Demonstration of ModelarDB: Model-Based Management of High-Frequency Time Series Across Edge, Cloud, and Client

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

Christian Schmidt Godiksen
Aalborg University, Denmark
csg@cs.aau.dk

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

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

ABSTRACT

Renewable Energy Sources (RESs) are monitored by many high-quality sensors that produce vast amounts of high-frequency time series data. This can be used to increase the renewable energy production and longevity of the RESs, e.g., yaw misalignment detection and predictive maintenance for wind turbines. It is currently not possible for wind turbine manufacturers and owners to use this data due to limits on bandwidth and storage that are infeasible to increase. Thus, they store simple aggregates which remove valuable outliers and fluctuations. As a remedy, we demonstrate the new model-based Time Series Management System (TSMS) ModelarDB. The participants can experience how ModelarDB ingests time series on the edge and compresses them as *segments* with metadata and so-called *models*. The models represent values within a user-defined absolute or relative error bound (even 0 or 0%). Participants can adjust many parameters and see how the segments are transferred to the cloud using much less bandwidth and storage than other popular solutions like Apache Parquet and Apache TsFile, e.g., up to 90%–99% less than Apache Parquet. Participants can analyze the time series on the edge, in the cloud, and on the client using SQL or Python. On the client, ModelarDB runs in-process to integrate with, e.g., Python. Thus, participants can see how ModelarDB efficiently manages high-frequency time series across edge, cloud, and client.

PVLDB Reference Format:
Søren Kejser Jensen, Christian Schmidt Godiksen, Christian Thomsen, and Torben Bach Pedersen. Demonstration of ModelarDB: Model-Based Management of High-Frequency Time Series Across Edge, Cloud, and Client. PVLDB, 18(12): 5247 - 5250, 2025.
doi:10.14778/3750601.3750643

1 INTRODUCTION

To efficiently manage Renewable Energy Sources (RESs) like wind turbines, they are monitored by many high-quality sensors. Thus, time series are ingested on the edge and transferred to the cloud for long-term storage. Finally, sub-sequences are copied to clients by data scientists for analysis as shown in Figure 1. We have learned from wind turbine manufacturers and owners that the time series

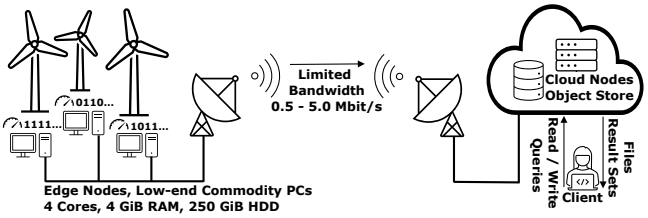


Figure 1: Wind turbine sensor data is ingested on the edge, transferred to the cloud over a connection with limited bandwidth, and stored in the cloud at high cost for use by clients

are generally stored as Apache Parquet files in an object store and that the analysis on the clients is generally done using Julia, Matlab, Python, or R. However, while the sensors produce vast amounts of high-frequency time series with valuable information as shown in Figure 2a, it is currently infeasible to transfer and store the raw time series in the cloud due to limited bandwidth and high storage cost. Instead, only simple aggregates, e.g., 10-minute averages, are stored which removes valuable outliers and fluctuations as shown in Figure 2b. Time Series Management Systems (TSMSs) that can manage time series across edge and cloud have been proposed [3, 4]. For example, Apache IoTDB can ingest time series on the edge and transfer them to the cloud. However, it provides limited compression and thus uses more bandwidth and storage than is feasible when managing high-frequency time series [1]. In addition, no TSMS integrates edge, cloud, and client [3, 4] in order to optimize the entire pipeline [5]. Thus, there is a need for a TSMS with integrated ingestion, transfer, storage, and analysis of high-frequency time series across edge, cloud, and client [5]. To remedy this, we developed the new model-based TSMS ModelarDB [8]. It is a full rewrite in Rust of a legacy JVM-based TSMS with the same name [6]. ModelarDB represents time series as segments with metadata and models such as simple polynomials. The models represent values

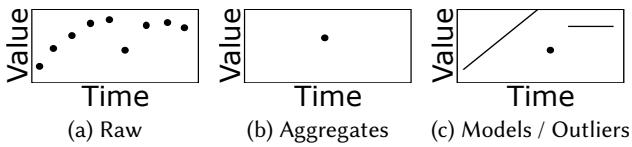


Figure 2: Representations of high-frequency time series

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750643

within a user-defined absolute or relative error bound (even 0 or 0%) and outliers are stored separately as shown in Figure 2c. This gives excellent compression as time series generally consist of simple reoccurring patterns that can be efficiently stored as model coefficients. For example, increasing values can be represented by one slope and intercept regardless of the number of values. The metadata makes it possible to reconstruct the values' exact timestamps. In addition, aggregates can be computed directly from the segments. Thus, using models reduces bandwidth, storage, and computing use. At the logical level, ModelarDB represents time series and non-time series data as relational tables that can be queried using SQL. In the scenario shown in Figure 1, ModelarDB instances on the edge ingest the time series and compress them as segments. The segments are then transferred to an object store in the cloud so they can be analyzed by ModelarDB instances running in the cloud and on the clients. An instance on a client is a Rust library that runs in-process to integrate with, e.g., Python. This allows data to be shared without copying it. In addition, data scientists can use the same API to express which data they need and how it should be transformed regardless of where it is stored. Currently, clients can use the API to query and write data to other ModelarDB instances, remote object stores, local disk, or local memory. Segments can also be copied or moved without decompressing them. Thus, data scientists can focus on data mining and machine learning while ModelarDB manages their data across edge, cloud, and client [5].

2 MODELARDB

Architecture: ModelarDB supports two types of relational tables: *Time Series Tables* and *Normal Tables*. *Table* is used to mean both. A Time Series Table can have a timestamp column of type `TIMESTAMP`, one or more error bounded value columns of type `FIELD`, and zero or more metadata columns of type `TAG`. For example, the SQL in Lines 1–3 in Listing 1 creates a Time Series Table for time series about wind. Data points can then be efficiently written using Apache Arrow `RecordBatches` or less efficiently using SQL as shown in Lines 5–7 in Listing 1. Time Series Tables compress time series as segments and store them in Delta Lake. A Normal Table can have any column types supported by Apache DataFusion and Delta Lake as it stores data directly in Delta Lake. Thus, the two types of tables only differ at the physical level and can, e.g., be queried together.

ModelarDB is purposely implemented using technologies that provide good integration with existing tools as it is infeasible for wind turbine manufacturers and owners to replace their entire infrastructure despite the benefits of ModelarDB. Apache Arrow is used to store data in memory and to share data in-process without making copies. Apache Arrow `RecordBatches` can also easily

```
1  CREATE TIME SERIES TABLE wind_turbine(
2    timestamp TIMESTAMP, wind_turbine TAG,
3    wind_speed FIELD, wind_direction FIELD(1.0%))
4
5  INSERT INTO wind_turbine VALUES (
6    '2025-09-01T12:00:00+00:00', 'JB007',
7     4.7987617, 12.195049)
```

Listing 1: Creating a Time Series Table and inserting data

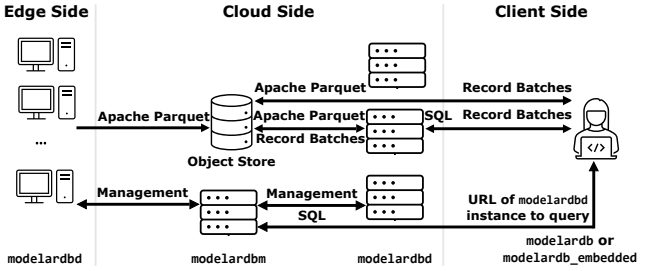
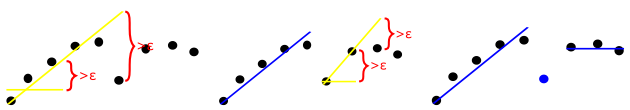


Figure 3: ModelarDB deployment on edge, cloud, and client

be converted to pandas with the `to_pandas()` method. Apache DataFusion is used for executing SQL queries against tables. Finally, Delta Lake is used to store data on disk. A full deployment of ModelarDB across edge, cloud, and client is shown in Figure 3. The TSMS server `modelardbd` manages tables and executes SQL queries, the manager `modelardbm` manages `modelardbd` instances, the Rust library `modelardb_embedded` manages tables and executes SQL queries in-process, and `modelardb` is a command-line client. As `modelardb_embedded` is a library it must be loaded into a process like CPython. Also, to make it simple to start deploying ModelarDB, `modelardbm` is not required for individual instances of `modelardbd`.

Edge: When data points are written to a Time Series Table they are batched and then compressed as shown in Figure 4. ModelarDB's model-based fitting method is designed to give a high compression ratio as bandwidth is the bottleneck and to have no parameters that users must configure except for the optional error bound. If a `FIELD` column has no error bound, lossless compression is used. ModelarDB currently implements extended versions of the model types Poor Man's Compression-MidRange (PMC-MR) [7] for fitting constant functions and Swing Filter (SWING) [2] for fitting linear functions to time series. Thus, sub-sequences that are constant within the error bound are represented by only a single coefficient no matter the number of data points, while sub-sequences that are linear within the error bound are represented by two coefficients no matter the number of data points. First, all model types fit as many data points as possible until they exceed the error bound as shown in Figure 4a. Then the model that uses the fewest bytes per value is combined with metadata to create a segment and compression continues from the data point that exceeded the selected model's error bound. If a segment represents so few data points that it does not provide any compression as shown in Figure 4b, compression is restarted from the following data point. Skipped data points are compressed using an extended version of the XOR-based method



(a) Start at 1st point (b) Start at 6th point (c) Finished batch

Figure 4: Ingestion of high-frequency time series as batches. Yellow models are considered, while blue models are selected

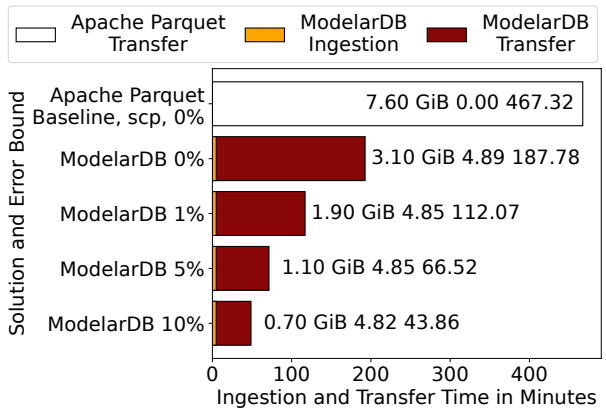


Figure 5: Ingestion and transfer time for wind turbine data

used by Gorilla [10] as shown in Figure 4c. The segments are continuously transferred to the object store in the cloud. The effect of compression on the transfer time over a realistic 2.5 Mbit/s connection [1] for Apache Parquet and ModelarDB with four error bounds can be seen in Figure 5. The bandwidth is clearly the bottleneck. Each `modelardbd` instance on the edge only executes the SQL queries it receives against the data it is storing on local disk.

Cloud: The object store in the cloud stores the data from all `modelardbd` instances on the edge. For workload balancing, SQL queries executed in the cloud are sent to `modelardbm` which returns the `modelardbd` instance in the cloud that the client should send the query to. By default, `modelardbd` instances in the cloud execute SQL queries against the data stored in the object store. However, clients can request that `modelardbd` instances in the cloud also forward the SQL query to `modelardbd` instances on the edge with an `INCLUDE address[, address]*` clause. The results are unioned by the `modelardbd` instance in the cloud and streamed to the client.

Client: Clients can use the command-line client `modelardb` or the Rust library `modelardbd_embedded` to query `modelardbd` instances. `modelardb` provides a read–eval–print loop that allows users to interactively execute SQL queries. `modelardbd_embedded` is designed to be loaded into a language runtime. It includes components from `modelardbd` and can manage tables in `modelardbd` instances, remote object stores, local disk, or local memory. For example, it can be loaded into CPython and then used through the simple API in Listing 2. The API operates on tables and supports creating tables, writing to tables, querying tables using SQL, etc. The API is designed to be low-level so users can easily create domain-specific abstractions. It is also designed to integrate well with other libraries, e.g., many methods accept SQL so users can write SQL by hand or use a library like SQLAlchemy Core. The API also makes it easy to copy or move data between tables efficiently. For example, a data scientist can copy data from a table managed by a `modelardbd` instance to a table managed by `modelardb_embedded` as shown in Listing 3. With the data on local disk it can be efficiently analyzed. Data is not copied or moved automatically to give data scientists full control. In addition, as `modelardbd_embedded` can manage tables in a remote object store, local disk, or local memory with its API, it can be used for in-process analytics without `modelardbd` instances.

```
def open_memory() -> Operations
def open_local(data_folder_path) -> Operations
def open_s3(endpoint, ...) -> Operations
def open_azure(account_name, ...) -> Operations
def connect(address) -> Operations
class Operations:
  def create(self, table_name, table_type)
  def tables(self) -> list[str]
  def schema(self, table_name) -> Schema
  def write(self, table_name, uncompressed_batch)
  def read(self, sql) -> RecordBatch
  def copy(self, sql, target, target_table_name)
  def read_time_series_table(...) -> RecordBatch
  def copy_time_series_table(self, ...)
  def move(self, source_table_name, target, ...)
  def truncate(self, table_name)
  def drop(self, table_name)
```

Listing 2: `modelardb_embedded`'s Python bindings to manage tables in `modelardbd` instances, remote object stores, local disk, or local memory. Parameters are shown as ... for space

```
import modelardb
from modelardb import operations
modelardbd = operations.connect(
  modelardb.Server("grpc://10.0.0.37:9999"))
copy_sql = "SELECT * FROM wind_turbine \
         WHERE wind_turbine = 'JB007'"
local =  operations.open_local("~/Data")
modelardbd.copy(copy_sql, local, 'wind_turbine')
read_sql = "SELECT * FROM wind_turbine \
         WHERE timestamp > '2025-09-01'"
pandas = local.read(read_sql).to_pandas()
```

Listing 3: Using `modelardb_embedded` to copy data from `modelardbd` to disk and then read it into a pandas DataFrame

3 DEMONSTRATION

Graphical User Interface: At the demonstration, participants can experience a full ModelarDB deployment with edge nodes, cloud nodes, and a client using the interface in Figure 6. The interface is designed to make it simple for participants to try different scenarios. For example, the number of data points to ingest per second can be set per table in the upper left corner. To keep the main window simple, some changes are performed using other windows. After making a change, participants can immediately see the effect. The box with the red bar on the left shows how much uncompressed data has been ingested per table, the box with the blue bar shows the size of the data as segments in Time Series Tables with different error bounds, and the box with the purple bar shows the size of the data in the solution ModelarDB is compared to which is Apache Parquet in Figure 6. Participants will be able to select from popular solutions to compare against. All tables are also shown for solutions that only use lossless compression as the number of data points to ingest can be changed at the table level. The bar chart at the top shows how the total compression ratio changes over time. The map shows the

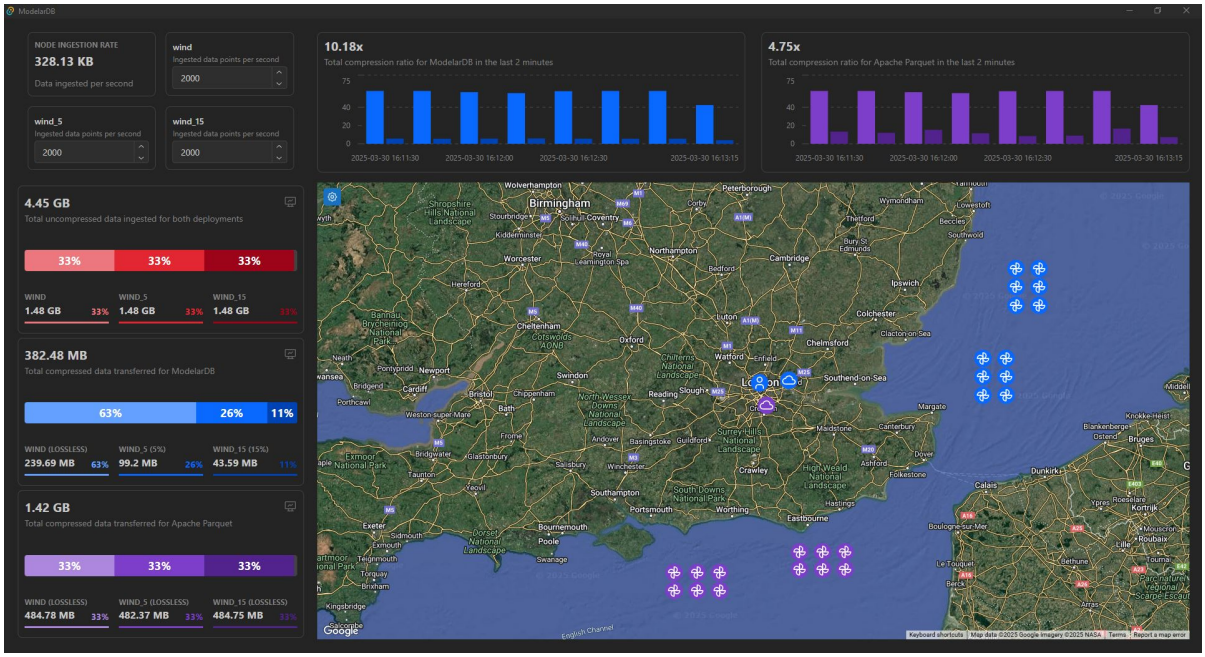


Figure 6: The ModelarDB demonstration interface with wind turbines in the English Channel whose data is analyzed in London

location of all edge nodes (wind turbine wings icon), cloud nodes (cloud icon), and clients (person icon). Clicking on an icon allows the participants to execute SQL queries on that node. Python code can also be executed on the clients using `modelardb_embedded`.

Data Set: For the demonstration, a recent real-life wind turbine data set will be used [9]. The data set was collected from a real-life wind turbine using a sampling interval of 2 seconds and contains data points for 10 days. It has time series about wind and nacelle direction. During the demonstration, the data set is extended by adding a continuously increasing offset to the original timestamps.

Demonstration Scenario: The demonstration allows participants to experience how ModelarDB can manage high-frequency time series across edge, cloud, and client. The scenario consists of a simulated production environment with a data scientist in London (client) analyzing sensor data from 12 wind turbines in the English Channel (edge nodes) continuously transferring data to a data center in London (cloud nodes). At the same time, an equivalent setup is configured to use a popular solution like Apache Parquet or Apache TsFile so participants can compare it with ModelarDB in real-time. The solution to use is selected by participants. Everything runs in Docker containers on a laptop, so the internet is not needed.

The participants first select the solution to compare ModelarDB with and then set the error bounds of the Time Series Tables, the number of data points to ingest per second, and the bandwidth of the connections between the nodes. The number of data points to ingest per second can be changed per table so participants can see how little bandwidth and storage ModelarDB requires compared to the other solutions when they ingest the same amount of data, and how much more data ModelarDB can store with the same amount of storage. Participants can also inspect how ModelarDB stores data at the physical level or execute SQL by clicking on an edge node, cloud node, or client. For the client, Python can also be executed. To make it simple for participants, configuration, query, and Python code examples will also be provided throughout the interface.

ACKNOWLEDGMENTS

This research was supported by 6G-XCEL (Horizon Europe grant 101139194). We also thank all our partners from the RES industry.

REFERENCES

[1] Abduvoris Abduvakhobov, Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2024. Scalable Model-Based Management of Massive High Frequency Wind Turbine Data with ModelarDB. *PVLDB* 17, 13 (2024), 4723–4732.

[2] Hazem Elmeleegy, Ahmed K Elmagarmid, Emmanuel Cecchet, Walid G Aref, and Willy Zwaenepoel. 2009. Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees. *PVLDB* 2, 1 (2009), 145–156.

[3] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. [n.d.]. Time Series Management Systems: A 2022 Survey. In *Data Series Management and Analytics (Forthcoming)*, Themis Palpanas and Kostas Zoumpatianos (Eds.). ACM. Preprint is at: https://vbn.aau.dk/da/publications/time-series-management-systems-a-2022-survey.

[4] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *TKDE* 29, 11 (2017), 2581–2600.

[5] Søren Kejser Jensen and Christian Thomsen. 2023. Holistic Analytics of Sensor Data from Renewable Energy Sources: A Vision Paper. In *ADBIS (Short Papers)*. Springer, 360–366.

[6] Søren Kejser Jensen, Christian Thomsen, Torben Bach Pedersen, Carlos Enrique Muñiz-Cuza, and Abduvoris Abduvakhobov. 2024. Why Model-Based Lossy Compression is Great for Wind Turbine Analytics. In *ICDE*. IEEE, 5667–5668.

[7] Iosif Lazaridis and Sharad Mehrotra. 2003. Capturing Sensor-Generated Time Series with Quality Guarantees. In *ICDE*. IEEE, 429–440.

[8] ModelarDB 2025. https://github.com/ModelarData/ModelarDB-RS. Commit: 66a4abf7f10f7a790dd195f0d962fe44183100cf, Viewed: 2025-07-24.

[9] Carlos Enrique Muñiz-Cuza, Søren Kejser Jensen, Jonas Brusokas, Nguyen Ho, and Torben Bach Pedersen. 2024. Evaluating the Impact of Error-Bounded Lossy Compression on Time Series Forecasting. In *EDBT*. OpenProceedings.org, 650–663.

[10] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *PVLDB* 8, 12 (2015), 1816–1827.