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

Søren Kejser Jensen (Aalborg University)

Torben Bach Pedersen (Aalborg University)

Christian Thomsen (Aalborg University)

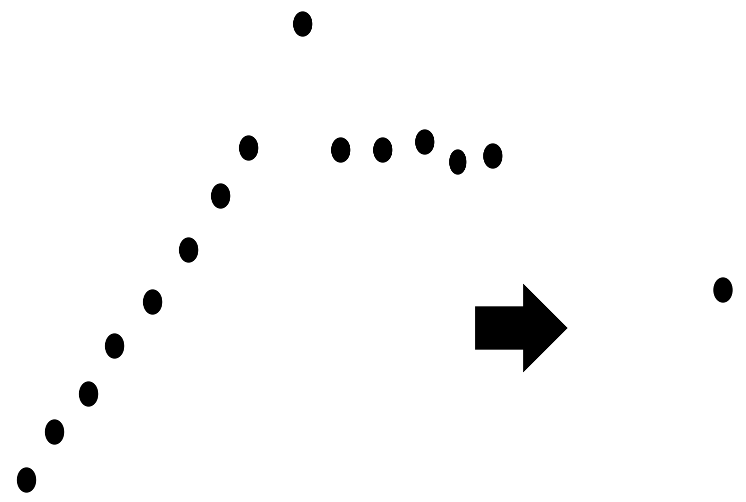
Motivation

- Large industrial systems produce big amounts of **high quality sensor data**
- Data is collected as **regular time series** with **only a few gaps** without values

- High frequency could benefit analysis but:
  - High frequency data requires very fast ingestion
  - 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)
PostgreSQL 10.1	782.87
RDBMS-X - Row	367.89
RDBMS-X - Column	166.83
InfluxDB 1.4.2 - Tags	4.33
InfluxDB 1.4.2 - Measurements	4.33

Storage Method	Size (GiB)
CSV Files	582.68
Apache Parquet Files	106.94
Apache ORC Files	13.50
Apache Cassandra 3.9	111.89
ModelarDB	2.41 - 2.84

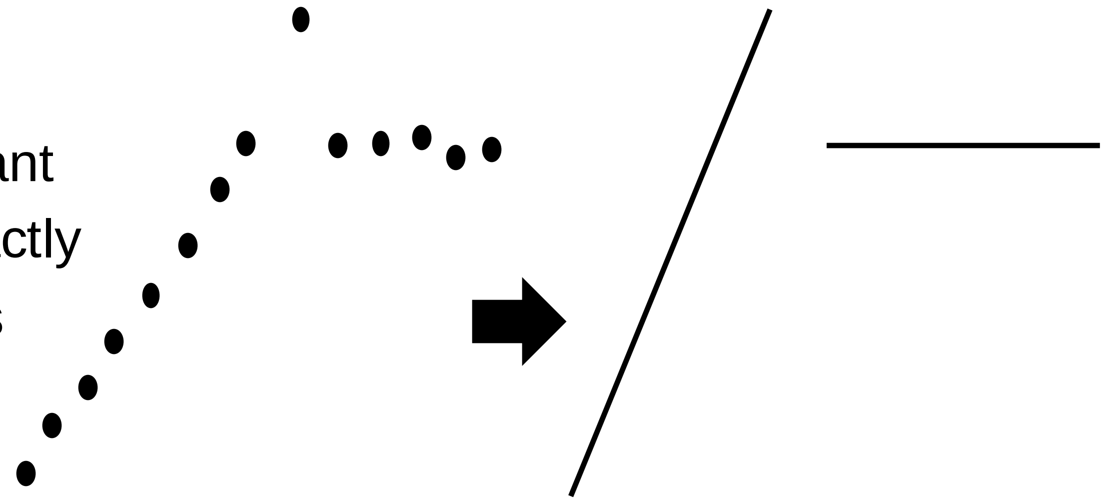


- Currently **simple aggregates are stored** with **outliers and fluctuations lost!**

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 ***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

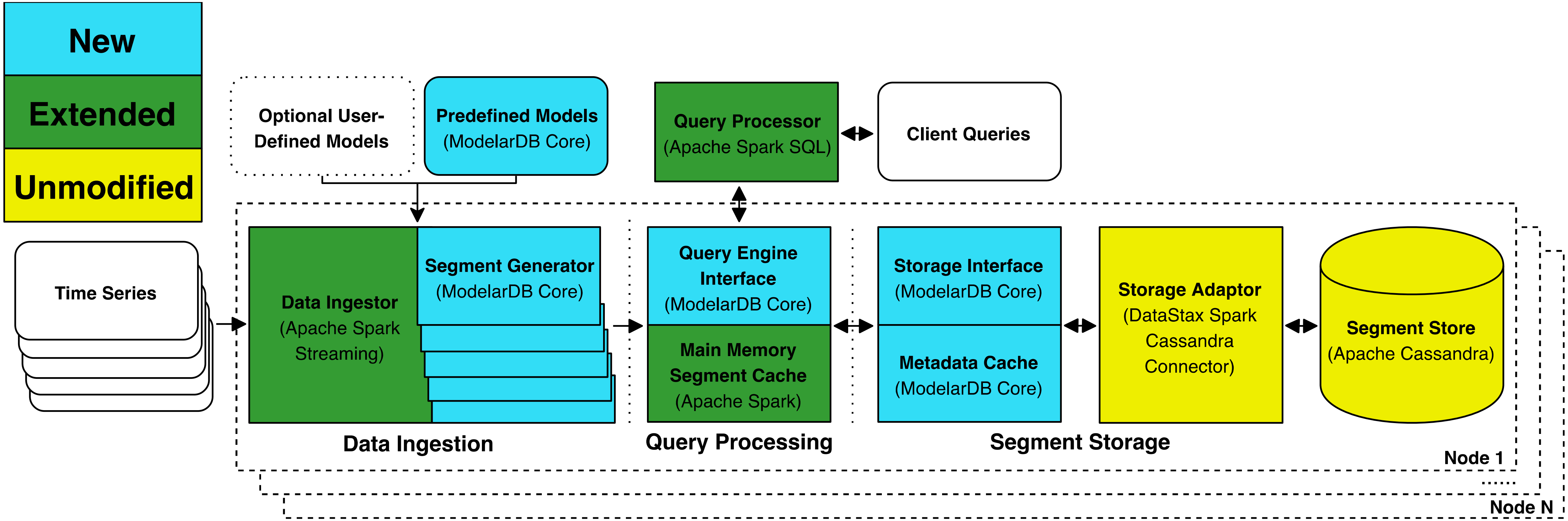


- Problems with model-based storage:
  - The best model for a time series changes over time
  - Long models for high compression increase latency
- Our contributions **remove both of these problems**

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
  - It is model-agnostic, extensible, allows gaps, and 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 addding 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

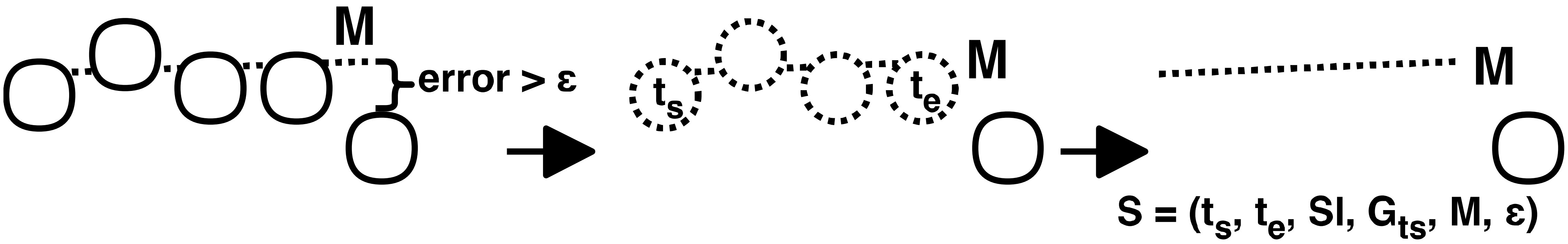
Architecture



- 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

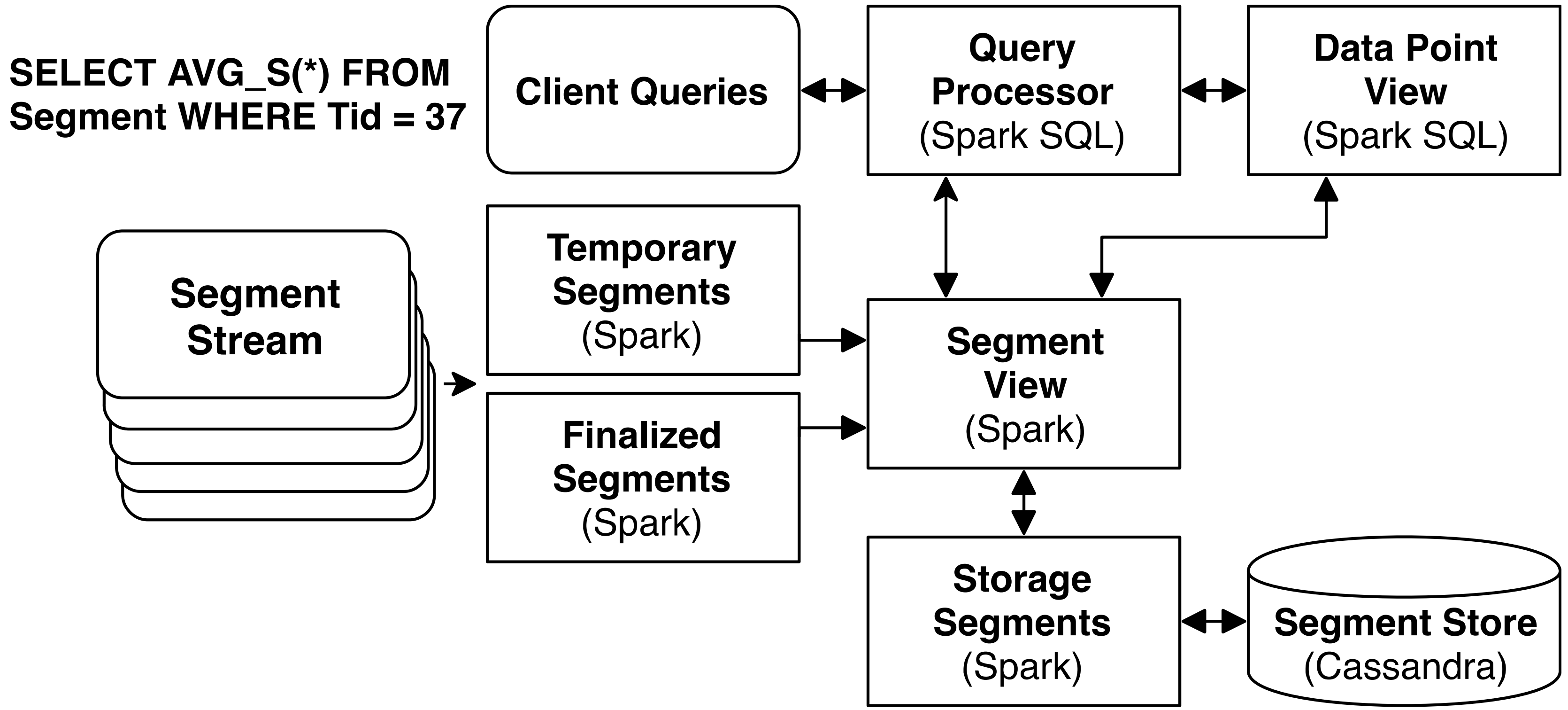
- Models are incrementally fitted and emitted as part of **segments with metadata**:
  - **Temporary Segment:** Holds an unfinished model cached in memory for low latency
  - **Finalized Segment:** Holds a finished model cached in memory and persisted to disk
- Models are fitted in sequence until all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

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



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

- The 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 the Segment View are 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;
}
```

Evaluation - Storage

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

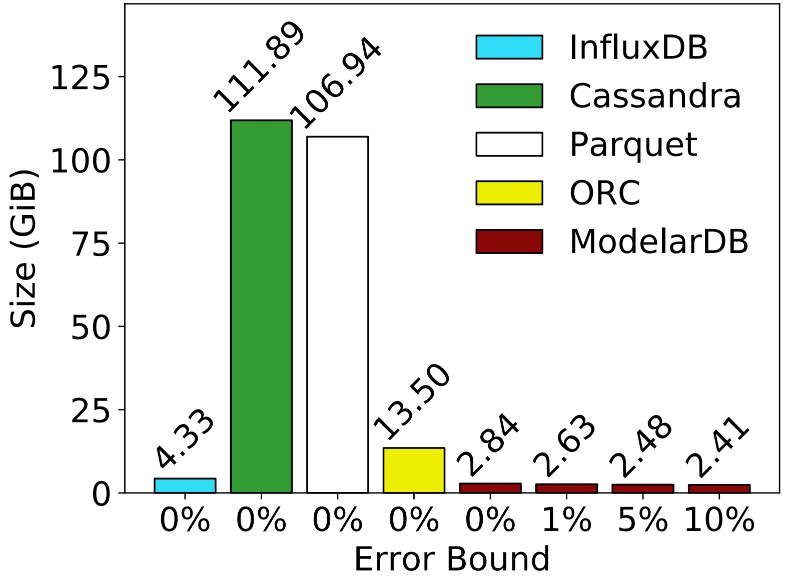


Figure: Storage, EH

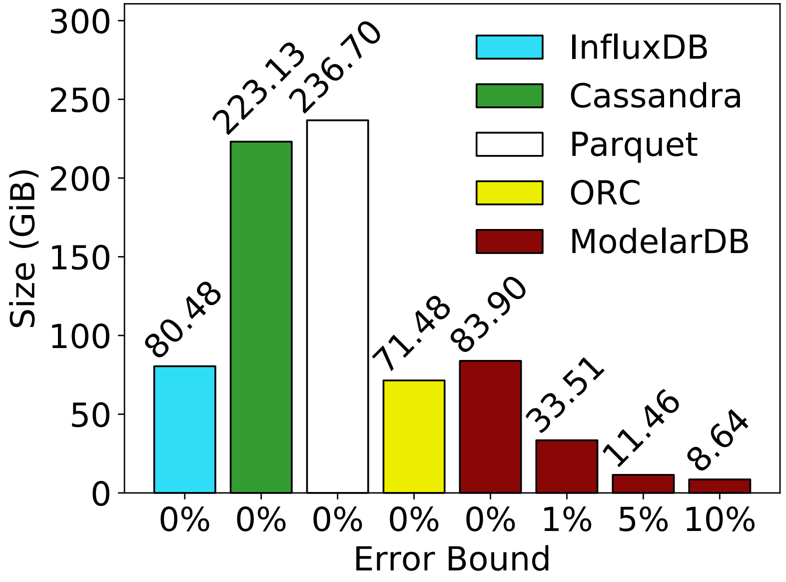


Figure: Storage, ER

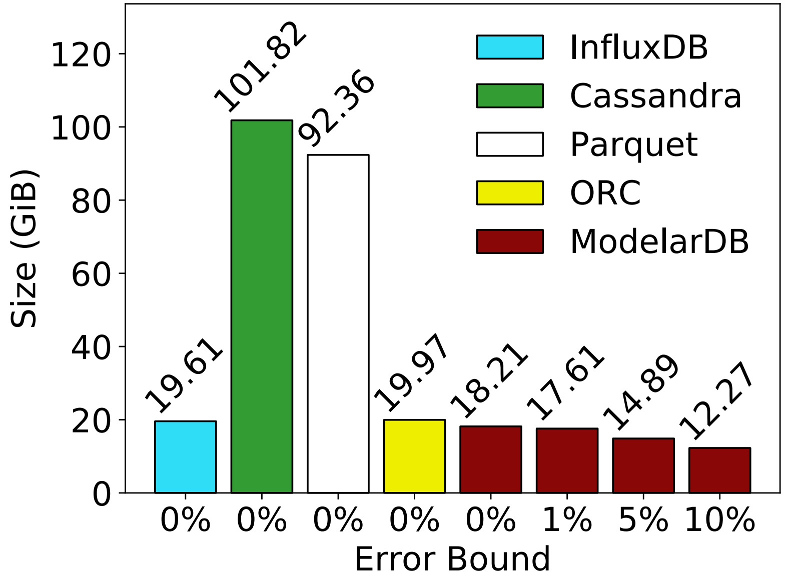


Figure: Storage, EP

- **ModelarDB provides better compression using model-based storage**
  - The best compression ratio is 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
  - The paper shows ModelarDB degrades gracefully as outliers increase in number

Evaluation - Adaptability

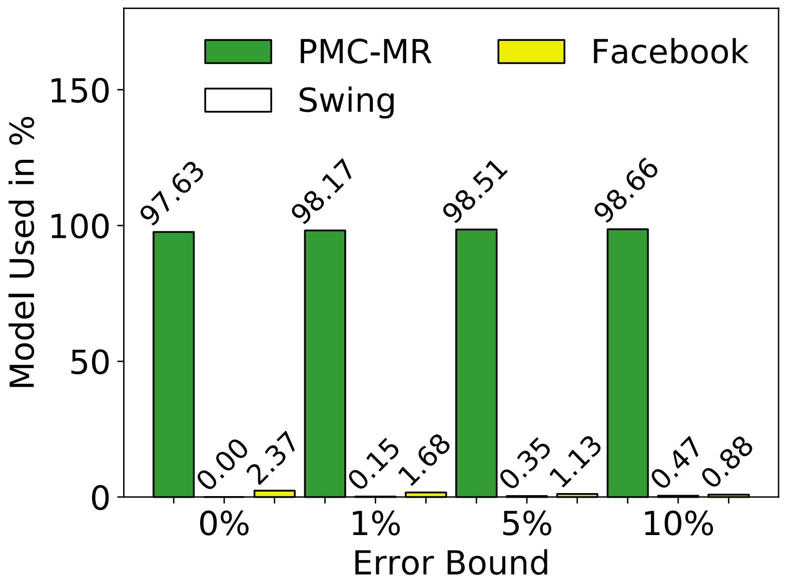


Figure: Adaptability, EH

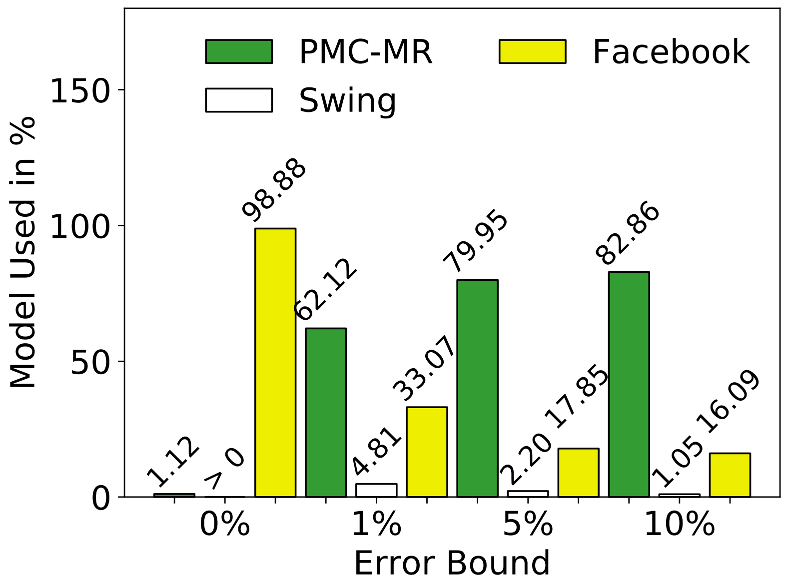


Figure: Adaptability, ER

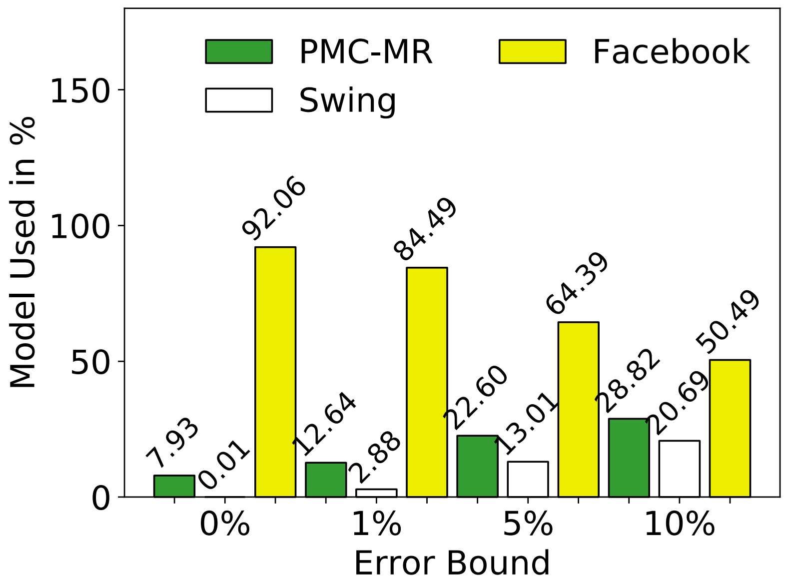


Figure: Adaptability, EP

- **ModelarDB chooses an appropriate model for each part of a series**
  - Different models are 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

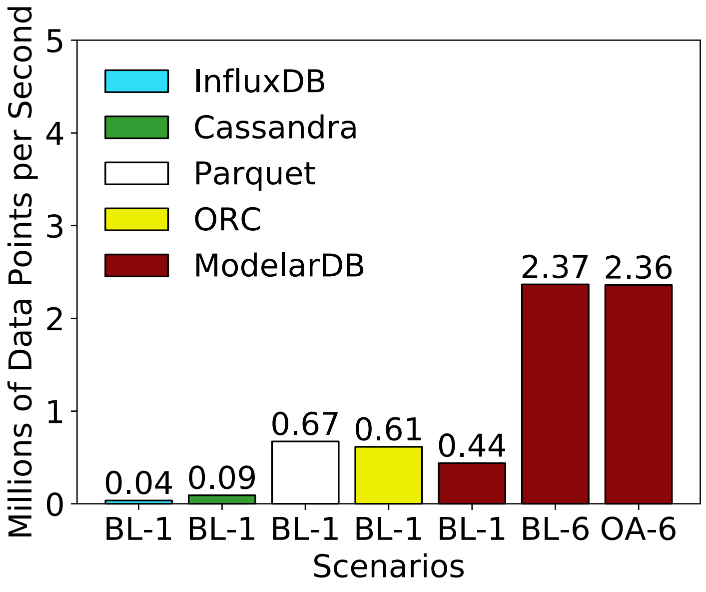


Figure: Ingestion, ER

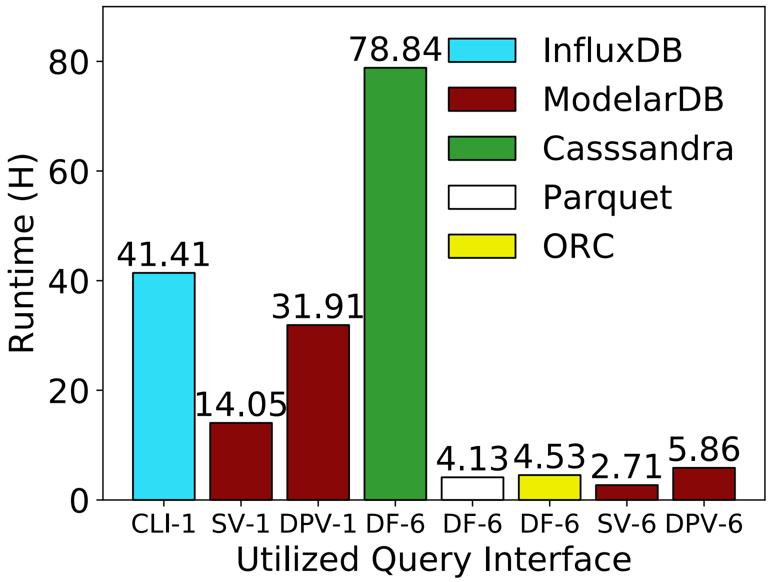


Figure: Aggregate Queries, ER

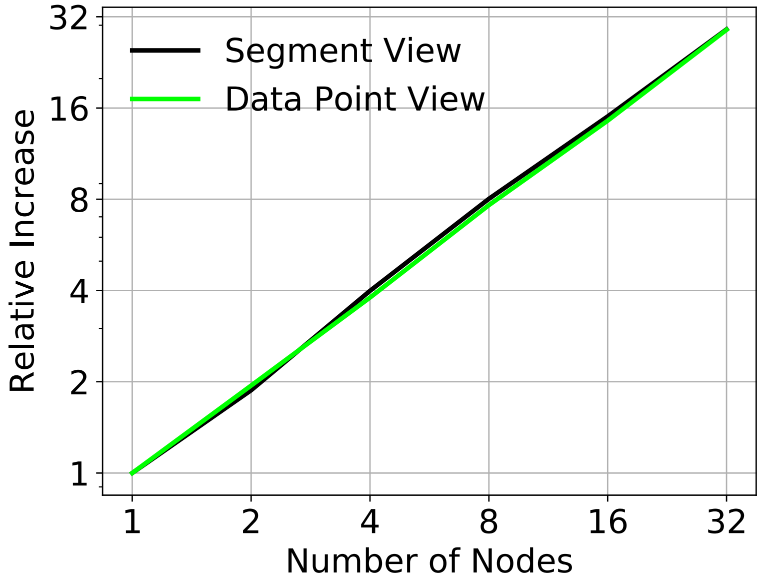


Figure: Scalability (Azure), ER

- **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

Summary and Future Work

- **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**

- **Future Work:**
  - Reduce storage required by storing correlated time series as one stream of segments
  - Increase query performance though new methods that operate on time series as segments