS [Weigel et al. 2010] implements an entirely new proprietary data store. The other systems instead reuse existing data stores such as Apache Cassandra, Apache HBase, and Couchbase to varying degrees. However, thirteen of the twenty-three systems implement an entirely new proprietary data processing engine. So while existing systems are reused as components to a high degree, the development of new data processing engines has been a higher priority than new data stores. One reason could be that TSMSs that use an external data store are primarily designed in this manner so an existing external data store can be reused. Although existing systems are often reused as components, three of the TSMSs have simple proof-of-concept implementations: the system by [Guo et al. 2013, 2014a,b], FluteDB [Li et al. 2017, 2018], and BitemporalDB [Sedighi et al. 2020]. Also, two systems exist only to evaluate new methods: the system by [Guo et al. 2013, 2014a,b] and BitemporalDB [Sedighi et al. 2020]. Eleven systems are classified as demonstration in terms of maturity. For example, Respawn [Buevich et al. 2013], servIoTicy [Pérez and Carrera 2015], and ModelarDB [Jensen et al. 2018, 2019, 2021, 2023]. Despite the significant reuse of existing systems as components, only nine of the twenty-three systems have robust mature implementations. For example, Druid [Yang et al. 2014], Gorilla [Pelkonen et al. 2015], and ByteSeries [Shi et al. 2020]. Approximation is supported by eleven of the twenty-three systems. Of these, eight support AQP while only three support lossy compression. Interestingly approximation is even supported by three of the nine mature systems. Specifically, Druid [Yang et al. 2014], the system by [Mickulicz et al. 2015], and Peregreen [Visheratin et al. 2020] support AQP. So AQP is, in general, the preferred approach for approximation. Real-time data processing is only supported by three of the twenty-three systems. SensorGrid implements support for continuous queries [Cuzzocrea and Saccà 2013], the system by [Williams et al. 2014] uses a proprietary stream processing platform as a component of the TSMS, while servIoTicy [Pérez and Carrera 2015] uses Apache Storm as a component of the TSMS. In summary, most of the TSMSs that use an external data store are distributed. Almost all of the systems use an existing data store as a component, while over half implement an entirely new proprietary data processing engine. Despite the significant reuse of existing systems as components, a few of the TSMSs have simple proof-of-concept implementations. However, the majority of the TSMSs with an external data store have implementations that are robust enough to evaluate new methods and architectures with real-life use cases, and a significant number of them are also ready to be or are already used in production. Advanced functionality like approximation and real-time data processing is implemented by over half of the systems. However, only four of these systems have mature implementations.

1.6 DBMS Extensions

1.6.1 Overview

Implementing a TSMS as DBMS extensions allows the DBMS's existing data processing engine and data store to be reused. This can reduce development time, and existing knowledge about how to efficiently use the DBMS can be reused. Also, the data processing engine can retrieve data directly from the data store, however, they generally cannot be scaled separately. Unless the DBMS is modified and not just extended, the TSMS must also be designed and implemented within the constraints provided by the DBMS's extension APIs, data model, and storage layout, possibly limiting the optimizations that can be performed. The DBMS may also have features that are not needed by a TSMS but adds overhead, e.g., support for transactions.

1.6.2 Systems

[Fischer et al. 2012b] proposed extending PostgreSQL to support model-based forecasting of time series in the form of the TSMS F^2DB. This centralized system is designed to provide forecasting for use in data warehouse-style business intelligence without needing to export the time series to other tools such as Python or R. Its architecture is shown in Figure 1.28. F^2DB supports creating models using ARIMA or exponential smoothing. However, it also provides a generic interface for implementing different forecasting methods. Thus, the system can be extended if the included model types are insufficient. Users must manually fit models to time series using a `CREATE MODEL` SQL statement. The models are then stored in the Model Pool and automatically indexed and maintained by F^2DB [Fischer et al. 2010]. To reduce the number of models to maintain, the system can answer forecast queries using models from other levels of the hierarchy than the one queried. To do so, users must specify the functional dependencies using `CREATE FORECAST HIERARCHY` and, optionally, how forecasted data points can be disaggregated using `CREATE DISAG SCHEME`. Based on the current set of models, their past accuracy, and a query workload, an alternative configuration of models can be computed using a Model Advisor [Fischer et al. 2011, 2013]. The advisor uses multiple heuristics to reduce the number of models it evaluates. Each heuristic is focused on one time series or the relationship between multiple time series. The choice of one of multiple models is determined based on a trade-off between their accuracy and performance. The final model configuration can be loaded into F^2DB. The system does not support approximation.

[Khalefa et al. 2012] implemented *TimeTravel* by extending PostgreSQL with support for model-based AQP and forecasting. Thus, it is centralized and provides a uniform SQL interface for exact queries on historical data points, approximate queries on historical data points, and forecasting. TimeTravel stores ingested data points in arrays ordered by time and creates models organized in a novel Hierarchical Model Index on top of the time series. The architecture of TimeTravel can be seen in Figure 1.29 and consists of Offline components for building and compressing the index, and Online components for query processing and

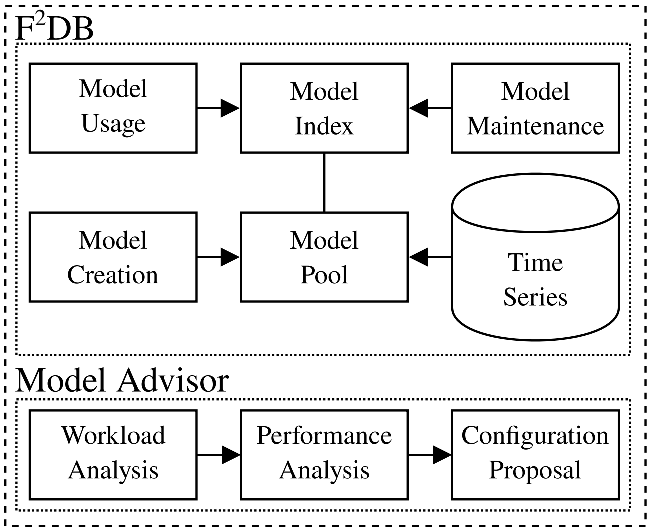


Figure 1.28: The architecture of F^{2}DB, redrawn from [Fischer et al. 2012b]

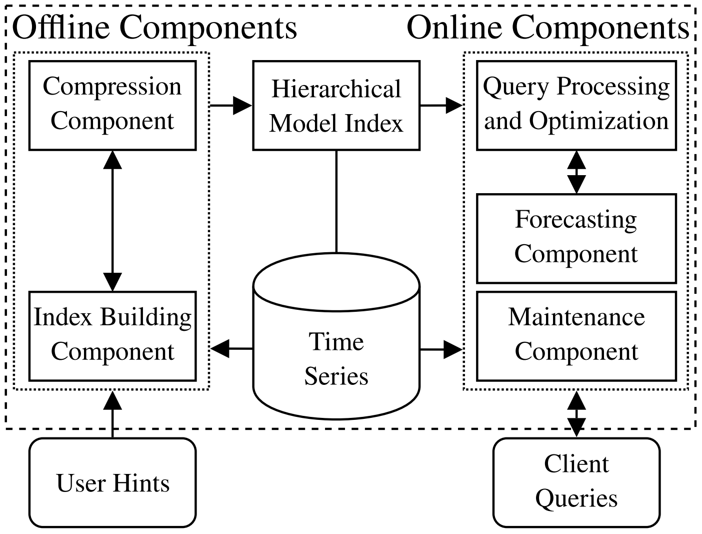


Figure 1.29: The architecture of TimeTravel, redrawn from [Khalefa et al. 2012]

maintenance of the index. The Hierarchical Model Index consists of multiple levels of models, with each level providing more detailed models. This hierarchy enables the system to efficiently process queries using models with an appropriate level of detail based on the required accuracy of the result. TimeTravel stores statistics about the index in the system's catalog for query planning. To build a Hierarchical Model Index for a time series, the user must specify the time series' seasonality, the required error bounds, and the forecasting method to use. The system then builds the index by first creating a model over the entire time series. Then it recursively splits the time series into disconnected segments and fits models to them according to the specified error bounds. The system automatically maintains the models in the index when new data points are ingested if they exceed the error bounds. To use the index for query processing, TimeTravel extends PostgreSQL's query optimizer with support for estimating the cost of using the index, e.g., for pruning, compared to only executing the query using the time series. As part of the MIRABEL smart grid project, the methods created for TimeTravel were also incorporated into an Electricity Data Management System (EDMS) [Fischer et al. 2012a].

[Huang et al. 2014] implemented a distributed TSMS as extensions to IBM Informix that provides a uniform SQL interface for querying multivariate time series and relational data together. The system primarily consists of three components and a set of data stores as shown in Figure 1.30. The Configuration component manages information about the data sources the system is ingesting from and the data stores available to the system. The Storage component ingests data points, organizes them into an appropriate storage format, and compresses them. The storage format and compression method used depend on the time series characteristics, e.g., how often data points are collected and if it is done at a regular or irregular time interval. The three supported storage formats are shown in Figure 1.31. For all formats, B-Trees are used to index the timestamp and id field. The first two formats store the data points of a single time series in a blob. For regular time series only the values are stored, while both the timestamps and values are stored for irregular time series. The last format stores multiple time series together by replacing the device id with a group id and storing the device id, timestamp, and value of each data point in the blob. The Mixed Grouping Data Structure is generally used for regular and irregular time series with a low sampling interval. For compression, the system stores stable values as linear functions while quantization is used for highly fluctuating values. Both methods support lossless compression and lossy compression within an error bound. Thus, the TSMS supports approximation through lossy compression. The Query component is implemented using IBM Informix's Virtual Table Interface and exposes the time series as virtual tables, thus allowing both relational data and time series to be queried together using standard SQL. IBM Informix's cost model is also extended to support the new storage formats.

TRISTAN is a centralized TSMS designed by **[Marascu et al. 2014]** for low-latency analysis of noisy irregular time series through the use of online dictionary compression and AQP. The system is based on the MiSTRAL architecture [Marascu et al. 2013] and is implemented as extensions to the open-source in-memory DBMS HYRISE. This architecture is shown in

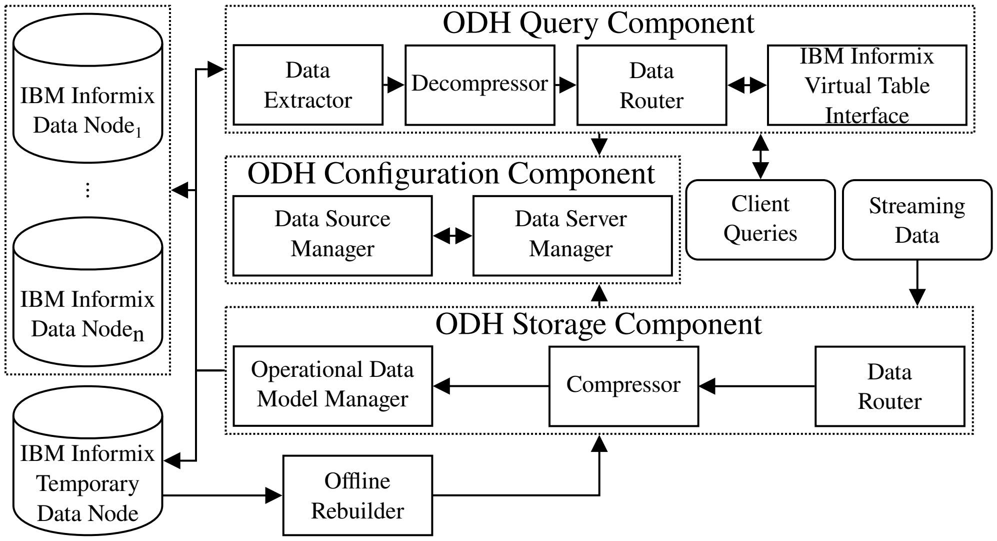


Figure 1.30: The architecture of the TSMS implemented as extensions to IBM Informix by [Huang et al. 2014]. The figure was redrawn from [Huang et al. 2014]

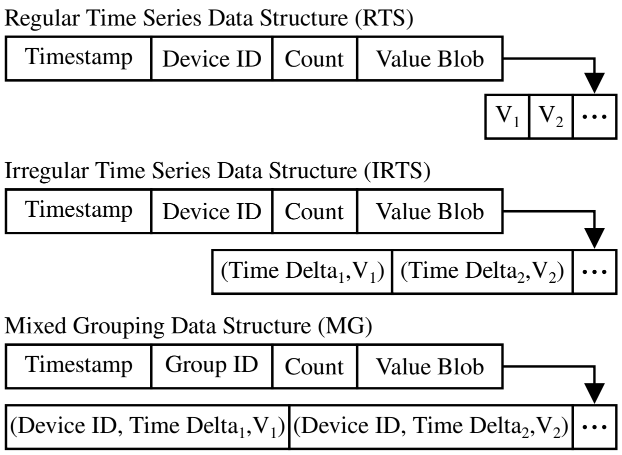

Regular Time Series Data Structure (RTS)
Timestamp
Device ID
Count
Value Blob
V_1 V_2 ⋯
Irregular Time Series Data Structure (IRTS)
Timestamp
Device ID
Count
Value Blob
(Time Delta$_1$,V_1) (Time Delta$_2$,V_2) ⋯
Mixed Grouping Data Structure (MG)
Timestamp
Group ID
Count
Value Blob
(Device ID, Time Delta$_1$,V_1) (Device ID, Time Delta$_2$,V_2) ⋯

Figure 1.31: The data structures used for storing time series in the TSMS by [Huang et al. 2014]. The figure was redrawn from [Huang et al. 2014]

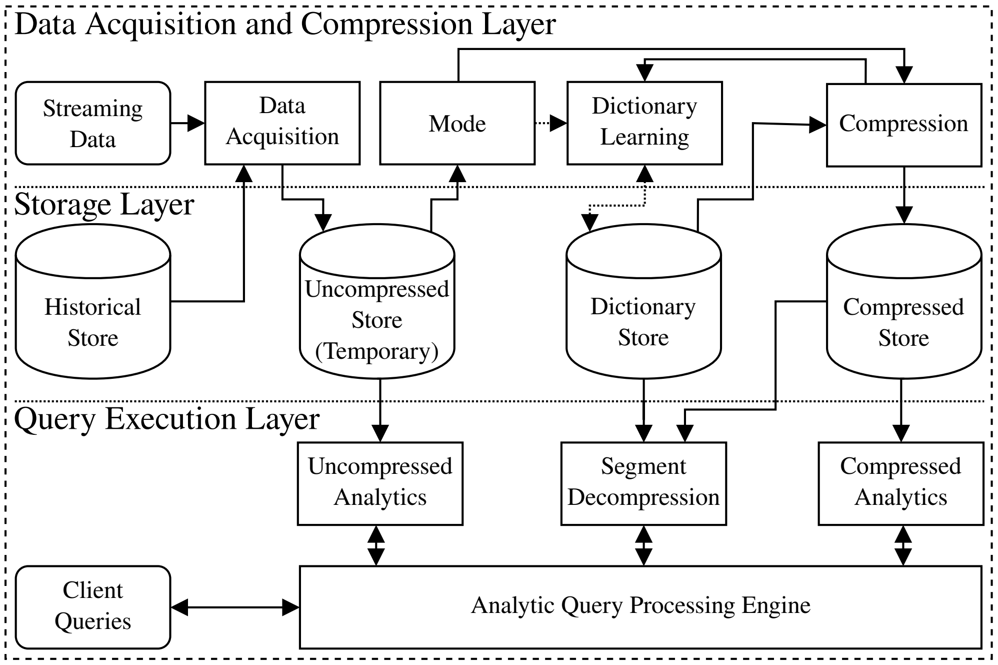


Figure 1.32: The MiSTRAL architecture realized by TRISTAN. Data flow that mainly occurs offline is shown as dotted lines. The figure was redrawn from [Marascu et al. 2013]

Figure 1.32 and consists of three layers: the Data Acquisition and Compression Layer, the Storage Layer, and the Query Execution Layer. The Data Acquisition and Compression Layer ingests time series and splits them into fixed-size segments which are temporarily stored in the Uncompressed Store managed by the Storage Layer. The time series can be ingested from a stream or be batch-loaded from the Historical Store managed by the Storage Layer. Then the segments are compressed as a sequence of weighted smaller fixed-size time series using a pre-trained dictionary provided as a parameter to the system. This dictionary is created offline; however, the system can adjust it at run-time if patterns emerge that cannot be efficiently represented using the current dictionary. The Query Execution Layer executes queries received through an undefined interface. Queries on recent data that have yet to be compressed are executed against the Uncompressed Store. Queries on data stored as compressed segments in the Compressed Store may be executed directly on the compressed representation or on data points reconstructed from the compressed segments. TRISTAN supports approximation through AQP as the number of weighted smaller fixed-size time series used from the compressed representation can be adjusted at query-time. The CORelation-Aware compression of time series streams based on sparse Dictionary coding (CORAD) extends TRISTAN's compression method by taking correlation into account [Khelifati et al. 2019]. When compressing a set of time series, CORAD normalizes the time series, splits them into segments, and computes their inter-segment correlation. Segments that are not correlated with a stored segment are compressed as a sequence of weighted fixed-size time series from a dictionary like in TRISTAN. If a segment is correlated with a stored segment, it is stored as a reference to that segment and a scaling factor. Thus, multiple segments might be read by a query to retrieve a single segment.

[Bakkalian et al. 2016] extended Oracle Database with support for storing and querying time series with metadata as models instead of data points. The centralized system stores time series using two different tables. One table stores the data points while the other table stores models that represent the intervals between consecutive data points. The current implementation only supports using linear functions as models. The use of linear functions allows the system to interpolate values between the data points. Thus, it supports approximation through AQP. Support for forecasting is also listed as a benefit of storing time series as models. However, forecasting is currently not supported by the system. Queries are executed using data points reconstructed from the models. The system builds directly on the following two papers. The first paper [Bebel et al. 2012] proposed a data model for OLAP analysis of sequential data that formalizes Events as n-tuples and Sequences as ordered collections of Events. Building upon this data model, the second paper [Koncilia et al. 2014] formalized the notion of Intervals as the gap between consecutive Events and defines Sequences as ordered collections of Intervals.

Chronix is a distributed open-source TSMS proposed by **[Lautenschlager et al. 2015, 2017]** for analyzing time series with metadata collected by monitoring distributed systems. Thus, it has comprehensive support for different types of operational data such as metrics, traces, and logs. It is implemented by extending Apache Solr and optimized for few large writes and many reads. When ingesting, the system first, optionally, creates a derived representation from the time series that is optimized for the intended queries. If all of the intended queries can be efficiently answered using this representation, the ingested time series may optionally be deleted. The time series or the derived representation is then split into fixed-size segments. These are then compressed and stored together with user-defined and system metadata such as the data format, ids, and the time interval the segment contains data for. For compressing the timestamps, Chronix uses the novel Date-Delta-Compaction (DDC) lossy compression method. It computes the delta-of-deltas between consecutive timestamps and discards the delta if the delta-of-delta is below a threshold. Also, if the cumulative difference becomes too large, a delta that corrects the difference is stored. As a result, Chronix supports approximation through lossy compression. After DDC the segment is compressed by a lossless compression method like gzip, LZ4, or XZ. The metadata is used for retrieving specific segments and a user-configurable subset is indexed by Apache Solr. Chronix executes queries by requesting the relevant segments from Apache Solr and then applying functions to the query result. The system is also extensible, e.g., administrators can add new data analysis methods or query-optimized representations.

[Yang et al. 2019] proposed the TSMS *EdgeDB* by extending BTrDB with support for managing correlated time series in groups to improve both ingestion speed and query performance. Specifically, the system is designed to run centralized on powerful edge nodes and execute queries on groups instead of on individual time series. The groups are created by the user based on metadata or for specific time series. The groups can also be changed dynamically at run-time. EdgeDB consists of a Data Merging component, an Indexing component, and a Data Storing component. Ingested data points are first split into segments based on a predefined time interval. Then the Merging component reorganizes the segments into tablets according to the groups. Each tablet also includes a header with the location of the segments in the tablet together with lightweight statistics about them. The size of the segments can also optionally be reduced using lossless or lossy compression. Thus, EdgeDB supports approximation through lossy compression. The Indexing component then indexes the tablets using a novel Time Partitioned Elastic Index. This index extends the k-ary tree proposed for BTrDB by storing tablets in the leaf nodes and creating multiple trees for different time intervals and resolutions. For example, seven trees that index the tablets created each day for the latest week, and five trees that index the tablets created each week for the latest month except those created in the latest week. As EdgeDB assumes that the data points are received in order, it only keeps the latest tree with the highest resolution in memory. The trees with the second-highest resolution are created from these, while the trees with the third-highest resolution are created from the trees with the second-highest one, and so on. During query processing, the location of the nodes on disk can be computed as the trees' structures are stored in memory. The Storing component merges tablets from different groups together using a novel Time Merged Tree. Its internal nodes contain a time interval and references to its child nodes. The leaf nodes contain a time interval, references to tablets storing segments for that time interval, and a bitmap indicating which time series the referenced tablets store data for. Leaf nodes are flushed when tablets have been inserted for all time series or a timeout occurs, e.g., if a sensor has failed. By batching writes, EdgeDB trades durability for faster ingestion speed based on the assumption that there is a high degree of redundancy in time series and that some sensors can retransmit their data.

[Agarwal et al. 2020] proposed extending PostgreSQL with support for multivariate time series prediction in the form of the open-source TSMS *tspDB*. This centralized system supports imputation and forecasting using a novel incremental prediction method based on matrix factorization. A prediction model is created using `CREATE PREDICTION_MODEL` with a table, the timestamp column, and the value columns as the arguments. By default tspDB automatically maintains the models when new data points are ingested. The system stores both data points and model parameters in tables. For each model, the parameters associated with the singular value decomposition and the linear regression coefficients are stored. Parameters derived from these are pre-computed for the latest models using a materialized view for query performance. Prediction is done using a `PREDICT` query which functions like `SELECT` but

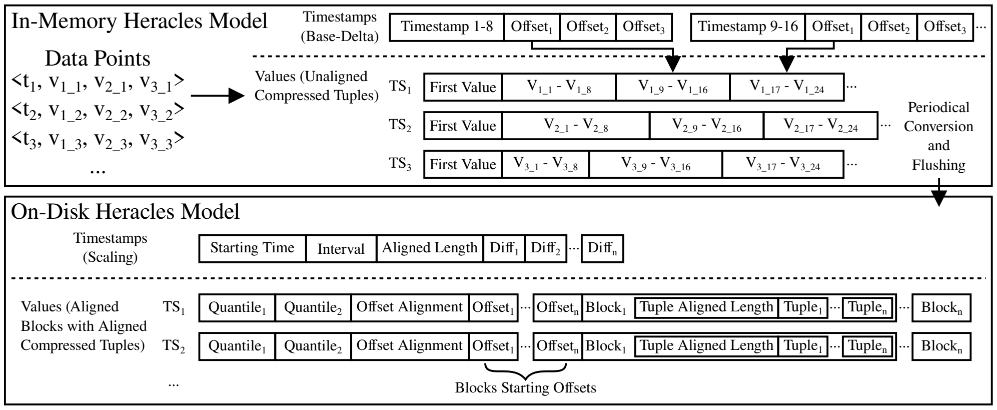


Figure 1.33: The two storage models used by Heracles, redrawn from [Wang et al. 2021]

performs imputation or forecasting depending on the queried time interval. Thus, the system supports AQP. The query results include both the estimated mean and the prediction interval.

Heracles is implemented as extensions to the TSMS Prometheus and is proposed by **[Wang et al. 2021]**. It is centralized, open-source, and optimized for multivariate time series with many values as these are often collected when monitoring entities. Specifically, Heracles splits multivariate time series into fixed-size segments and stores the timestamps and values separately as Tuples. Each Tuple contains a fixed number of timestamps or values as shown in Figure 1.33. In memory, timestamps are compressed using delta compression, while the values are compressed using an XOR-based compression method similar to the one used by Gorilla. As the timestamps and values are compressed online, Heracles checks if the encoding used by each value Tuple can be improved once it has been filled. Mapping from timestamps to values is done using offsets. When Tuples are flushed to disk, Heracles changes the compression methods used and how it maps from timestamps to values. A Tuple of timestamps TS is compressed as $get(TS,i) - (first(TS) + i \times interval)$ where i is the index of the timestamp being compressed and $interval = (last(TS) - first(TS))/length(TS)$. Values are compressed using the same XOR-based compression method, but the initial value is selected based on the values' quantiles. Offsets are not stored for the on-disk format, instead, padding is used so the value Tuples have a known alignment. Heracles does not immediately delete flushed Tuples from memory as they can be used to answer queries. Instead, an epoch-based memory manager is used that gradually reuses older memory blocks. It ensures that the blocks currently in use are not reused by assigning the current epoch to all queries when they start executing and then only reusing blocks from epochs earlier than the currently executing queries. Heracles queries consist of metadata specifying which time series to query and a time range. A query is first translated to group ids, time series ids, and a time range. Then, Heracles retrieves the relevant

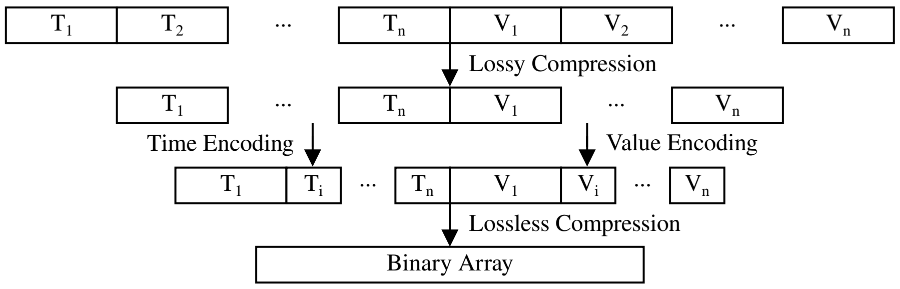

T_1 T_2 ··· T_n V_1 V_2 ··· V_n
Lossy Compression
T_1 ··· T_n V_1 ··· V_n
Time Encoding
Value Encoding
T_1 T_i ··· T_n V_1 V_i ··· V_n
Lossless Compression
Binary Array

Figure 1.34: The compression method used by TVStore, redrawn from [An et al. 2022]

data points from memory before retrieving the relevant data points from disk. Binary search is used in both cases to prune based on the time range. Heracles does not support approximation.

[An et al. 2022] implemented the distributed open-source TSMS *TVStore* by extending Apache IoTDB. TVStore supports limiting the amount of storage used by the system to a user-defined upper bound while storing important segments with little or no error. Instead of immediately deleting the oldest segments to reduce the amount of storage required, TVStore uses the novel Time-Varying Compression (TVC) framework to store segments with different amounts of error. Thus, TVStore supports approximation through lossy compression. TVC determines the compression ratio required for each segment using Time-Dependent Functions. These are non-decreasing functions that map from the age of a segment to a compression ratio. The exponential, power-law, and constant functions are included. TVC uses power-law by default. Thus, the system is generally designed for analytics in domains where the most recent data points are the most important. TVC only supports fixed-ratio compressors, i.e., compression methods that target a specific compression ratio. In addition, the compression methods must support recompression of segments without decompression, decompression that does not change if a segment has been compressed multiple times, and computation of error bounds when recompressing. TVC uses Piecewise Linear Approximation by default. However, both user-defined Time-Dependent Functions and compression methods can be added by implementing their respective Java interfaces. TVC also stores the compression ratio and error for each compressed segment and supports dropping segments based on a user-defined compression ratio or error threshold. While ingesting, TVStore automatically determines when to compress segments so the user-defined storage bound is not exceeded and the amount of error is not increased unnecessarily. TVStore also allows users to specify an amount of recent data points that should be stored without any error. The segments are compressed in batches to reduce the number of times each segment is recompressed. To determine how much and when compression is required to not exceed the storage bound, TVStore continuously monitors the amount of data stored, the disk's average read throughput, the disk's average write

Table 1.5: High-level overview of the surveyed systems implemented as DBMS extensions

	Year	**Primary Purpose**	**Deployment**	**Maturity**	**Approximation**	**Latency**
F^2DB	2012	Data Analytics	Centralized	Demonstration	Not Supported	Near Real-Time
TimeTravel	2012	Data Analytics	Centralized	Demonstration	Approximate Query Processing	Near Real-Time
Unnamed	2014	Monitoring	Distributed	Mature	Lossy Compression	Near Real-Time
TRISTAN	2014	Data Analytics	Centralized	Demonstration	Approximate Query Processing	Near Real-Time
Unnamed	2016	Evaluation	Centralized	Proof-of-Concept	Approximate Query Processing	Batch
Chronix☆	2017	Data Analytics	Distributed	Demonstration	Lossy Compression	Batch
EdgeDB	2019	Monitoring	Centralized	Demonstration	Lossy Compression	Near Real-Time
tspDB☆	2020	Data Analytics	Centralized	Demonstration	Approximate Query Processing	Near Real-Time
Heracles☆	2021	Monitoring	Centralized	Demonstration	Not Supported	Near Real-Time
TVStore☆	2022	Data Analytics	Distributed	Demonstration	Lossy Compression	Near Real-Time

throughput, and the ingestion throughput. Using this information, compression is started at a storage threshold which guarantees that compression can be performed without exceeding the storage bound. After the lossy compression, Apache IoTDB's lossless compression methods are used as shown in Figure 1.34. Like Apache IoTDB, TVStore stores statistics and metadata about the compressed data. Decompression is performed before the data is returned to the data processing engine, so TVStore provides the exact same query functionality as Apache IoTDB.

1.6.3 Discussion

A high-level overview of the surveyed TSMSs implemented as extensions to existing DBMSs is shown in Table 1.5. These systems are predominately centralized with the only exceptions being: the system by [Huang et al. 2014], Chronix [Lautenschlager et al. 2015, 2017], and TVStore [An et al. 2022]. Instead, distributed TSMSs generally use an external data store as shown in Section 1.5. Five of the ten systems extend an RDBMS, with three extending PostgreSQL [Agarwal et al. 2020, Fischer et al. 2012b, Khalefa et al. 2012], one extending IBM Informix [Huang et al. 2014], and one extending Oracle Database [Bakkalian et al. 2016]. Surprisingly, only three of the latest systems extend an existing TSMS. Specifically, EdgeDB [Yang et al. 2019] extends BTrDB, Heracles [Wang et al. 2021] extends Prometheus, and TVStore [An et al. 2022] extends Apache IoTDB. The implementations of most of the TSMSs are classified as demonstration. Specifically, only the system by [Bakkalian et al. 2016] is classified as a proof-of-concept system, eight of the ten TSMSs are classified as demonstration systems, and only the system by [Huang et al. 2014] is classified as a mature system. Many of the implemented extensions add support for advanced query functionality such as forecasting [Agarwal et al. 2020, Fischer et al. 2012b, Khalefa et al. 2012], interpolation [Bakkalian et al. 2016, Marascu et al. 2014], and anomaly detection [Lautenschlager et al. 2015, 2017].

However, the system by [Huang et al. 2014] also supports multiple different storage formats, Chronix [Lautenschlager et al. 2015, 2017] can store time series using query-optimized representations in addition to compressed data points, EdgeDB [Yang et al. 2019] manages time series in groups to improve read and write performance, Heracles' [Wang et al. 2021] data store is optimized for multivariate time series, and TVStore [An et al. 2022] limits the amount of storage used to a user-defined upper bound. Approximation is supported by eight of the ten systems, with four systems supporting AQP [Agarwal et al. 2020, Bakkalian et al. 2016, Khalefa et al. 2012, Marascu et al. 2014] and four systems supporting lossy compression [An et al. 2022, Huang et al. 2014, Lautenschlager et al. 2015, 2017, Yang et al. 2019]. None of the systems implemented as DBMS extensions support real-time data processing. In summary, the TSMSs that are implemented as extensions to DBMSs are generally: not mature, centralized, implemented as extensions to RDBMSs, add advanced query functionality, and almost always support approximation. However, none of them support real-time data processing.

1.7 Future Work

1.7.1 Support for Data Analytics

The majority of the surveyed TSMSs only provide a few built-in methods for data analytics, if any at all. For example, only NilmDB [Paris et al. 2014], RINSE [Zoumpatianos et al. 2015a], MetricQ [Ilsche et al. 2019], TubeDB [Wöllauer et al. 2021], and SensorGrid [Cuzzocrea and Saccà 2013] support visualization. So the time series must be exported to external applications like Python or R before they can be analyzed, thus adding an unnecessary performance overhead and additional complexity for the user. By integrating support for data analytics directly into the TSMSs the need for exporting the data can be avoided completely. Information about which analyses will be performed may also enable the creation of sophisticated dynamic optimizers that can optimize the entire data analytics pipeline instead of only data retrieval [Bagnall et al. 2019, Palpanas 2015, 2016a,b, Zoumpatianos and Palpanas 2018]. Another limitation of the current TSMSs for data analytics is the lack of extensibility which limits users to the few methods included with the systems. Of the surveyed systems only NilmDB [Paris et al. 2014], TubeDB [Wöllauer et al. 2021], M-DB [Arora et al. 2019] and Chronix [Lautenschlager et al. 2015, 2017] support adding user-defined methods for data analytics. Similarly, only SensorGrid [Cuzzocrea and Saccà 2013], the system by [Williams et al. 2014], and servIoTicy [Pérez and Carrera 2015] support real-time query processing using user-defined continuous queries. So to fully benefit from integrating data analytics with TSMSs, the systems must be very simple for users to extend and they must treat user-defined methods like native methods instead of black boxes [Sichert and Neumann 2022].

1.7.2 Integration of Edge and Cloud

Time series collected from sensors are often ingested on the edge and then transferred to the cloud for analysis. However, most of the surveyed systems are only optimized for one type of deployment. Thus, users generally have to use multiple TSMSs and manually transfer the data points safely, e.g., ensuring that lost data points are retransmitted. Of the surveyed systems, only Apache IoTDB [Wang et al. 2020], Respawn [Buevich et al. 2013], Storacle [Cejka et al. 2015], and ModelarDB [Jensen et al. 2018, 2019, 2021, 2023] support transferring data points from the edge to the cloud, while VergeDB [Paparrizos et al. 2021] optimizes the representation used for the time series on the edge based on the analytics that will be performed in the cloud. However, Apache IoTDB, ModelarDB, Respawn, and Storacle provide limited functionality for data analytics while VergeDB does not implement any support for transferring the ingested data points to the cloud. For a TSMS to manage time series across both the edge and the cloud, it must provide ingestion and compression on the edge, transfer of compressed data points from the edge to the cloud, and query processing across both the edge and the cloud. Deploying an integrated TSMS across the edge and the cloud also enables the system to be resource and workload aware. For example, the system may use different strategies for deciding which data points to transfer to the cloud based on the available bandwidth and the queries being executed in the cloud. Similarly, the sampling strategy and compression methods used on the edge could also be dynamically changed to minimize latency or error [Chiariotti et al. 2022, Holm et al. 2021, Hülsmann et al. 2020, 2021, Paparrizos et al. 2021, Traub et al. 2017]. For example, a lossy compression method with little or no impact on the query results becomes functionally lossless [An et al. 2022, Lautenschlager et al. 2015, 2017]. However, the impact of lossy compression is not well understood and should be investigated [Cappello et al. 2020].

1.7.3 Standardization and Portability

There is very little standardization across the surveyed TSMSs despite all of them being designed for managing time series. Specifically, there are currently no standard query interfaces, query languages, data models, or benchmarks for TSMSs. The lack of a standard interface significantly reduces interoperability between the systems. Multiple systems expose time series as relations and use SQL for queries, e.g., Db2 Event Store [Garcia-Arellano et al. 2020], ModelarDB [Jensen et al. 2018, 2019, 2021, 2023], and the system based on IBM Informix by [Huang et al. 2014]. While this allows the systems to interface with existing tooling and makes them accessible to users familiar with RDBMSs, the relational data model and SQL are arguably not effective for time series [Palpanas 2015, 2016a,b, Solleza et al. 2022, Zoumpatianos and Palpanas 2018]. Standardization of benchmarking is also an open problem. Multiple benchmarks for TSMSs have been proposed, e.g., IoT-X [Huang et al. 2014], the benchmark by [Zoumpatianos et al. 2015b], and TS-Benchmark [Hao et al. 2021]. However, the surveyed TSMSs are generally evaluated using system-specific benchmarks instead of a set of standard benchmarks like TPC, thus making comparing the TSMSs unnecessarily complex. Of course, how to design benchmarks that use both synthetic data sets and query workloads while still being representative of real-life use cases is in itself an open problem [Kraska 2021].

1.8 Conclusion

The enormous amount of time series being collected has created a need for specialized TSMSs. This chapter provided a thorough survey and classification of TSMSs that are developed through academic or industrial research and documented through peer-reviewed papers. Directions for future work have also been proposed based on the limitations of the surveyed TSMSs.

The surveyed TSMSs with an *internal data store* are predominately centralized and developed without reusing existing systems as components. Instead, most of the TSMSs implement entirely new proprietary data processing engines and data stores. Thus, a significant increase in development time is traded for the opportunity to create deeply integrated systems. Despite not reusing existing systems, the TSMSs' implementations are generally robust enough to evaluate new methods and architectures with real-life use cases, and a significant number of them are ready to be or are already used in production. However, approximation is not even supported by half of the TSMSs and none of the TSMSs support real-time data processing.

The surveyed TSMSs with an *external data store* are predominately distributed and developed by reusing existing systems as components. Although, over half implement an entirely new proprietary data processing engine. Thus, the possibility to create deeply integrated systems is traded for significantly reduced development time. Their implementations are also generally robust enough to evaluate new methods and architectures with real-life use cases, and a significant number of them are also ready to be or are already used in production. In addition, over half of the TSMSs support approximation, real-time data processing, or both.

The surveyed TSMSs implemented as *extensions to existing DBMSs* are predominately centralized. However, only one of them has a mature implementation. They mostly extend RDBMSs instead of existing TSMSs. Many of the implemented extensions add advanced query functionality such as forecasting, interpolation, and anomaly detection. In addition, almost all of the TSMSs support approximation but none of them support real-time data processing.

Based on the limitation of the surveyed TSMSs it is clear that support for data analytics, better integration of TSMSs designed for deployment on the edge and in the cloud, and development of standard interfaces and benchmarks are all relevant directions for future work.

Acknowledgments

This research was supported by the SEMIOTIC project funded by Danish Council for Independent Research grant number 8022–00284B and the MORE project funded by Horizon 2020 grant number 957345. Thanks to Abduvoris Abduvakhobov and Sigmundur Vang for providing feedback on drafts of the chapter.

Bibliography

C. Adams, L. Alonso, B. Atkin, J. Banning, S. Bhola, R. Buskens, M. Chen, X. Chen, Y. Chung, Q. Jia, N. Sakharov, G. Talbot, N. Taylor, and A. Tart. 2020. Monarch: Google's Planet-Scale In-Memory Time Series Database. *Proc. VLDB Endowment*, 13(12): 3181–3194.

A. Agarwal, A. Alomar, and D. Shah. 2020. tspDB: Time Series Predict DB. In *NeurIPS 2020 Competition and Demonstration Track*, pp. 27–56. PMLR.

C. C. Aggarwal, ed. 2013. *Managing and Mining Sensor Data*. Springer.

C. C. Aggarwal. 2015. *Data Mining: The Textbook*. Springer.

N. Agrawal and A. Vulimiri. 2017. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *Proc. 26th ACM Symp. on Operating System Principles*, pp. 647–664. ACM.

Y. An, Y. Su, Y. Zhu, and J. Wang. 2022. TVStore: Automatically Bounding Time Series Storage via Time-Varying Compression. In *Proc. 20th USENIX Conf. on File and Storage Technologies*, pp. 83–99. USENIX.

M. P. Andersen and D. E. Culler. 2016. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *Proc. 14th USENIX Conf. on File and Storage Technologies*, pp. 39–52. USENIX.

M. P. Andersen, S. Kumar, C. Brooks, A. von Meier, and D. E. Culler. 2015. DISTIL: Design and Implementation of a Scalable Synchrophasor Data Processing System. In *2015 IEEE Int. Conf. on Smart Grid Communications*, pp. 271–277. IEEE.

V. Arora, M. J. Amiri, D. Agrawal, and A. E. Abbadi. 2019. M-DB: A Continuous Data Processing and Monitoring Framework for IoT Applications. In *2019 Int. Conf. on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*, pp. 1096–1105. IEEE.

B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. 2002. Models and Issues in Data Stream Systems. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pp. 1–16. ACM.

A. Bader, O. Kopp, and F. Michael. 2017. Survey and Comparison of Open Source Time Series Databases. In *Datenbanksysteme für Business, Technologie und Web - Workshopband*, pp. 249–268. GI.

A. J. Bagnall, R. L. Cole, T. Palpanas, and K. Zoumpatianos. 2019. Data Series Management (Dagstuhl Seminar 19282). *Dagstuhl Reports*, 9(7): 24–39.

G. Bakkalian, C. Koncilia, and R. Wrembel. 2016. On Representing Interval Measures by Means of Functions. In *Proc. 6th Int. Conf. on Model and Data Engineering*, pp. 180–193. Springer.

M. Bakli, M. Sakr, and E. Zimanyi. 2019. Distributed Moving Object Data Management in MobilityDB. In *Proc. 8th ACM SIGSPATIAL Int. Workshop on Analytics for Big Geospatial Data*. ACM.

B. Bebel, M. Morzy, T. Morzy, Z. Krolikowski, and R. Wrembel. 2012. OLAP-Like Analysis of Time Point-Based Sequential Data. In *Proc. ER 2012 Workshops CMS, ECDM-NoCoDA, MoDIC, MORE-BI, RIGiM, SeCoGIS, WISM*, pp. 153–161. Springer.

M. Buevich, A. Wright, R. Sargent, and A. Rowe. 2013. Respawn: A Distributed Multi-Resolution Time-Series Datastore. In *Proc. IEEE 34th Real-Time Systems Symposium*, pp. 288–297. IEEE.

L. Burkhalter, A. Hithnawi, A. Viand, H. Shafagh, and S. Ratnasamy. 2020. TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control. In *Proc. 17th USENIX Symp. on Networked Systems Design & Implementation*, pp. 835–850. USENIX.

T. Cai, P. Liu, D. Niu, J. Shi, and L. Li. 2021. The Embedded IoT Time Series Database for Hybrid Solid-State Storage System. *Scientific Programming*, 2021.

T. Cai, Y. Ma, P. Liu, D. Niu, and L. Li. 2022. A New NVM Device Driver for IoT Time Series Database. *Micromachines*, 13(3).

W. Cao, Y. Gao, B. Lin, X. Feng, Y. Xie, X. Lou, and P. Wang. 2018. TcpRT: Instrument and Diagnostic Analysis System for Service Quality of Cloud Databases at Massive Scale in Real-time. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 615–627. ACM.

W. Cao, Y. Gao, F. Li, S. Wang, B. Lin, K. Xu, X. Feng, Y. Wang, Z. Liu, and G. Zhang. 2020. Timon: A Timestamped Event Database for Efficient Telemetry Data Processing and Analytics. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 739–753. ACM.

F. Cappello, S. Di, and A. M. Gok. 2020. Fulfilling the Promises of Lossy Compression for Scientific Applications. In *17th Smoky Mountains Computational Sciences and Engineering Conference*, pp. 99–116. Springer.

S. Cejka, R. Mosshammer, and A. Einfalt. 2015. Java embedded storage for time series and meta data in Smart Grids. In *2015 IEEE Int. Conf. on Smart Grid Communications*, pp. 434–439. IEEE.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proc. 7th USENIX Symp. on Operating System Design and Implementation*, pp. 205–218. USENIX.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Database Syst.*, 26(2).

B. Chardin, O. Pasteur, and J. Petit. 2011. An FTL-agnostic Layer to Improve Random Write on Flash Memory. In *Proc. 16th Int. Conf. on Database Systems for Advanced Applications*, pp. 214–225. Springer.

B. Chardin, J. Lacombe, and J. Petit. 2016. Chronos: a NoSQL system on flash memory for industrial process data. *Distrib. Parall. Databases*, 34(3): 293–319.

B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. McCormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, R. Ebenstein, N. Mikhaylin, H. Lee, X. Zhao, T. Xu, L. Perez, F. Shahmohammadi, T. Bui, N. Mckay, S. Aya, V. Lychagina, and B. Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *Proc. VLDB Endowment*, 12(12): 2022–2034.

F. Chiariotti, J. Holm, A. E. Kalør, B. Soret, S. K. Jensen, T. B. Pedersen, and P. Popovski. 2022. Query Age of Information: Freshness in Pull-Based Communication. *IEEE Trans. Commun.*, 70(3): 1606–1622.

J. A. Colmenares, R. Dorrigiv, and D. G. Waddington. 2017. A Single-Node Datastore for High-Velocity Multidimensional Sensor Data. In *Proc. 2017 IEEE Int. Conf. on Big Data*, pp. 445–452. IEEE.

A. Cuzzocrea and D. Saccà. 2013. Exploiting compression and approximation paradigms for effective and efficient online analytical processing over sensor network readings in data grid environments. *Concurrency and Computation — Practice & Experience*, 25(14): 2016–2035.

A. Cuzzocrea, J. Cecilio, and P. Furtado. 2014. StreamOp: An Innovative Middleware for Supporting Data Management and Query Functionalities over Sensor Network Streams Efficiently. In *2014 17th Int. Conf. on Network-Based Information Systems*, pp. 310–317. IEEE.

DB-Engines Ranking of Time Series DBMS, 2023. https://db-engines.com/en/ranking/time+series+dbms. Viewed: March 1, 2023.

L. Deri, S. Mainardi, and F. Fusco. 2012. tsdb: A Compressed Database for Time Series. In *Proc. 4th Int. Workshop on Traffic Monitoring and Analysis*, pp. 143–156. Springer.

X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao. 2018. UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics. *Proc. VLDB Endowment*, 11(7): 787–799.

G. Douglas and R. Lawrence. 2014. LittleD: A SQL Database for Sensor Nodes and Embedded Applications. In *Proc. 2014 ACM Symp. on Applied Computing*, pp. 827–832. ACM.

K. Echihabi, K. Zoumpatianos, and T. Palpanas. 2021. Big Sequence Management: Scaling up and Out. In *Proc. 24th Int. Conf. on Extending Database Technology*, pp. 714–717. OpenProceedings.org.

H. Elmeleegy, A. K. Elmagarmid, E. Cecchet, W. G. Aref, and W. Zwaenepoel. 2009. Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees. *Proc. VLDB Endowment*, 2(1): 145–156.

P. Esling and C. Agon. 2012. Time-Series Data Mining. *ACM Comput. Surv.*, 45(1).

A. Fakhrazari and H. Vakilzadian. 2017. A Survey on Time Series Data Mining. In *2017 IEEE Int. Conf. on Electro Information Technology*, pp. 476–481. IEEE.

M. Faschang, S. Cejka, M. Stefan, A. Frischenschlager, A. Einfalt, K. Diwold, F. P. Andrén, T. Strasser, and F. Kupzog. 2017. Provisioning, deployment, and operation of smart grid applications on substation level. *Comput. Sci. Res. Dev.*, 32(1-2): 117–130.

U. Fischer, F. Rosenthal, M. Böhm, and W. Lehner. 2010. Indexing Forecast Models for Matching and Maintenance. In *Proc. 14th Int. Conf. on Database Eng. and Applications*, pp. 26–31. ACM.

U. Fischer, M. Böhm, and W. Lehner. 2011. Offline Design Tuning for Hierarchies of Forecast Models. In *Datenbanksysteme für Business, Technologie und Web*, pp. 167–186. GI.

U. Fischer, D. Kaulakienė, M. E. Khalefa, W. Lehner, T. B. Pedersen, L. Šikšnys, and C. Thomsen. 2012a. Real-Time Business Intelligence in the MIRABEL Smart Grid System. In *Enabling Real-Time Business Intelligence, Proc. BIRTE Workshop*, pp. 1–22. Springer.

U. Fischer, F. Rosenthal, and W. Lehner. 2012b. F^{2}DB: The Flash-Forward Database System. In *Proc. 28th Int. Conf. on Data Engineering*, pp. 1245–1248. IEEE.

U. Fischer, C. Schildt, C. Hartmann, and W. Lehner. 2013. Forecasting the Data Cube: A Model Configuration Advisor for Multi-Dimensional Data Sets. In *Proc. 29th Int. Conf. on Data Engineering*, pp. 853–864. IEEE.

T. Fu. 2011. A review on time series data mining. *Eng. Appl. Artif. Intell.*, 24(1): 164–181.

C. Garcia-Arellano, A. J. Storm, D. Kalmuk, H. Roumani, R. Barber, Y. Tian, R. Sidle, F. Özcan, M. Spilchen, J. Tiefenbach, D. C. Zilio, L. Pham, K. Rakopoulos, A. Cheung, D. Pepper, I. Sayyid,

G. Gershinsky, G. Lushi, and H. Pirahesh. 2020. Db2 Event Store: A Purpose-Built IoT Database Engine. *Proc. VLDB Endowment*, 13(12): 3299–3312.

D. Garg, P. Shirolkar, A. Shukla, and Y. Simmhan. 2020. TorqueDB: Distributed Querying of Time-Series Data from Edge-local Storage. In *Proc. 26th Int. Conf. on Parallel and Distributed Computing*, pp. 281–295. Springer.

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

N. Glombiewski, P. Götze, M. Körber, A. Morgen, and B. Seeger. 2020. Designing an Event Store for a Modern Three-layer Storage Hierarchy. *Datenbank-Spektrum*, 20(3): 211–222.

A. J. Goodwin, D. Eytan, R. W. Greer, M. Mazwi, A. Thommandram, S. D. Goodfellow, A. Assadi, A. Jegatheeswaran, and P. C. Laussen. 2020. A practical approach to storage and retrieval of high-frequency physiological signals. *Physiol. Meas.*, 41(3).

T. Guo, T. G. Papaioannou, and K. Aberer. 2013. Model-View Sensor Data Management in the Cloud. In *Proc. 2013 IEEE Int. Conf. on Big Data*, pp. 282–290. IEEE.

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

T. Guo, T. G. Papaioannou, H. Zhuang, and K. Aberer. 2014b. Online Indexing and Distributed Querying Model-View Sensor Data in the Cloud. In *Proc. 19th Int. Conf. on Database Systems for Advanced Applications*, pp. 28–46. Springer.

T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. 2014. Bolt: Data management for connected homes. In *Proc. 11th USENIX Symp. on Networked Systems Design & Implementation*, pp. 243–256. USENIX.

H. B. Hamadou, T. Bach Pedersen, and C. Thomsen. 2020. The Danish National Energy Data Lake: Requirements, Technical Architecture, and Tool Selection. In *Proc. 2020 IEEE Int. Conf. on Big Data*, pp. 1523–1532. IEEE.

Y. Hao, X. Qin, Y. Chen, Y. Li, X. Sun, Y. Tao, X. Zhang, and X. Du. 2021. TS-Benchmark: A Benchmark for Time Series Databases. In *Proc. 37th Int. Conf. on Data Engineering*, pp. 588–599. IEEE.

J. Holm, A. E. Kalør, F. Chiariotti, B. Soret, S. K. Jensen, T. B. Pedersen, and P. Popovski. 2021. Freshness on Demand: Optimizing Age of Information for the Query Process. In *Proc. IEEE Int. Conf. on Communications*. IEEE.

J. Huang, A. Badam, R. Chandra, and E. B. Nightingale. 2015. WearDrive: Fast and Energy-Efficient Storage for Wearables. In *Proc. USENIX 2015 Annual Technical Conf.*, pp. 613–625. USENIX.

S. Huang, Y. Chen, X. Chen, K. Liu, X. Xu, C. Wang, K. Brown, and I. Halilovic. 2014. The Next Generation Operational Data Historian for IoT Based on Informix. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 169–176. ACM.

X. Huang, J. Wang, R. K. Wong, J. Zhang, and C. Wang. 2016. PISA: An Index for Aggregating Big Time Series Data. In *Proc. 25th ACM Int. Conf. on Information and Knowledge Management*, pp. 979–988. ACM.

X. Huang, J. Fan, Z. Deng, J. Yan, J. Li, and L. Wang. 2021. Efficient IoT Data Management for Geological Disasters Based on Big Data-Turbocharged Data Lake Architecture. *ISPRS Int. J. Geo-Inf.*, 10(11).

J. Hülsmann, J. Traub, and V. Markl. 2020. Demand-based Sensor Data Gathering with Multi-Query Optimization. *Proc. VLDB Endowment*, 13(12): 2801–2804.

J. Hülsmann, C. Li, J. Traub, and V. Markl. 2021. Automatic Tuning of Read-Time Tolerances for Optimized On-Demand Data-Streaming from Sensor Nodes. In *Proc. 24th Int. Conf. on Extending Database Technology*, pp. 517–522. OpenProceedings.org.

T. Ilsche, D. Hackenberg, R. Schöne, M. Höpfner, and W. E. Nagel. 2019. MetricQ: A Scalable Infrastructure for Processing High-Resolution Time Series Data. In *2019 IEEE/ACM Industry/University Joint Int. Workshop on Data-center Automation, Analytics, and Control*, pp. 7–12. IEEE.

S. K. Jensen, T. B. Pedersen, and C. Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Trans. Knowl. and Data Eng.*, 29(11): 2581–2600.

S. K. Jensen, T. B. Pedersen, and C. Thomsen. 2018. ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra. *Proc. VLDB Endowment*, 11(11): 1688–1701.

S. K. Jensen, T. B. Pedersen, and C. Thomsen. 2019. Demonstration of ModelarDB: Model-Based Management of Dimensional Time Series. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 1933–1936. ACM.

S. K. Jensen, T. B. Pedersen, and C. Thomsen. 2021. Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB$_+$. In *Proc. 37th Int. Conf. on Data Engineering*, pp. 1380–1391. IEEE.

S. K. Jensen, C. Thomsen, and T. B. Pedersen. 2023. ModelarDB: Integrated Model-Based Management of Time Series from Edge to Cloud. *Trans. Large-Scale Data- and Knowledge-Centered Syst.*, 53: 1–33.

T. Kamina and T. Aotani. 2019. An Approach for Persistent Time-Varying Values. In *Proc. 2019 ACM SIGPLAN Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 17–31. ACM.

T. Kamina, T. Aotani, and H. Masuhara. 2021. Signal Classes: A Mechanism for Building Synchronous and Persistent Signal Networks. In *35th European Conf. on Object-Oriented Programming*. Schloss Dagstuhl.

S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul. 2020. Towards Observability Data Management at Scale. *ACM SIGMOD Rec.*, 49(4): 18–23.

Y. Katsis, Y. Freund, and Y. Papakonstantinou. 2015. Combining Databases and Signal Processing in Plato. In *Proc. 7th Biennial Conf. on Innovative Data Systems Research*.

M. E. Khalefa, U. Fischer, T. B. Pedersen, and W. Lehner. 2012. Model-based Integration of Past & Future in TimeTravel. *Proc. VLDB Endowment*, 5(12): 1974–1977.

A. Khelifati, M. Khayati, and P. Cudré-Mauroux. 2019. CORAD: Correlation-Aware Compression of Massive Time Series using Sparse Dictionary Coding. In *Proc. 2019 IEEE Int. Conf. on Big Data*, pp. 2289–2298. IEEE.

U. Khurana, S. Parthasarathy, and D. S. Turaga. 2014. FAQ: A Framework for Fast Approximate Query Processing on Temporal Data. In *Proc. 3rd Int. Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pp. 29–45. JMLR.org.

C. Koncilia, T. Morzy, R. Wrembel, and J. Eder. 2014. Interval OLAP: Analyzing Interval Data. In *Proc. 16th Int. Conf. Data Warehousing and Knowledge Discovery*, pp. 233–244. Springer.

I. Kosen, C. Huang, Z. Chen, X. Zhang, L. Min, D. Zhou, L. Zhu, and Y. Liu. 2020. UPS: Unified PMU-Data Storage System to Enhance T+D PMU Data Usability. *IEEE Trans. Smart Grid*, 11(1): 739–748.

T. Kraska. 2021. Towards instance-optimized data systems. *Proc. VLDB Endowment*, 14(12): 3222–3232.

L. Lan, R. Shi, B. Wang, L. Zhang, and J. Shi. 2019. A Lightweight Time Series Main-Memory Database for IoT Real-Time Services. In *Proc. 6th Int. Conf. on Internet of Vehicles. Technologies and Services Toward Smart Cities*, pp. 220–236. Springer.

F. Lautenschlager, A. Kumlehn, J. Adersberger, and M. Philippsen. 2015. Fast and Efficient Operational Time Series Storage: The Missing Link in Dynamic Software Analysis. *Softwaretechnik-Trends*, 35(3).

F. Lautenschlager, M. Philippsen, A. Kumlehn, and J. Adersberger. 2017. Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data. In *Proc. 15th USENIX Conf. on File and Storage Technologies*, pp. 229–242. USENIX.

I. Lazaridis and S. Mehrotra. 2003. Capturing Sensor-Generated Time Series with Quality Guarantees. In *Proc. 19th Int. Conf. on Data Engineering*, pp. 429–440. IEEE.

A. Lerner and D. Shasha. 2003. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pp. 345–356. VLDB Endowment, Morgan Kaufmann.

C. Li, J. Li, J. Si, and Y. Zhang. 2017. FluteDB: An Efficient and Dependable Time-Series Database Storage Engine. In *Proc. SpaCCS 2017 Int. Workshops*, pp. 446–456. Springer.

C. Li, B. Li, M. Z. A. Bhuiyan, L. Wang, J. Si, G. Wei, and J. Li. 2018. FluteDB: An efficient and scalable in-memory time series database for sensor-cloud. *J. Parallel Distributed Comput.*, 122: 95–108.

J. Li, A. Badam, R. Chandra, S. Swanson, B. Worthington, and Q. Zhang. 2014. On the Energy Overhead of Mobile Storage Systems. In *Proc. 12th USENIX Conf. on File and Storage Technologies*, pp. 105–118. USENIX.

C. Lin, E. Boursier, and Y. Papakonstantinou. 2020. Plato: Approximate Analytics System over Compressed Time Series with Tight Deterministic Error Guarantees. *Proc. VLDB Endowment*, 13(7): 1105–1118.

A. Llusà-Serra, T. Escobet-Canal, and S. Vila-Marta. 2013. A Model for a Multiresolution Time Series Database System. In *Proc. 12th Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases*, pp. 55–60. WSEAS.

C. Luo, P. Tözün, Y. Tian, R. Barber, V. Raman, and R. Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *Proc. 22th Int. Conf. on Extending Database Technology*, pp. 1–12. OpenProceedings.org.

A. MacDonald. 2016. PhilDB: the time series database with built-in change logging. *PeerJ Comput. Sci.*, 2.

A. Marascu, P. Pompey, E. Bouillet, O. Verscheure, M. Wurst, M. Grund, and P. Cudre-Mauroux. 2013. MiSTRAL: An Architecture for Low-Latency Analytics on Massive Time Series. In *Proc. 2013 IEEE Int. Conf. on Big Data*, pp. 15–21. IEEE.

A. Marascu, P. Pompey, E. Bouillet, M. Wurst, O. Verscheure, M. Grund, and P. Cudre-Mauroux. 2014. TRISTAN: Real-Time Analytics on Massive Time Series Using Sparse Dictionary Compression. In *Proc. 2014 IEEE Int. Conf. on Big Data*, pp. 291–300. IEEE.

N. D. Mickulicz, R. Martins, P. Narasimhan, and R. Gandhi. 2015. When Good-Enough is Enough: Complex Queries at Fixed Cost. In *2015 IEEE 1st Int. Conf. on Big Data Computing Service and Applications*, pp. 89–98. IEEE.

S. K. Monga, S. K. R, and Y. Simmhan. 2019. ElfStore: A Resilient Data Storage Service for Federated Edge and Fog Resources. In *Proc. IEEE Int. Conf. on Web Services*, pp. 336–345. IEEE.

B. Mozafari and N. Niu. 2015. A Handbook for Building an Approximate Query Engine. *IEEE Data Eng. Bull.*, 38(3): 3–29.

C. Palmer, P. Lazik, M. Buevich, J. Gao, M. Berges, A. Rowe, R. L. Pereira, and C. Martin. 2014. Demo Abstract: Mortar.io: A Concrete Building Automation System. In *Proc. 1st ACM Conf. on Embedded Systems for Energy-Efficient Buildings*, pp. 204–205. ACM.

T. Palpanas. 2015. Data Series Management: The Road to Big Sequence Analytics. *ACM SIGMOD Rec.*, 44(2): 47–52.

T. Palpanas. 2016a. Data Series Management: The Next Challenge. In *Proc. Workshops of 32nd Int. Conf. on Data Engineering*, pp. 196–199. IEEE.

T. Palpanas. 2016b. Big Sequence Management: A glimpse of the Past, the Present, and the Future. In *Proc. 42nd Int. Conf. on Current Trends in Theory and Practice of Computer Science*, pp. 63–80. Springer.

T. Palpanas. 2020. Evolution of a Data Series Index. In *13th Int. Workshop on Information Search, Integration, and Personalization*, pp. 68–83. Springer.

T. Palpanas and V. Beckmann. 2019. Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA). *ACM SIGMOD Rec.*, 48(3): 36–40.

J. Paparrizos, C. Liu, B. Barbarioli, J. Hwang, I. Edian, A. J. Elmore, M. J. Franklin, and S. Krishnan. 2021. VergeDB: A Database for IoT Analytics on Edge Devices. In *Proc. 11th Conf. on Innovative Data Systems Research*.

J. Paris, J. S. Donnal, and S. B. Leeb. 2014. NilmDB: The Non-Intrusive Load Monitor Database. *IEEE Trans. Smart Grid*, 5(5): 2459–2467.

T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endowment*, 8(12): 1816–1827.

K. S. Perera, M. Hahmann, W. Lehner, T. B. Pedersen, and C. Thomsen. 2015. Modeling Large Time Series for Efficient Approximate Query Processing. In *Database Systems for Advanced Applications 2015 Int. Workshops, SeCoP, BDMS, and Posters*, pp. 190–204. Springer.

K. S. Perera, M. Hahmann, W. Lehner, T. B. Pedersen, and C. Thomsen. 2016. Efficient Approximate OLAP Querying Over Time Series. In *Proc. 20th Int. Conf. on Database Eng. and Applications*, pp. 205–211. ACM.

J. L. Pérez and D. Carrera. 2015. Performance Characterization of the servIoTicy API: an IoT-as-a-Service Data Management Platform. In *2015 IEEE 1st Int. Conf. on Big Data Computing Service and Applications*, pp. 62–71. IEEE.

V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endowment*, 6(11): 1080–1091.

S. Rhea, E. Wang, E. Wong, E. Atkins, and N. Storer. 2017. LittleTable: A Time-Series Database and Its Uses. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 125–138. ACM.

J. Ronkainen and A. Iivari. 2015. Designing a Data Management Pipeline for Pervasive Sensor Communication Systems. *Procedia Comput. Sci.*, 56: 183–188.

R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. 2001a. A Sequential Pattern Query Language for Supporting Instant Data Mining for e-Services. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pp. 653–656. VLDB Endowment, Morgan Kaufmann.

R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. 2001b. Optimization of Sequence Queries in Database Systems. In *Proc. 20th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pp. 71–81. ACM.

R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. 2004. Expressing and Optimizing Sequence Queries in Database Systems. *ACM Trans. Database Syst.*, 29(2): 282–318.

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

A. Sedighi, S. D'Mello, G. Isaacs, and D. Jacobson. 2020. BitemporalDB: A Bitemporal Database in the Cloud for Financial Services Data. In *2020 IEEE Int. Conf. on Smart Cloud*, pp. 68–73. IEEE.

M. Seidemann and B. Seeger. 2017. ChronicleDB: A High-Performance Event Store. In *Proc. 20th Int. Conf. on Extending Database Technology*, pp. 144–155. OpenProceedings.org.

M. Seidemann, N. Glombiewski, M. Körber, and B. Seeger. 2019. ChronicleDB: A High-Performance Event Store. *ACM Trans. Database Syst.*, 44(4).

A. L. Serra, S. Vila-Marta, and T. Escobet Canal. 2016. Formalism for a multiresolution time series database model. *Inf. Syst.*, 56: 19–35.

P. Seshadri, M. Livny, and R. Ramakrishnan. 1994. Sequence Query Processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 430–441. ACM.

P. Seshadri, M. Livny, and R. Ramakrishnan. 1995. SEQ: A Model for Sequence Databases. In *Proc. 11th Int. Conf. on Data Engineering*, pp. 232–239. IEEE.

P. Seshadri, M. Livny, and R. Ramakrishnan. 1996. The Design and Implementation of a Sequence Database System. In *Proc. 22th Int. Conf. on Very Large Data Bases*, pp. 99–110. VLDB Endowment, Morgan Kaufmann.

I. Shafer, R. R. Sambasivan, A. Rowe, and G. R. Ganger. 2013. Specialized Storage for Big Numeric Time Series. In *Proc. 5th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX.

A. B. Sharma, F. Ivančić, A. Niculescu-Mizil, H. Chen, and G. Jiang. 2014. Modeling and Analytics for Cyber-Physical Systems in the Age of Big Data. *SIGMETRICS Perf. Eval. Rev.*, 41(4): 74–77.

X. Shi, Z. Feng, K. Li, Y. Zhou, H. Jin, Y. Jiang, B. He, Z. Ling, and X. Li. 2020. ByteSeries: An In-Memory Time Series Database for Large-Scale Monitoring Systems. In *Proc. 11th ACM Symp. on Cloud Computing*, pp. 60–73. ACM.

M. Sichert and T. Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endowment*, 15(5): 1119–1131.

E. Siow, T. Tiropanis, X. Wang, and W. Hall. 2018. TritanDB: Time-series Rapid Internet of Things Analytics. *CoRR*, abs/1801.07947. https://arxiv.org/abs/1801.07947v1.

F. Solleza, A. Crotty, S. Karumur, N. Tatbul, and S. Zdonik. 2022. Mach: A Pluggable Metrics Storage Engine for the Age of Observability. In *Proc. 12th Conf. on Innovative Data Systems Research*.

Statista. 2022. Smart home. Technical Report did–27165–1, Statista.

M. Stonebraker and U. Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proc. 21st Int. Conf. on Data Engineering*, pp. 2–11. IEEE.

J. Traub, S. Breß, T. Rabl, A. Katsifodimos, and V. Markl. 2017. Optimized On-Demand Data Streaming from Sensor Nodes. In *Proc. 8th ACM Symp. on Cloud Computing*, pp. 586–597. ACM.

N. Tsiftes and A. Dunkels. 2011. A Database in Every Sensor. In *Proc. 9th Int. Conf. on Embedded Networked Sensor Systems*, pp. 316–332. ACM.

Á. Villalba, J. L. Pérez, D. Carrera, C. Pedrinaci, and L. Panziera. 2015. servIoTicy and iServe: a Scalable Platform for Mining the IoT. *Procedia Comput. Sci.*, 52: 1022–1027.

A. A. Visheratin, A. Struckov, S. Yufa, A. Muratov, D. A. Nasonov, N. Butakov, Y. Kuznetsov, and M. May. 2020. Peregreen - modular database for efficient storage of historical time series in cloud environments. In *Proc. USENIX 2020 Annual Technical Conf.*, pp. 589–601. USENIX.

B. Wang and J. S. Baras. 2013. HybridStore: An Efficient Data Management System for Hybrid Flash-Based Sensor Devices. In *Proc. 10th European Conf. on Wireless Sensor Networks*, pp. 50–66. Springer.

C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. A. McGrail, P. Wang, D. Luo, J. Yuan, J. Wang, and J. Sun. 2020. Apache IoTDB: Time-series Database for Internet of Things. *Proc. VLDB Endowment*, 13(12): 2901–2904.

Z. Wang, J. Xue, and Z. Shao. 2021. Heracles: An Efficient Storage Model and Data Flushing for Performance Monitoring Timeseries. *Proc. VLDB Endowment*, 14(6): 1080–1092.

R. S. Weigel, D. M. Lindholm, A. Wilson, and J. Faden. 2010. TSDS: high-performance merge, subset, and filter software for time series-like data. *Earth Sci. Informatics*, 3(1–2): 29–40.

J. W. Williams, K. S. Aggour, J. Interrante, J. McHugh, and E. Pool. 2014. Bridging High Velocity and High Volume Industrial Big Data Through Distributed In-Memory Storage & Analytics. In *Proc. 2014 IEEE Int. Conf. on Big Data*, pp. 932–941. IEEE.

S. Wöllauer, D. Zeuss, F. Hänsel, and T. Nauss. 2021. TubeDB: An on-demand processing database system for climate station data. *Computers & Geosciences*, 146.

J. Wu, P. Wang, N. Pan, C. Wang, W. Wang, and J. Wang. 2019. KV-Match: A Subsequence Matching Approach Supporting Normalization and Time Warping. In *Proc. 35th Int. Conf. on Data Engineering*, pp. 866–877. IEEE.

F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. 2014. Druid: A Real-time Analytical Data Store. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 157–168. ACM.

Y. Yang, Q. Cao, and H. Jiang. 2019. EdgeDB: An Efficient Time-Series Database for Edge Computing. *IEEE Access*, 7: 142295–142307.

H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Trans. Knowl. and Data Eng.*, 27(7): 1920–1948.

E. Zimányi, M. Sakr, and A. Lesuisse. 2020. MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS. *ACM Trans. Database Syst.*, 45(4).

K. Zoumpatianos and T. Palpanas. 2018. Data Series Management: Fulfilling the Need for Big Sequence Analytics. In *Proc. 34th Int. Conf. on Data Engineering*, pp. 1677–1678. IEEE.

K. Zoumpatianos, S. Idreos, and T. Palpanas. 2014. Indexing for Interactive Exploration of Big Data Series. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 1555–1566. ACM.

K. Zoumpatianos, S. Idreos, and T. Palpanas. 2015a. RINSE: Interactive Data Series Exploration with ADS+. *Proc. VLDB Endowment*, 8(12): 1912–1915.

K. Zoumpatianos, Y. Lou, T. Palpanas, and J. Gehrke. 2015b. Query Workloads for Data Series Indexes. In *Proc. 21st ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, p. 1603–1612. ACM.

K. Zoumpatianos, S. Idreos, and T. Palpanas. 2016. ADS: the adaptive data series index. *VLDB J.*, 25(6): 843–866.

K. Zoumpatianos, S. Idreos, and T. Palpanas. 2019. T-Store: Tunable Storage for Large Sequential Data. In *North East Database Day*.

Author Biographies

Søren Kejser Jensen

Søren Kejser Jensen is a postdoctoral researcher at the Center for Data-Intensive Systems (Daisy) at the Department of Computer Science, Aalborg University, Denmark. His research interests span multiple areas of computer science, including programming language design, compiler design and implementation, developer tooling, parallel and distributed computing, big data, database management systems, data warehousing, and extract-transform-load processing.

Torben Bach Pedersen

Torben Bach Pedersen is a Professor at the Center for Data-Intensive Systems (Daisy), Aalborg University, Denmark. His research concerns Predictive, Prescriptive, and Extreme-Scale Data Analytics with Digital Energy as the main application area. He is an ACM Distinguished Scientist, an IEEE Distinguished Contributor, a member of the Danish Academy of Technical Sciences, and holds an honorary doctorate from TU Dresden.

Christian Thomsen

Christian Thomsen is an associate professor at the Center for Data-Intensive Systems (Daisy) at the Department of Computer Science, Aalborg University, Denmark. His research interests include big data management, data lakes, data warehousing, and extract-transform-load processing.