Extreme-Scale Model-Based Time Series Management with ModelarDB

Professor Torben Bach Pedersen

Aalborg University and ModelarData

tbp@cs.aau.dk

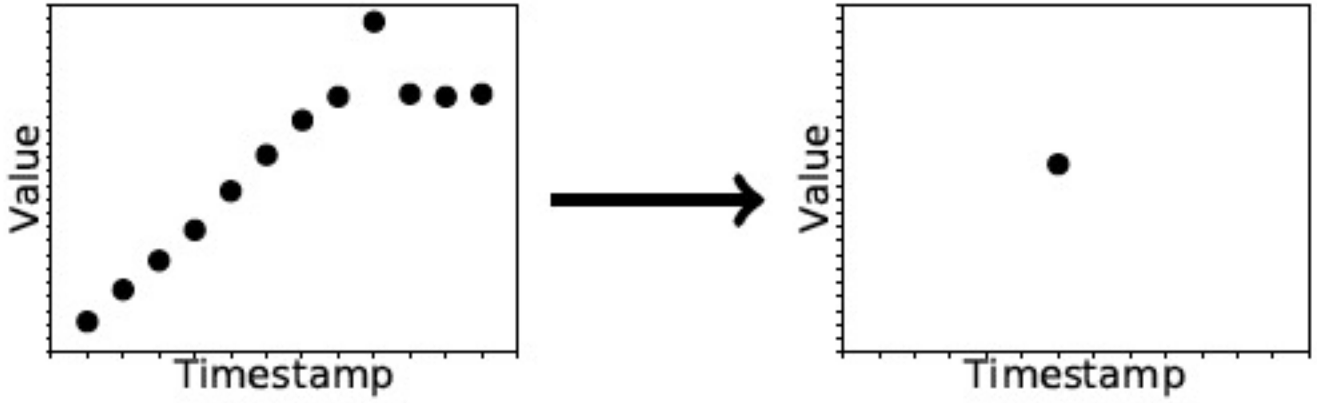
Joint work with Søren Kejser Jensen and Christian Thomsen

A Tale of Some Really Big and Fast Data…

- Denmark is **world no. 1** i wind energy
  - Come and visit and you will feel why ☺
  - No 1 turbine maker Vestas is DK, No 2 Siemens Gamesa has most R&D in DK
  - World record electricity from wind: **50% 2019**, going towards 100% wind/solar/..
- Wind turbines
  - **500 sensors** -> more than **2500 derived data streams**
  - 8 byte values sampled at 100+ Hz, 100+ turbines in a park
  - 100*100*2500 = 25 million values/second = **200+ MB/sec**
  - 200 MB * 3600 * 24 = 17.5+ TB/day = **8+ PB/year/park**
  - They want to store **20+ years for 1000s of parks**…
- Industry state of the art: **500 col SQL tables with 10 min avg**…
  - Makes high-frequency series impossibe - how can we improve?
- Data characteristics:
  - **Regular sampling interval, out-or-order corrected, short gaps**

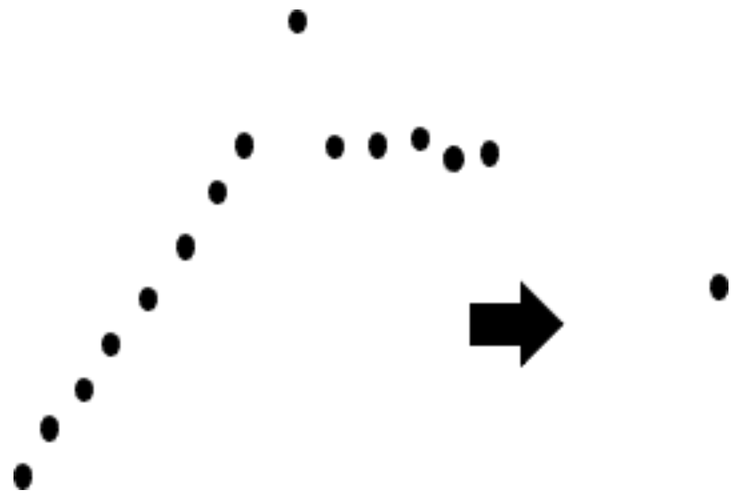
Observations

- From meetings with manufacturers, owners, and energy traders:
  - Turbines have **high-quality sensors** with **wired power and connectivity**
  - The storage needed makes **storing high-frequency sensor data infeasible**
  - Simple aggregates (e.g. **10-minute averages**) are **stored instead** of the **high-frequent series**, thereby **removing useful fluctuations and outliers**
- Many of the collected time series are **correlated** with each other
- They can be stored within a **user-defined error bound (possibly 0%)**
- **Metadata** is also stored and **aggregates** are the **primary query type**



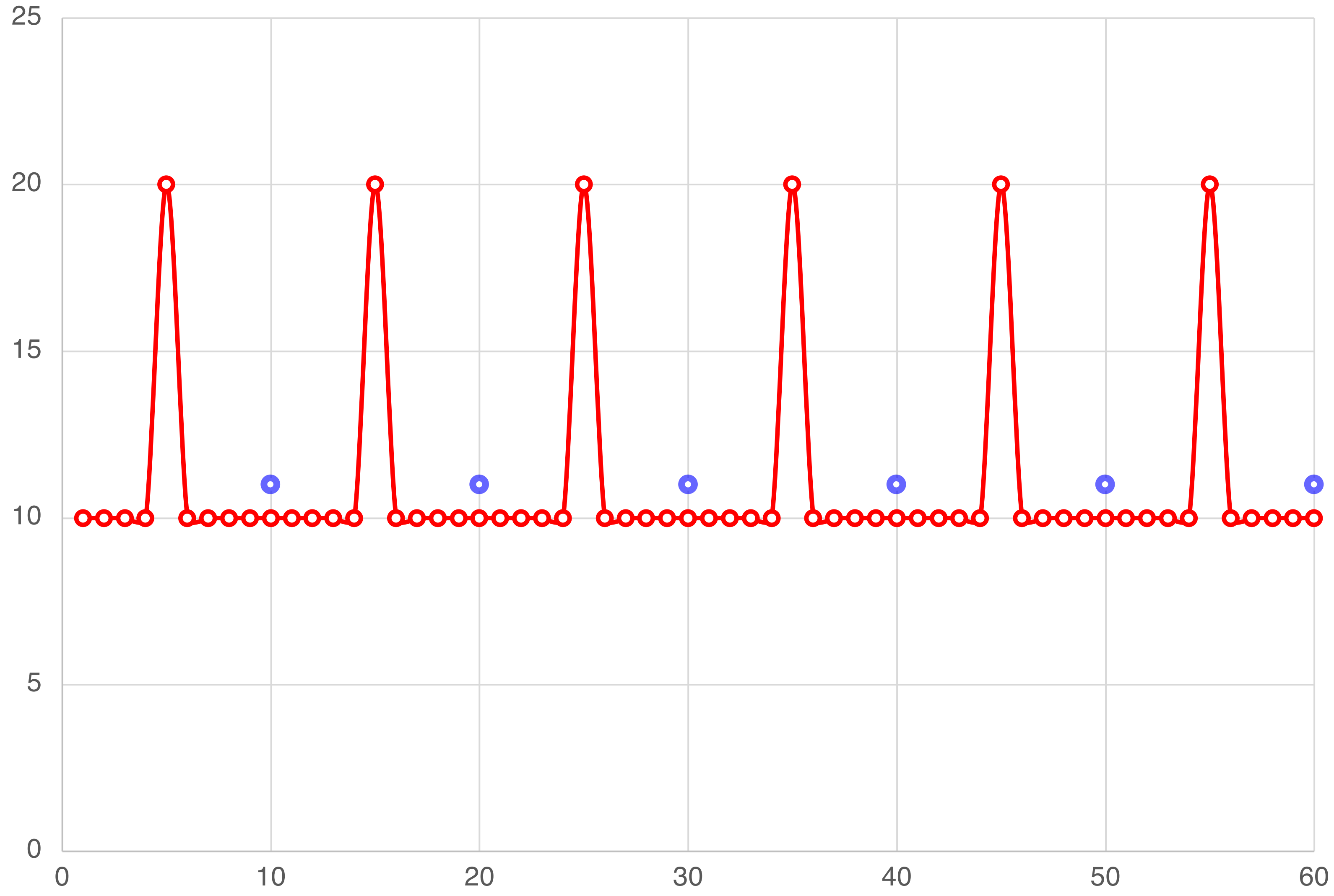
The current situation

- The available information is currently not exploited or stored
- Many monitoring solutions consider few (~100) sensor streams and store only a single value for every *x* minutes (e.g., the average)
  - *x* is typically ½, 1, 2, 5, or 10



- Important things might not be seen since **outliers and fluctuations are lost**

Example of ”missing the point” :-)



What we want to do…

- Store and use **all available sensor data**
- Support **efficient aggregate queries** on historical data
- Support **analysis** of data **while** it is being **ingested**
- **Detect** underperformance and other **problems immediately**
- Enable **predictive maintenance**

Why is that good? € + $CO2$

- For example, detect and **fix** a **problem before** the wind **turbine breaks**
- **Reduced costs** for service and spare parts
  - No over-time hours, crane booked in advance
- Service when there is little wind anyway
- Less downtime → more production
  - Delivery of a gearbox or wing can take months
- The **service cost** represents **11-30% of** the **onshore wind energy cost**
- Global wind service revenue: **8 billion USD**

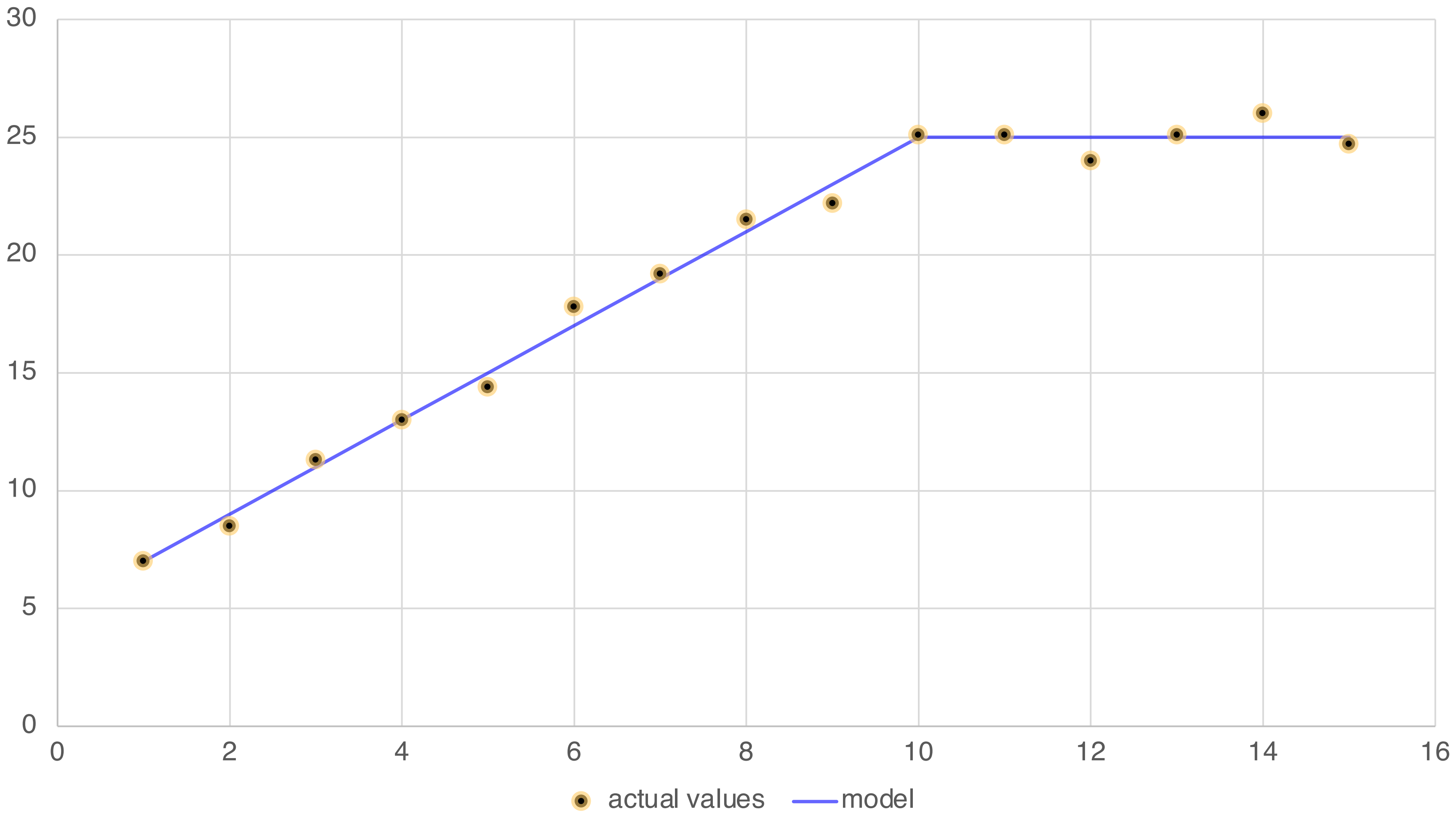
- **Most** importantly: making wind power cheaper **helps the Green Transition**
  - **Onshore wind** is already the **cheapest way** to install **new generation capacity**

How we do it

- Time-series can contain millions of points
- An efficient way to store and process them is to represent them by ***models***
- We use a ***model-based*** approach for the time-series data
- A **(user-defined) error-bound** can be set
  - For example **5%, 1%, or even 0%**

- **Allowing** an **error** in the representation can lead **to better compression** and **performance**

Simple example of models



More observations and some first results

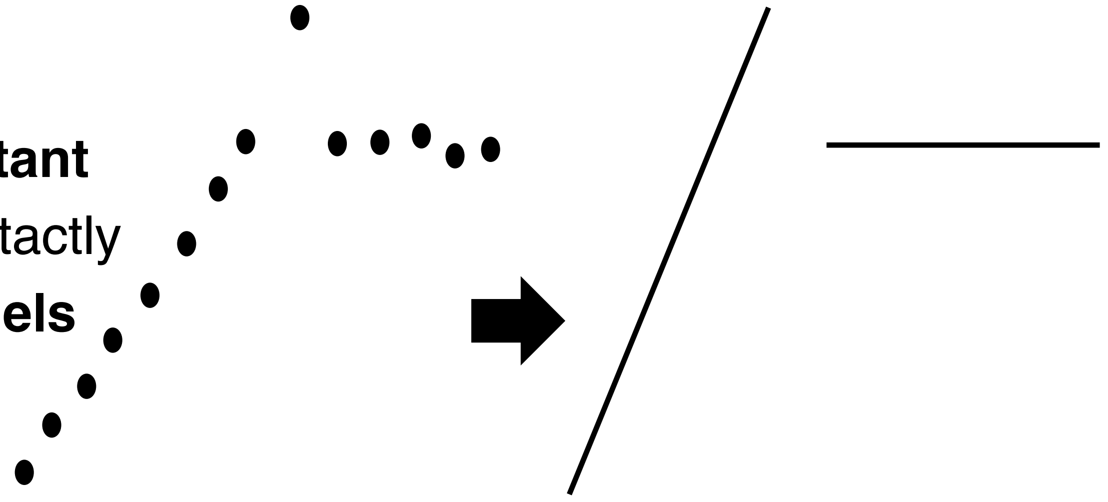
- Wind turbines produce big amounts of **high-quality sensor data**
- Data is collected as **regular time series** with **only few gaps** without values

- **High frequency** could benefit analysis but:
  - High frequency data **cannot be ingested fast enough**
  - High **query processing time limits use of historical data**
  - **Unfeasible** high **amounts of storage** are required
- Storage of real-life wind turbine data:

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

Model-based storage of time series

- A **model** is a **lossy** or **lossless** representation of a time series
- E.g., a linear function reduces the **values of N data points** to $\boldsymbol{a * x + b}$

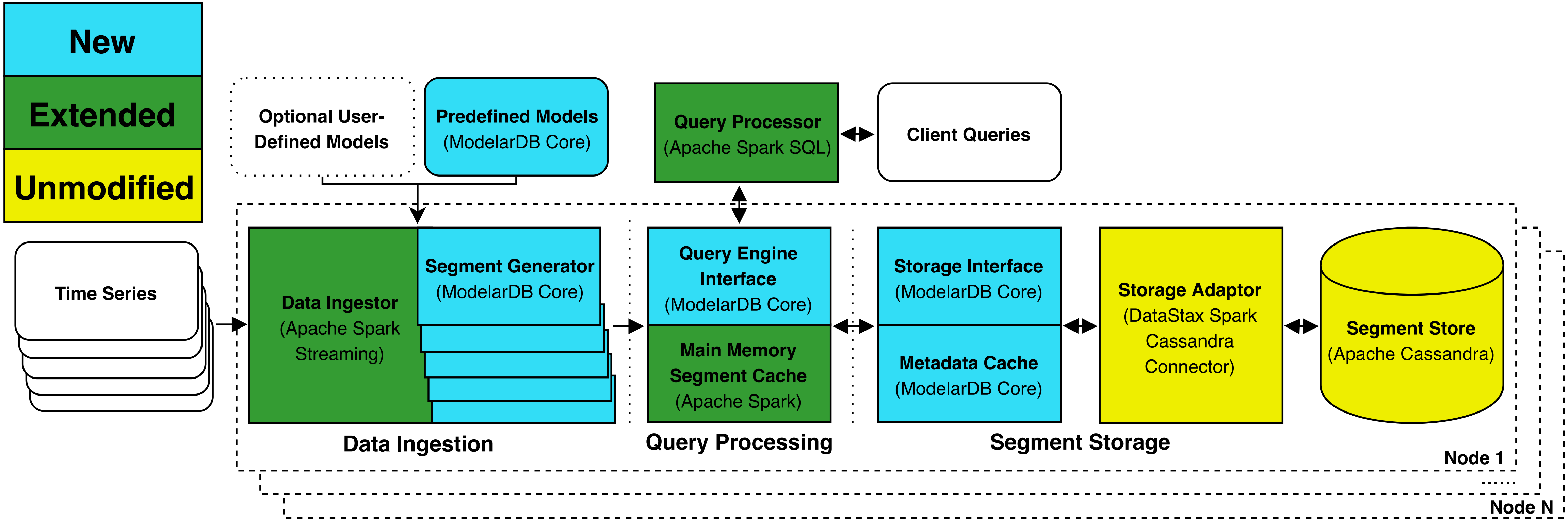


- Benefits from model-based storage:
  - The **storage** needed for a model can be **constant**
  - The **structure** of a time series is **preserved** intactly
  - **Queries** can be **answered directly from models**
- Problems with model-based storage:
  - The **best model** for a time series **changes over time**
  - **Long models** for high compression **increase latency**
- Our contributions **remove both of these problems**

ModelarDB Contributions

- A **general-purpose architecture** for a modular model-based time series management system (TSMS)
- An **adaptive online algorithm for multi-model compression** of time series
  - **Model-agnostic, extensible, allows gaps + offers low latency and high compression**
- A set of **methods and optimizations** for a model-based TSMS:
  - A **database schema** to store multiple time series as models
  - Methods to **push-down predicates** to a key-value store storing models
  - Methods to **execute optimized aggregate functions** directly on models
  - Use of static code-generation to **optimize projections**
  - **Dynamic extensibility** for **adding models** without recompiling the TSMS
- **ModelarDB** – an **open-source** implementation of our architecture
  - Available at github.com/skejserjensen/ModelarDB under version 2.0 of the Apache License

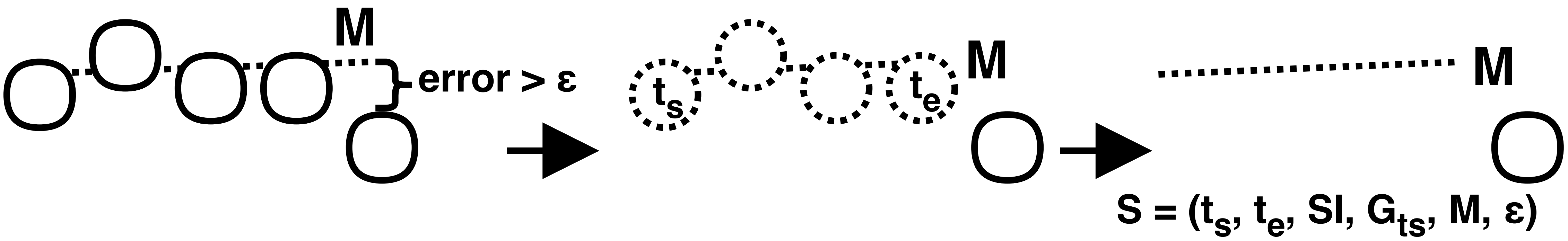
Architecture



- All portable functionality is part of a separate library named **ModelarDB Core**
- Our implementation interfaces **ModelarDB Core** with **Spark** and **Cassandra**
  - ModelarDB can be deployed on unmodified instances of Spark and Cassandra

Ingestion

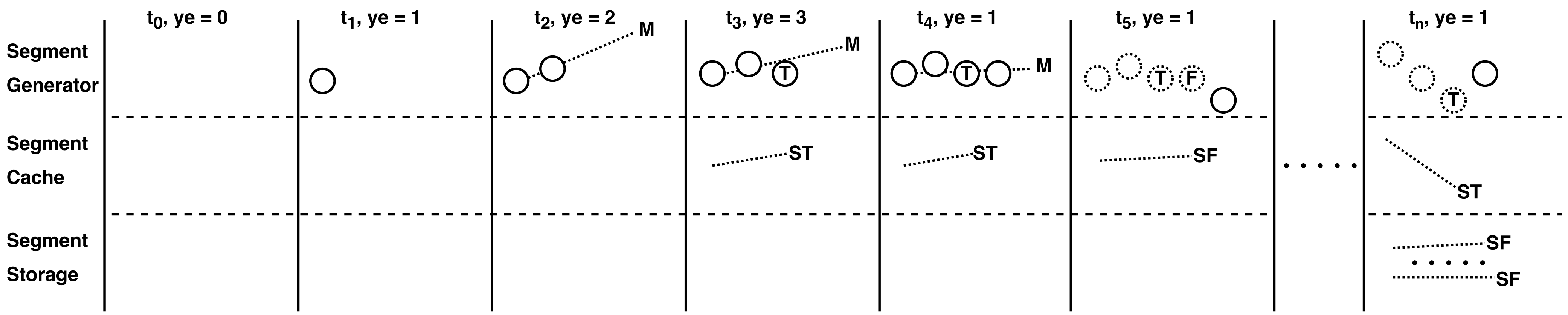
- Models are incrementally fitted and emitted as part of **segments with metadata**:
  - **Temporary Segment:** Holds an unfinished model cached in memory for low latency
  - **Finalized Segment:** Holds a finished model cached in memory and persisted to disk
- Models are fitted in sequence until all would exceed the error bound:



- ModelarDB Core includes four model types, users can **optionally add** more:
  - **PMC-MR** (Constant), **Swing** (Linear), **Facebook** (Lossless), **Uncompressed** (None)

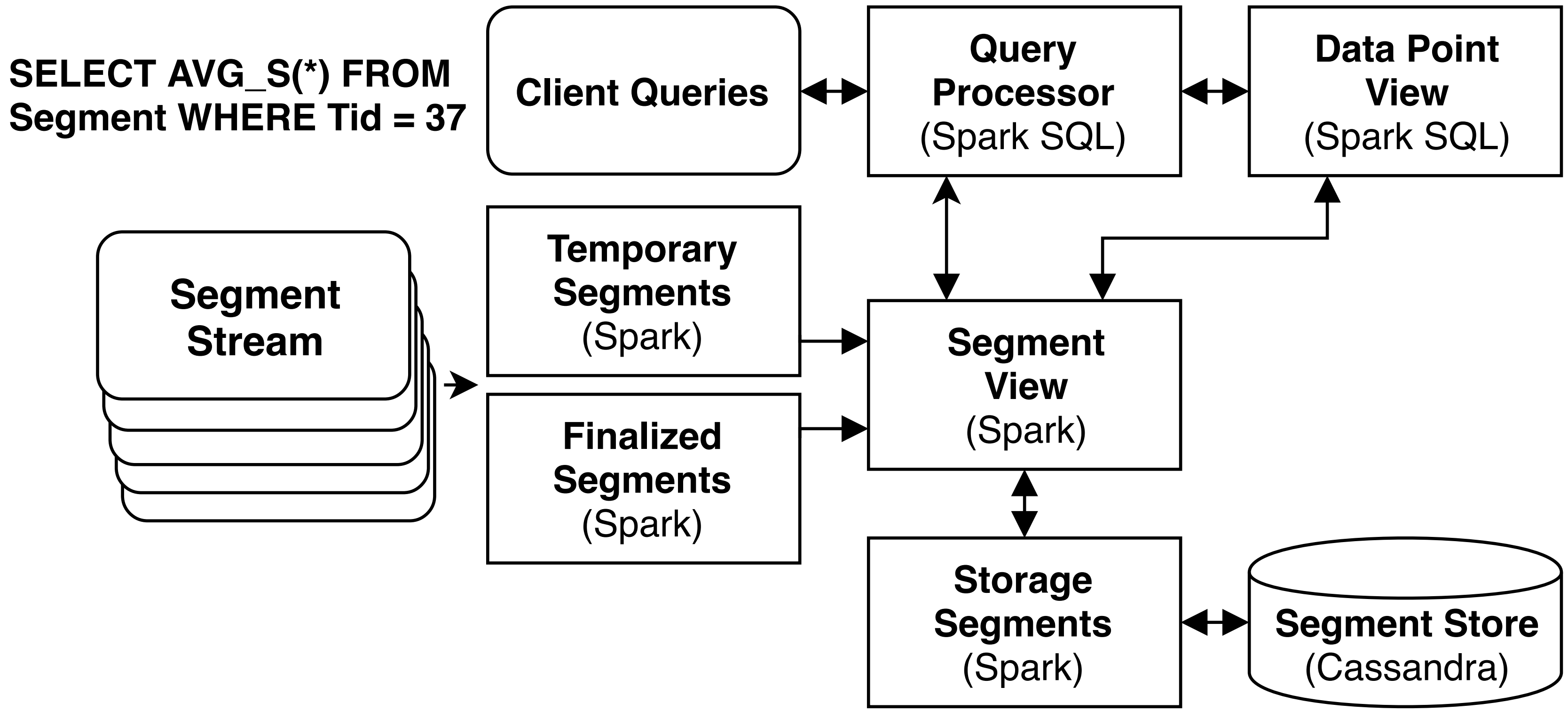
More details on ingestion

- **Fitting** seens as a **black box** to **support user-defined models**
  - Each **model implements representing data points+measure error**
  - **Model** is **passed data points** while user-def **error-bound** holds
- Models provide **trade-off between compression and latency**
  - **Longer models** give **better compression** but **higher latency**
- Example: **max latency of 3 data points (ye)**
  - Single model (**linear**) used to represent data points
  - Multiple models: the next model is evaluated when this fails



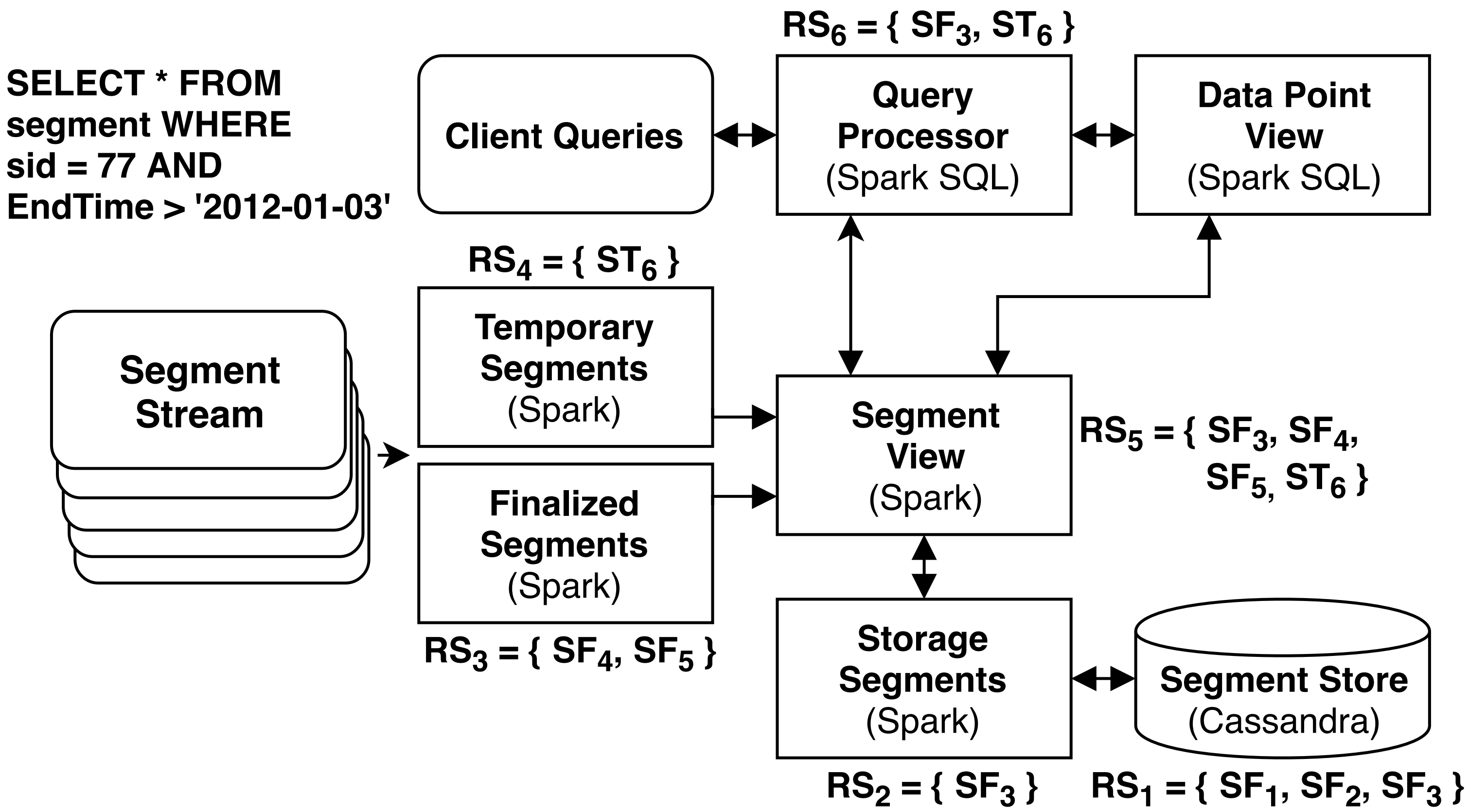
Query Processing

- ModelarDB uses **SQL for queries** to provide a familiar interface
- Two views are provided to allow for queries at different granularities:
  - **DataPoint View:** executes queries on data points reconstructed from segments
  - **Segment View:** efficiently executes aggregate queries directly on segments



Query Processing Example

- Example using the Segment View
  - Results is $\mathbf{SF_3}$ and $\mathbf{ST_6}$
  - $\mathbf{SF_3}$ resides on **disk**
  - $\mathbf{ST_6}$ resides **in memory**
- Abbreviations:
  - **RS:** Result Set
  - **ST:** Temporary Segment
  - **SF:** Finalized Segment



Storage

Tid (PK)	SI
1	60000
2	120000
3	30000

Time Series

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

Segment

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

Model

- **Time Series:** Stores time series metadata
- **Model:** Stores model types utilized for segments
- **Segment:** Stores segments emitted for each time series
- The bulk of the data is stored as part of the segment table

Predicate Push-Down

- ModelarDB uses a three table schema for storing time series as segments

Tid (PK)	SI
1	60000
2	120000
3	30000

Time Series

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

Segment

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

Model

- ModelarDB performs predicate push-down as a multi-step procedure:
  - **Data Point View:** Predicates are rewritten and pushed to the Segment View
  - **Segment View:** Predicates are pushed without changes to the Storage Interface
  - **Storage Interface:** Predicates are rewritten and pushed to the Segment Store
    - The Segment Store can have imprecise evaluation of the predicates (i.e., with false positives)

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

Data Point View

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

Segment View

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

Segment View

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

Casandra Segment Storage

Code Generation for Projection

- **Overhead of projections** are **reduced** using **optimized lambda functions**
- As the **columns** are **known**, the projection **code** is **generated** at **compile time**
- The correct function is found using a key created from the requested columns

```
def getDataPointGridFunction
   (columns: Array[String]): (DataPoint => Row) = {
  val target = getTarget(columns, dataPointView)
   (target: @switch) match {
     //Permutations of ('tid')
     case 1 => (dp: DataPoint) => Row(dp.tid)
     ...
     //Permutations of ('tid', 'ts', 'value')
     ...
     case 321 => (dp: DataPoint) => Row(dp.value,
       new Timestamp(dp.timestamp), dp.tid)
   }
}
```

Model-based aggregation

- **Queries** on **Segment View** executed **directly on segments if possible**
- Segments can **implement optimized methods for aggregate queries**
  - E.g., **sum for Swing** can be computed in **constant time** as shown below
- Aggregates are computed from reconstructed data points as a **fallback**

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

Extensibility with user-defined models

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

Model

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

Segment

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

- ModelarDB Core includes a few models, but users can load more dynamically (no need to recompile/restart)
- Models and segments must implement this interface to be used by ModelarDB

Query Examples

```
SELECT SUM(Value) FROM DataPoint WHERE Tid = 3
SELECT SUM_S(*) FROM Segment WHERE Tid = 3

SELECT AVG_SS( START(*, '2012-01-03 12:30') )
FROM Segment WHERE EndTime > '2012-01-03 12:30'

SELECT * FROM DataPoint WHERE Tid = 3
AND TS < '2012-04-22 12:25'
```

Listing 2: Query examples supported in ModelarDB

- A set of example queries supported by ModelarDB's two views
- Operations on the Segment View are implemented as **UDAFs and UDFs**

Evaluation - Storage

- 6 + 1 Laptops, **EH** (583 GiB, 100 ms), **ER** (488 GiB, 1 s), **EP** (339 GiB, 60 s)

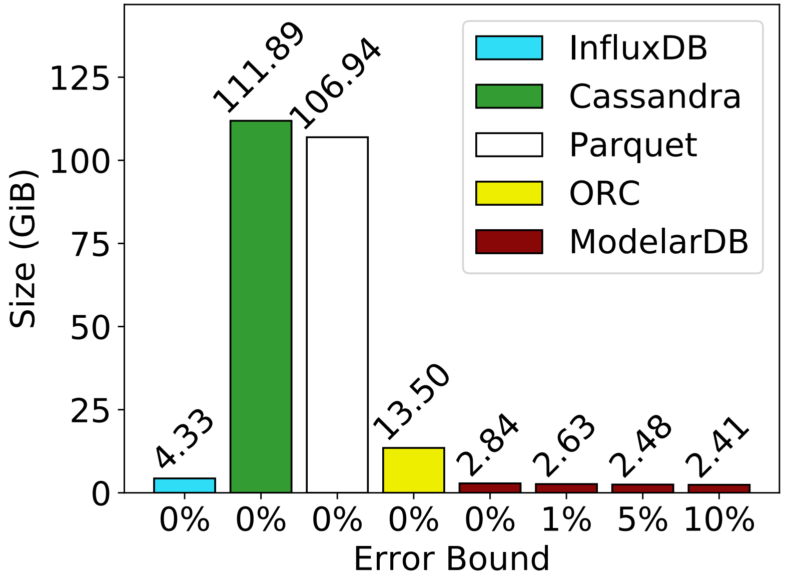


Figure: Storage, EH

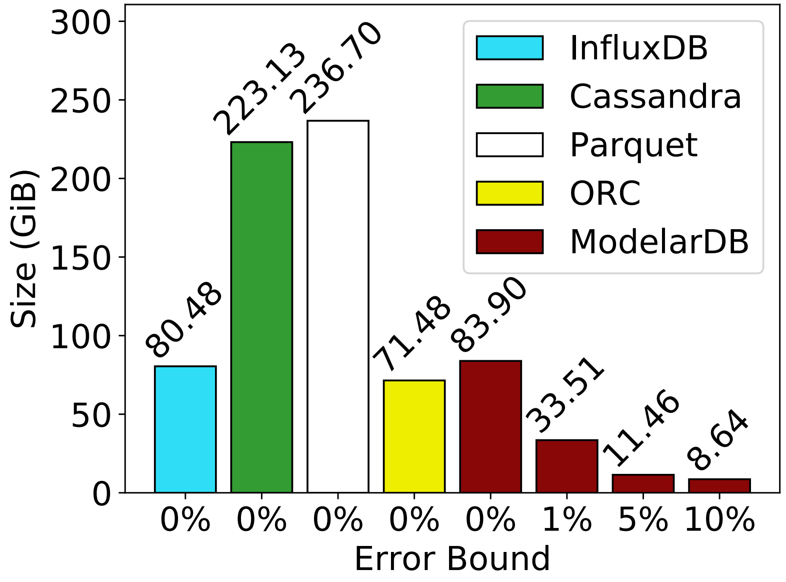


Figure: Storage, ER

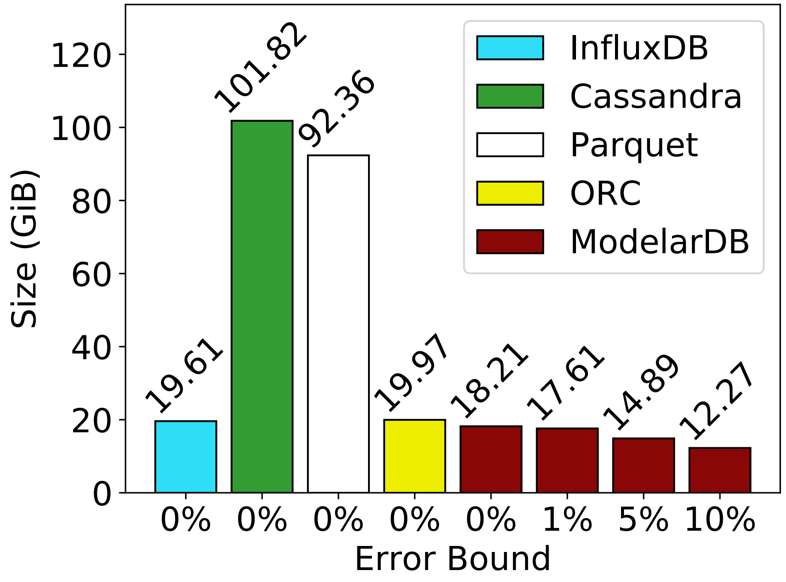


Figure: Storage, EP

- **ModelarDB provides better compression using model-based storage**
  - **Best compression** ratio **for high frequency data** (EH, ER) and **increases** with **error bound**
  - **Average error** is **0.005%** (EH**), 2.5%** (ER) and **0.73%** (EP) for a 10% error bound
  - ModelarDB **degrades gracefully with more outliers**

Evaluation - Adaptability

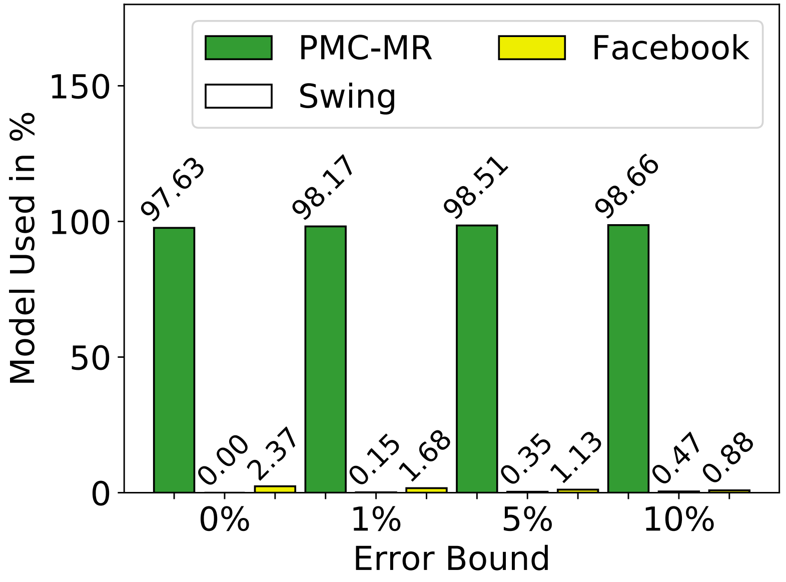


Figure: Adaptability, EH

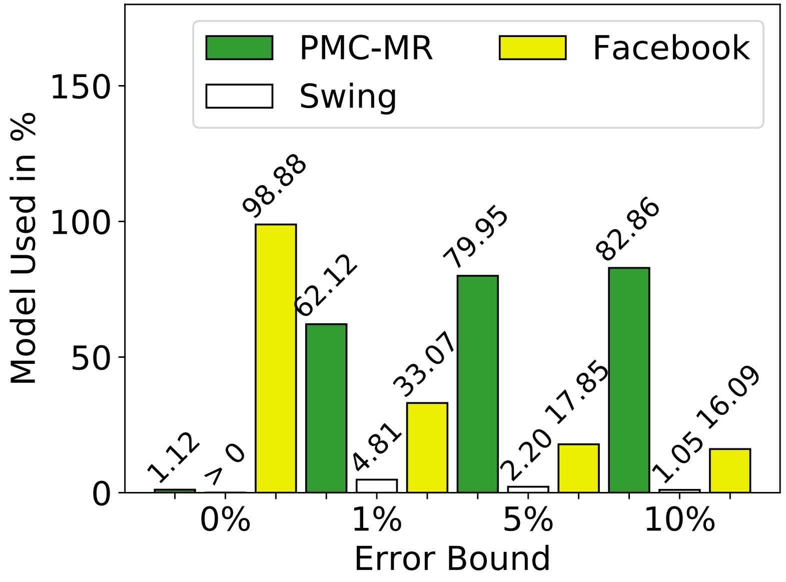


Figure: Adaptability, ER

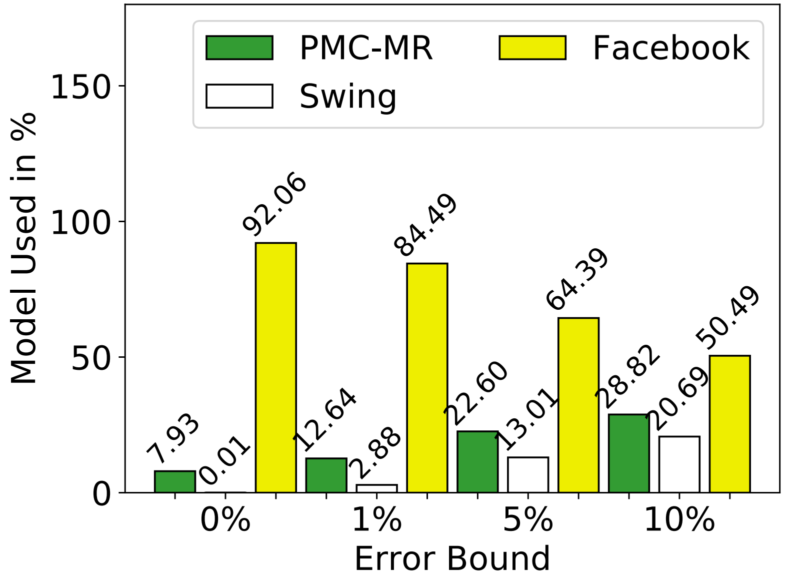


Figure: Adaptability, EP

- **ModelarDB chooses an appropriate model for each part of a series**
  - **Different models used** for each data set and **linear models** are used with **0%** error bound
  - The system is **extensible** and users can implement other models to increase **adaptability**

Evaluation – Ingestion and Query Processing

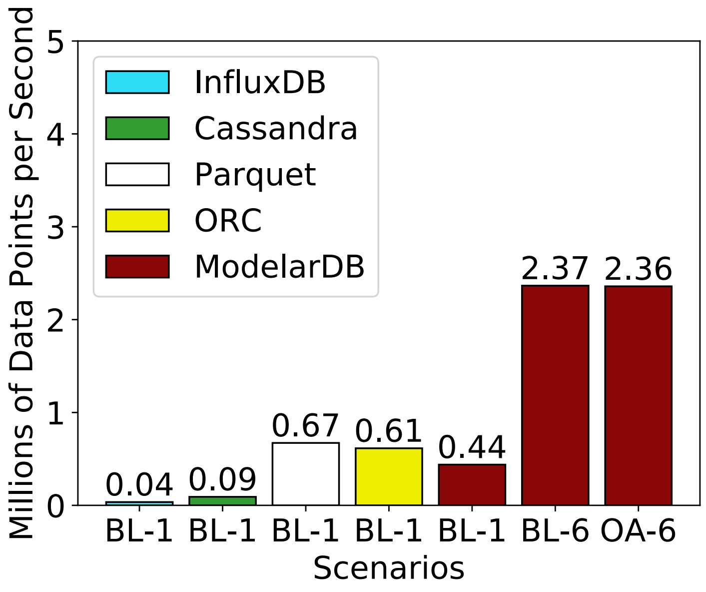


Figure: Ingestion, ER

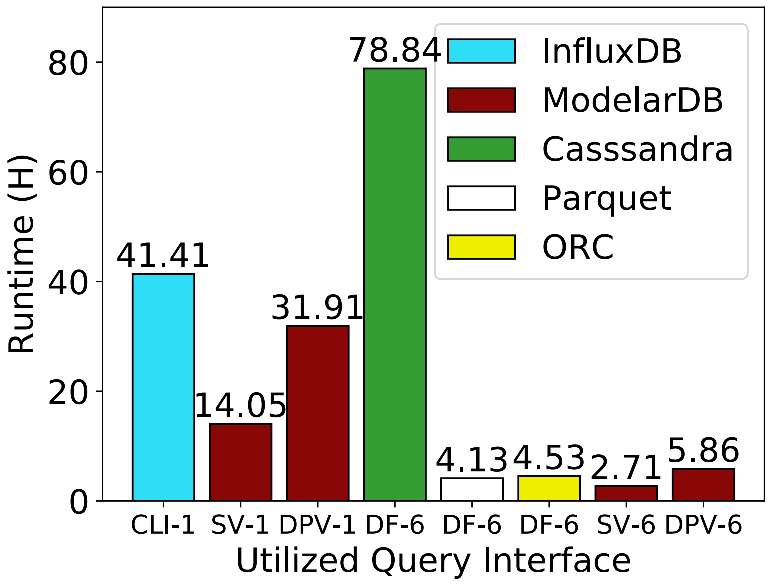


Figure: Aggregate Queries, ER

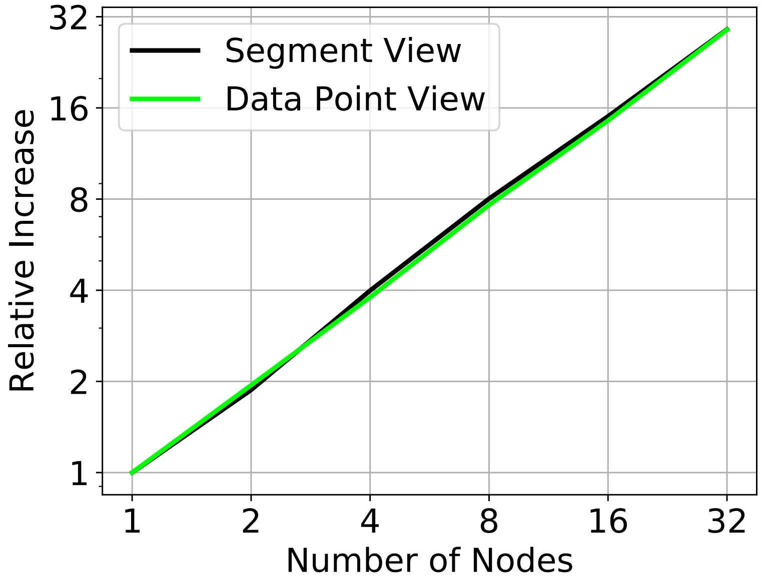


Figure: Scalability (Azure), ER

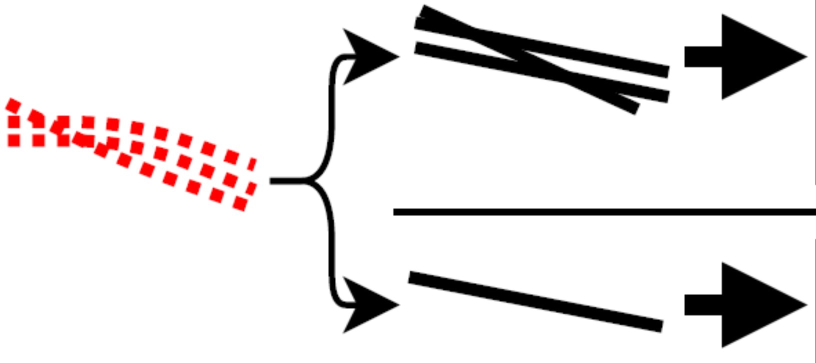
- **ModelarDB has fast ingestion, fast large aggregates and linear scalability**
  - Only InfluxDB, Cassandra, and ModelarDB can answer queries while ingesting data points
  - The paper shows ModelarDB is competitive with other systems for small scale queries

ModelarDB v1: how far did we get?

- **Summary:**
  - Storing sensor data as **simple aggregates discards valuable information**
  - **Model-based compression provides multiple benefits** over simple aggregates
  - Proposed the model-based TSMS **ModelarDB** based on:
    - A **general architecture** for a **modular model-based TSMS**
    - An algorithm for **online multi-model compression** of time series
    - A set of **methods and optimizations for a model-based TSMS**
  - Evaluation showed that **ModelarDB hits a sweet spot** by providing:
    - **Fast ingestion**
    - **Good compression**
    - **Fast, scalable online aggregate query processing**

- But we can do **even better…**

Next step: Exploiting correlation



Start Time	End Time	Model
100	2300	v = -0.047t+192.2
100	2300	v = -0.046t+180.1
100	2300	v = -0.057t+205.4

Start Time	End Time	Model
100	2300	v = -0.0465t+186.1

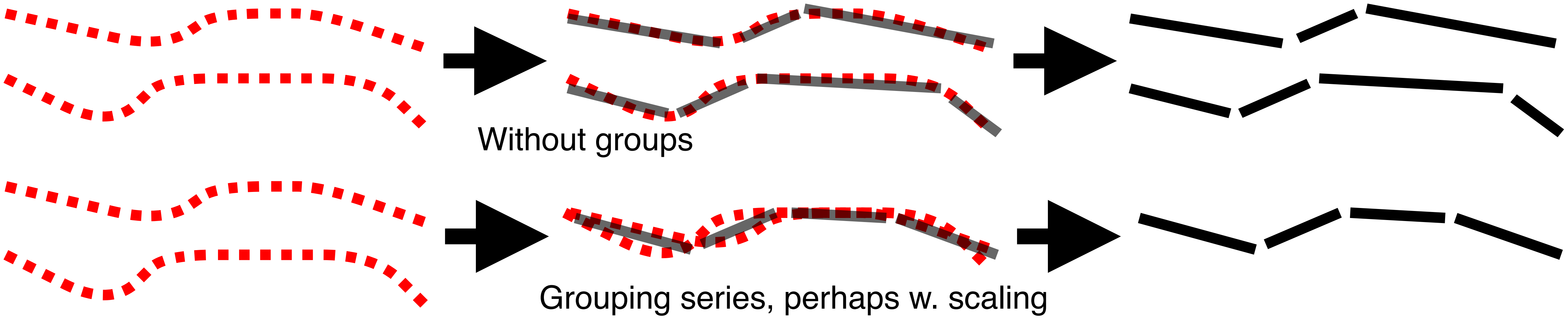
- **Detecting** correlation in data is an orthogonal problem
  - We let the user hint correlation
- If the time series in a group can not (no longer) be represented by a single model, ModelarDB *splits* the group
  - To respect the error bound
  - The time series can be *joined* again later

ModelarDB+ (v2) contributions

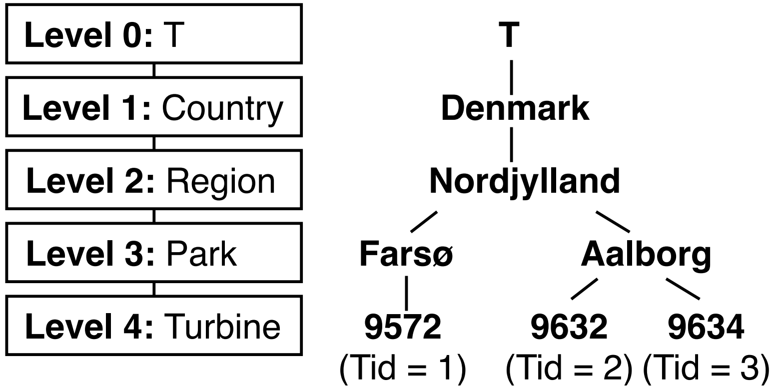
- Compression of time series groups using multiple model types, we call this type of compression **Multi-Model Group Compression (MMGC)**
- **Group Online Lossy and lossless Extensible Multi-Model (GOLEMM)**
  - **First Multi-Model Group Compression method** for time series and model types **extended** to **compress** time series **groups**
- **Primitives** for users to effectively group time series, and a method that **automatically groups** time series **using** their **metadata** as **dimensions**
- Algorithms for executing simple and **multi-dimensional aggregate queries** on models representing values from time series groups
- **ModelarDB+** a version of the **open-source** distributed model-based time series management systems ModelarDB with our methods added:
  - Available at github.com/skejserjensen/ModelarDB under Apache 2.0

Grouping Correlated Time Series

- Additional compression is achieved by **compressing time series in groups**

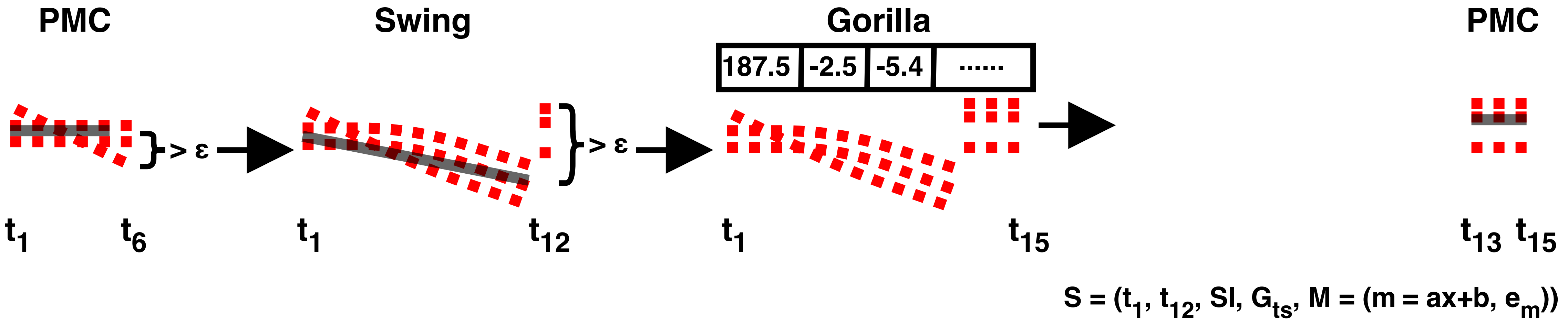


- Time series first **statically** grouped using metadata:
  - Time series source and dimensions
  - **Dimensions** contain hierarchically organized members
- Users can indicate correlation using our **primitives**:
  - Time series sources, members in dimensions, and the distance between two sets of dimensions



Ingesting Correlated Time Series

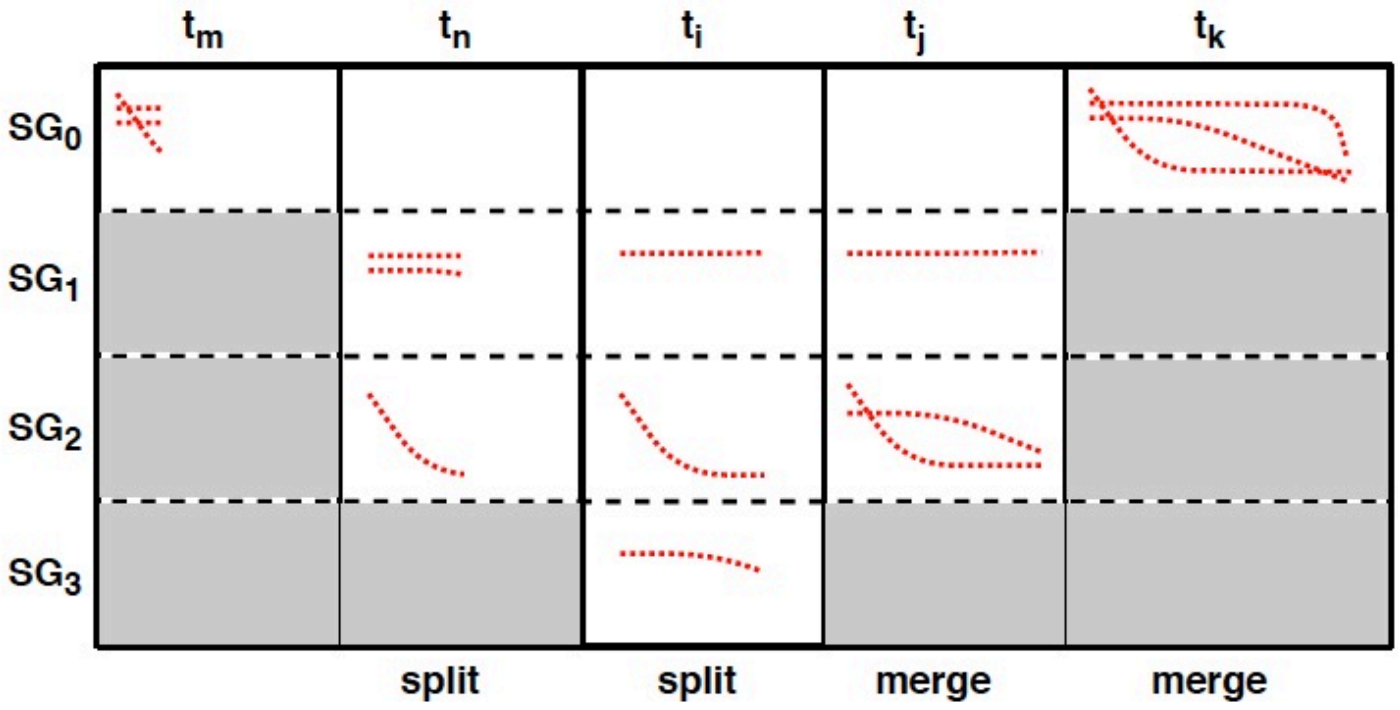
- Models are incrementally fitted and emitted as part of **segments with metadata**:
  - **Temporary Segment:** Holds an unfinished model cached in memory for low latency
  - **Finalized Segment:** Holds a finished model cached in memory and persisted to disk
- Models are fitted in sequence until all would exceed the error bound:



- ModelarDB Core includes four model types, users can **optionally add** more:
  - **PMC-Mean** (Constant), **Swing** (Linear), **Facebook** (Lossless), **Uncompressed** (None)

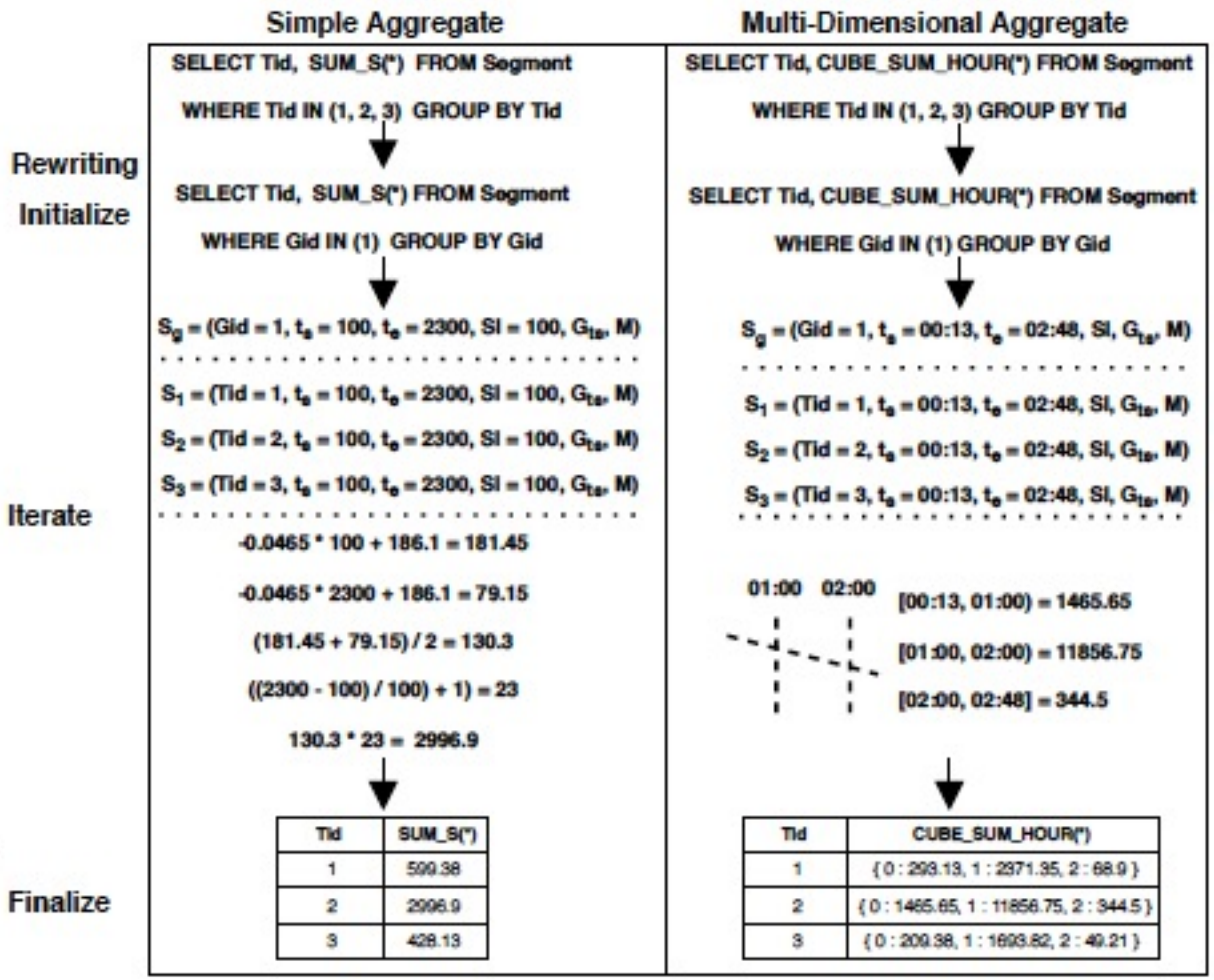
Dynamic Grouping

- The time series in a group might **temporarily not** be **correlated**:
- For **example**, a **temperature** sensor can be **obscured by clouds**
- This can be **efficiently detected** as the **compression ratio** is **reduced**
- **Dynamically splitting** and **merging** groups **remedies** this **problem**:



ModelarDB+ Query Processing

- **Multidimensional** aggregates, e.g., CUBE
- Denormalized user-defined **<Dimensions>**
- Data Point View
  - Interface: Tid int, TS timestamp, Value float, <Dimensions>
- Segment View
  - Interface: Tid int, StartTime timestamp, EndTime timestamp, SI int, Mid int, Parameters blob, Gaps blob, <Dimensions>
- UDAFs for aggregation on segments are suffixed with _S,
  - COUNT_S
- UDAFs for aggregation over time on segments are suffixed with a time interval
  - COUNT_MINUTE, MIN_HOUR, MAX_MONTH, and SUM_YEAR

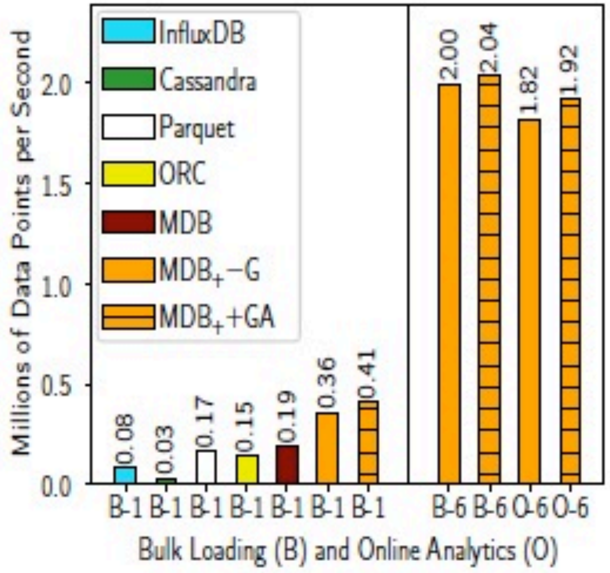


ModelarDB+ Evaluation

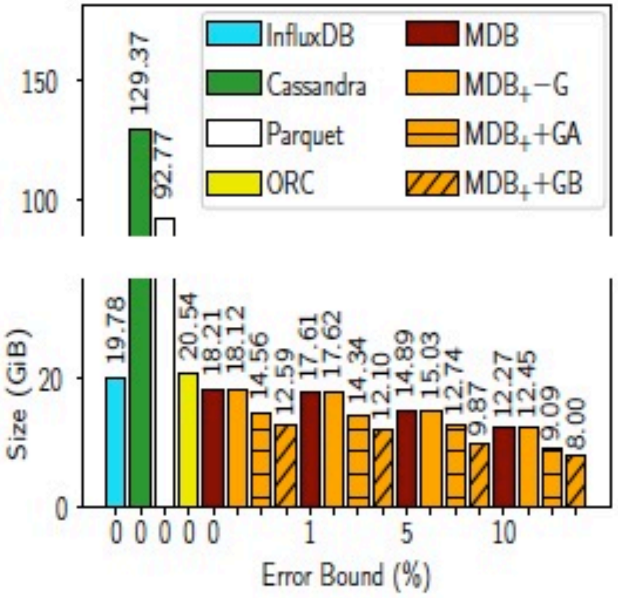
- Evaluation primarily uses **real-life data sets** from the **energy** domain:
  - **EP 45,353 time series** collected from **energy producers** with a sampling interval of **60s** and occupying **339 GiB** as uncompressed **CSV**
  - **EF 197 time series** collected from **wind turbines** with a sampling interval of **200ms** and occupying **372 GiB** as uncompressed **CSV**
- Most experiments are performed on a small cluster of commodity PCs
- Scalability experiments are performed using Microsoft Azure
- ModelarDB+ configurations:
  - with **no** grouping (MDB+ -G)
  - with **auto** (MDB+ +GA)
  - with the **best** (handtuned) primitives per data set (MDB+ +GB)

Ingestion and Storage Results ModelarDB+

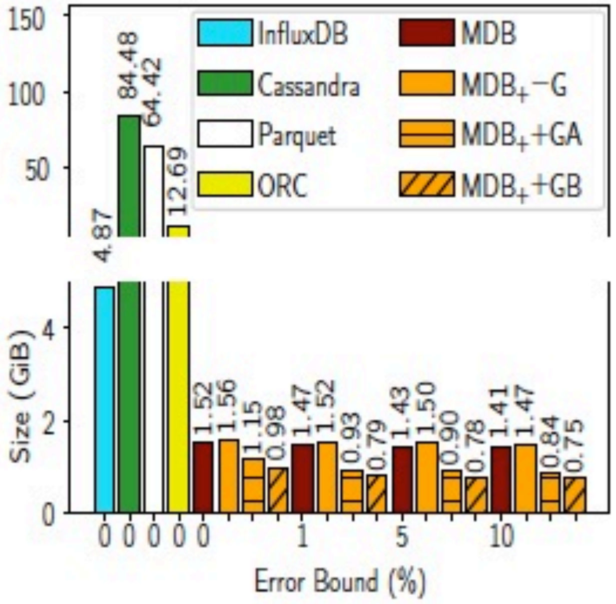
- ModelarDB+ provides both a much **higher ingestion rate** and requires much **less storage than InfluxDB, Cassandra, Parquet, and ORC**
- Using **multiple model types** allows GOLEMM to **automatically adapt**
- **Grouping improves** the **ingestion rate** due to the higher compression
- **Creating groups automatically** from metadata **improves** the **compression**



Ingestion Rate, EP



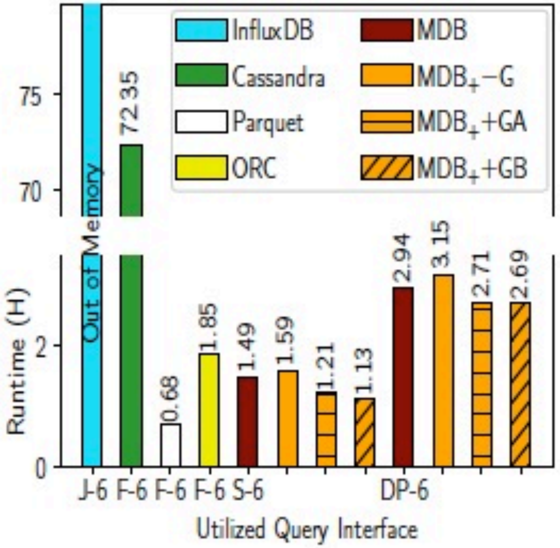
Storage Used, EP



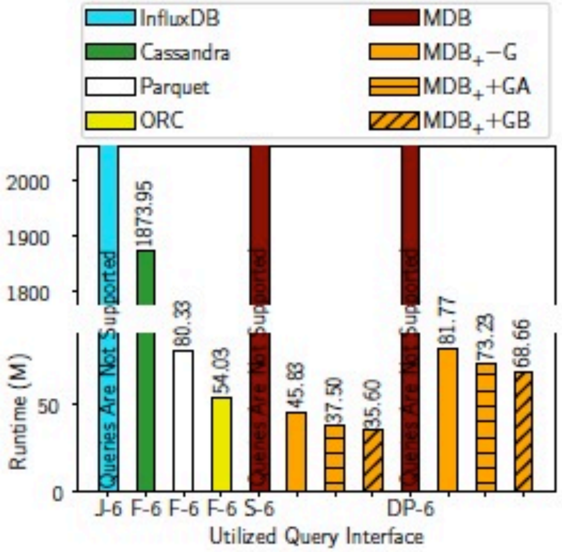
Storage Used, EH

Aggregate Query Results ModelarDB+

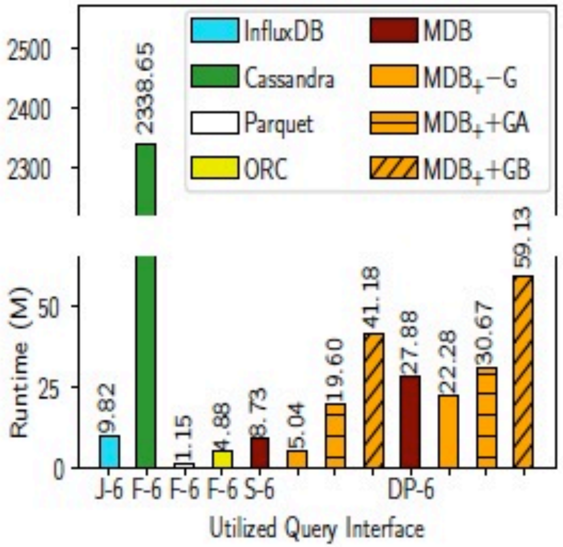
- ModelarDB+ is **faster than** the other formats **when timestamps and values are used**, and experiments on **Azure** show that it **scales linearly**
- **Grouping time series decreases query time** for queries that **use most or all** of the time **series in each group** (Large Scale / Month and Category)
- Grouping **can** also **increase** query time if the query **only use** a **few** time **series from each group** (Small Scale) or if the groups are on one worker



Large Scale, EP



Month and Category, EP



Small Scale, EH

ModelarDB+ Conclusion: where are we now?

- Summary
  - **Wind turbines** produce **huge time series** that should be stored/queried at **high frequency**
  - **Current practice**: storing sensor data as **simple aggregates**, discards valuable information
  - **Grouping time series** and **storing them as models** provides **many benefits** over storing them as simple aggregates or raw data points
  - We proposed methods for creating (Primitives), compressing (the MMGC method GOLEMM), and querying time series groups
  - The evaluation of ModelarDB+ showed that **grouping** can **provide even faster ingestion** speed, **reduced storage** required, and **faster aggregate queries**
- Future Work
  - **Indexing** techniques exploiting that data is stored as models
  - **Query and cluster-aware** grouping and partitioning methods
  - Support for **high-level analytical queries and machine learning directly on models**

MORE project

- **Management Of Real-time Energy data (MORE)**
  - Call topic ICT-51-2020: Big Data technologies and extreme-scale analytics
  - October 2020 - September 2023
  - Athena RC (coordinator), AAU, InAccess, IBM Research Dublin, Perception Dynamics, LABORELEC (ENGIE subsidiary), ModelarData
- ModelarDB concept both for **edge computing** and **cloud**
  - **Optimizing edge storage** and **transfer** to **cloud with models**
- Advanced **time series analytics** and **machine learning directly on** (streaming) **models**
- Main use cases
  - **Massive solar park streams** (Inaccess)
  - **Massive wind park streams** (ENGIE/Laborelec)

MORE

Management of Real-time Energy Data

ModelarData Spinout

- ModelarData spinout established for commercial exploitation and uptake
- Cloudera-like business model
  - Free open source base version
  - Paid premium feature
    - Managed subscription, documentation, training, support, consulting + custom development
- Focus: **extreme-scale analytics** mainly for **renewable energy data**

ModelarData

References

- Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, Time Series Management Systems: A Survey. In *IEEE Transactions on Knowledge and Data Engineering*, Volume 29, Number 11, Pages 2581–2600, November, 2017.
- Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra. In *Proceedings of the VLDB Endowment*, Volume 11, Number 11, Pages 1688–1701, July, 2018.
- Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, Demonstration of ModelarDB: Model-Based Management of Dimensional Time Series. In *Proceedings of the 2019 International Conference on Management of Data*, Pages 1933–1936, 2019.
- Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB+". In the *37th IEEE International Conference on Data Engineering (ICDE),* 2021
- MORE Project: https://more2020.eu
- ModelarData https://modelardata.com