TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

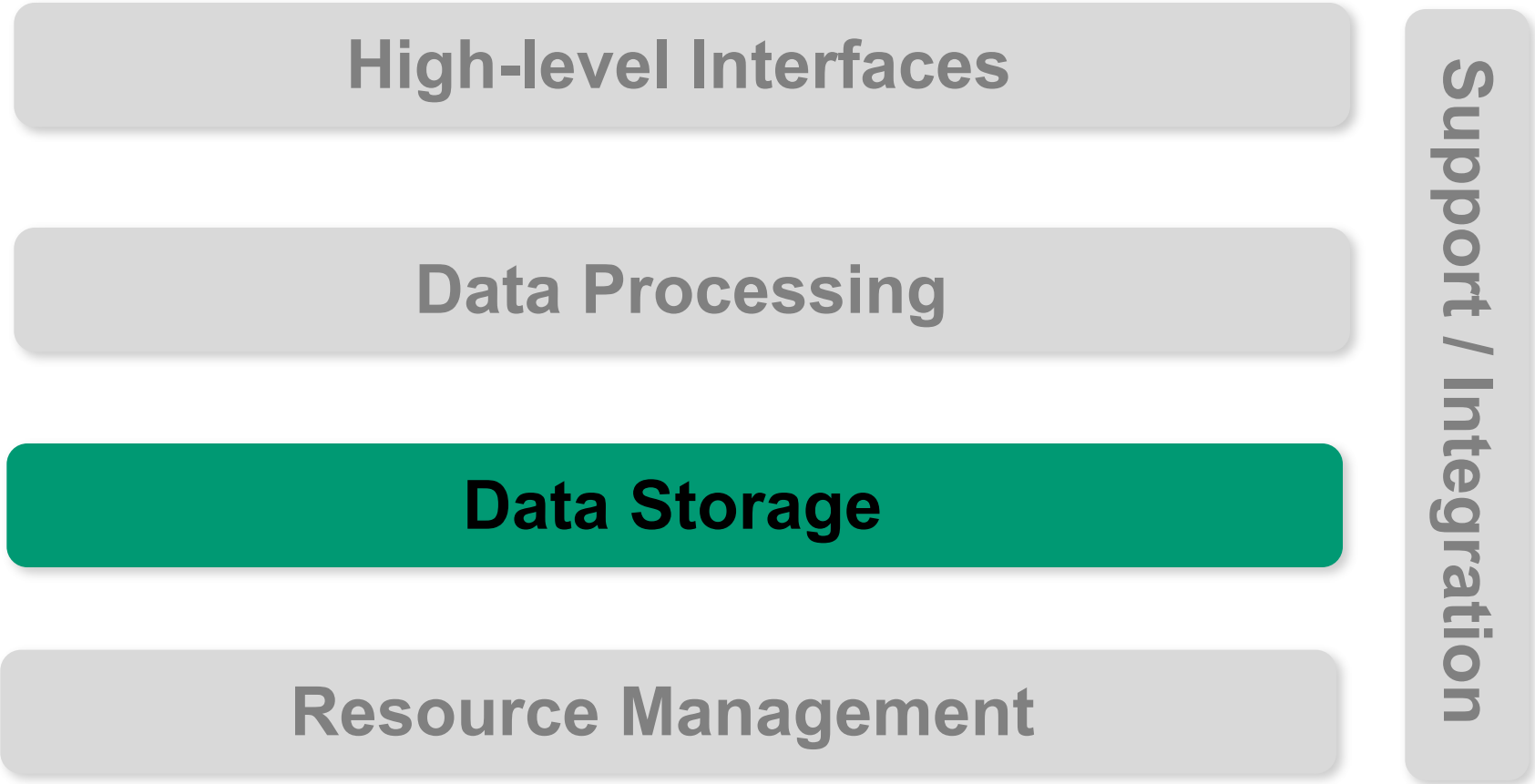
Time Series Database

A.A. 2025/26

Matteo Nardelli

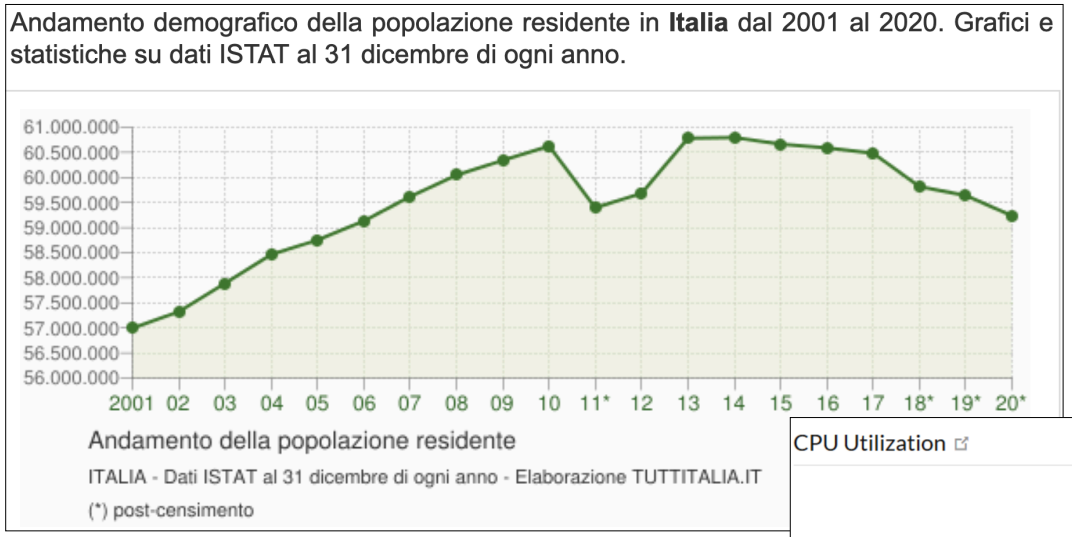
Laurea Magistrale in Ingegneria Informatica - II anno

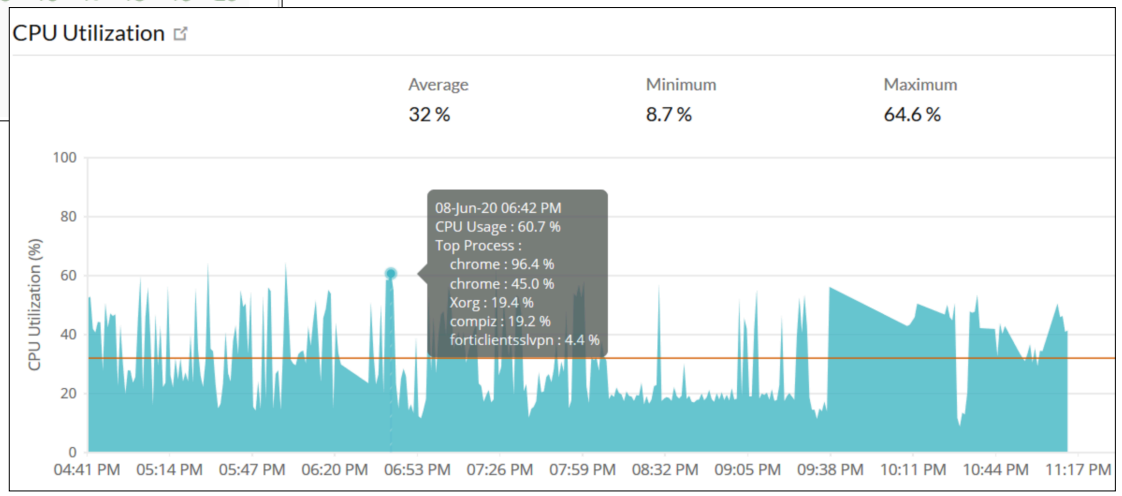
The reference Big Data stack



Time Series

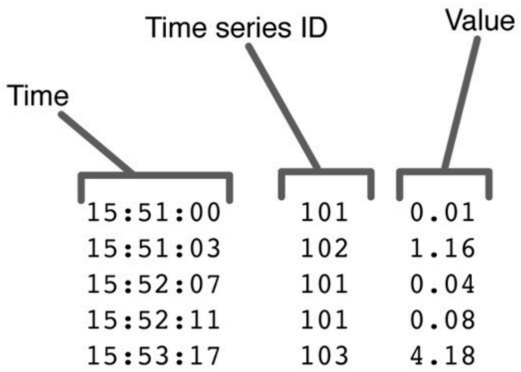
- Time series data are measurements (or events) collected over time
  - Usually taken at equally spaced data points
  - Discrete-time measurements



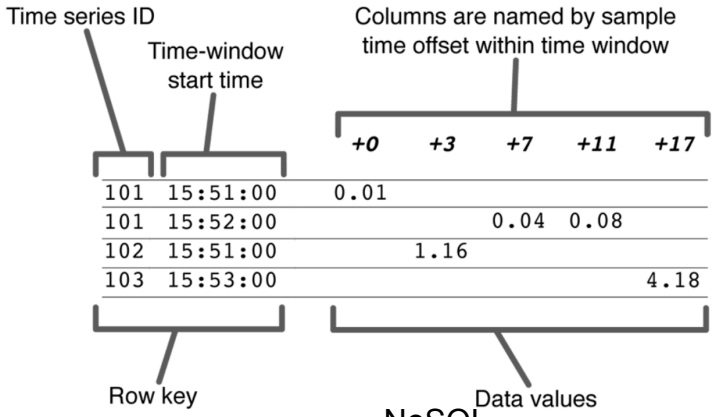


Store Time Series

- Where to store data?
- Flat files
  - Limited utility for time series, data will outgrow them, access is inefficient
- Relational DBMS
  - Possible scalability issues
- NoSQL
  - Better scaling, efficient queries based on time range
  - The design can be challenging:
    - Row key as time series ID, column as time offset ?
    - Wide table stores data point-by-point
    - Hybrid design with wide tables and blob (aggregation of points)



RDBMS



NoSQL

Time Series Database (TSDB)

- Optimized for time series data
  - Key notion: time
  - Database optimized for handling time-stamped data
  - Examples: application performance monitoring, network data, sensor data, events, clicks, trades in a market, and many other types of analytics data.
- Similar to a key/value store:
  - Where the key is a timestamp
  - The value is the measurement, which can have multiple fields.
- A TSBD is optimized for measuring changes over time
  - Considers data lifecycle management
  - Optimizes data storage: storage, compression, data retention, and sharding
  - Optimizes data query: time-aware queries, data aggregation, large range scan of records

Time Series Database (TSDB)

- Ranking of TSDBs on DB-Engines

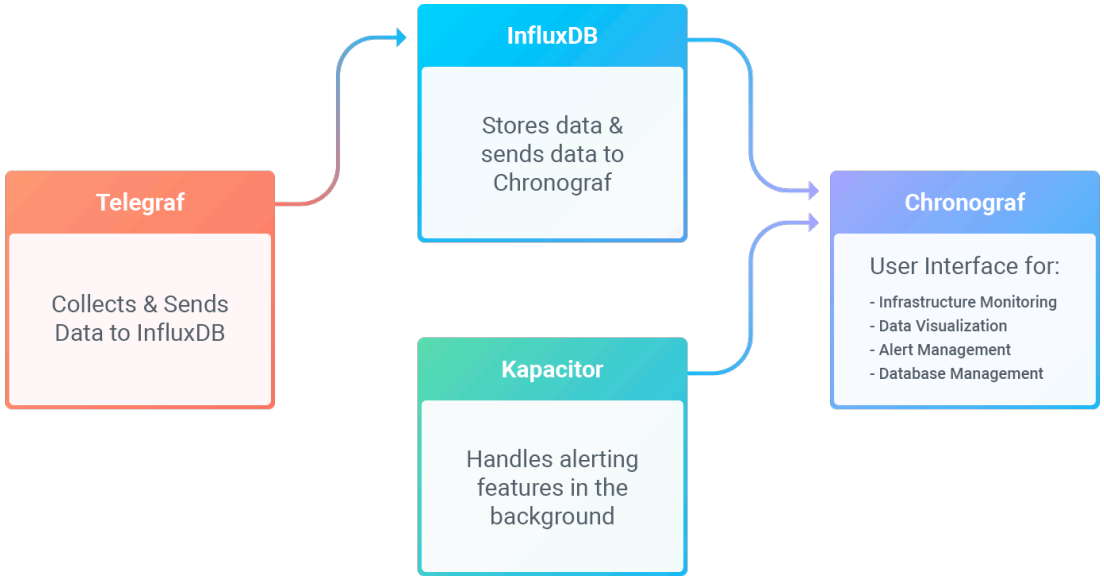
☐ include secondary database models — 45 systems in ranking, April 2026

Rank Apr 2026	Rank Mar 2026	Rank Apr 2025	DBMS	Database Model	Score Apr 2026	Score Mar 2026	Score Apr 2025
1.	1.	1.	InfluxDB	Time Series, Multi-model	20.49	-0.25	-1.05
2.	2.	↑ 3.	Prometheus	Time Series	9.07	+0.36	+2.59
3.	3.	↓ 2.	Kdb	Multi-model	7.60	+0.18	+1.11
4.	4.	↑ 5.	TimescaleDB	Time Series, Multi-model	5.49	+0.07	+1.90
5.	5.	↑ 8.	DolphinDB	Multi-model	4.91	+0.33	+2.71
6.	6.	↓ 4.	Graphite	Time Series	4.56	+0.05	+0.30
7.	7.	↓ 6.	QuestDB	Time Series, Multi-model	3.74	+0.08	+0.62
8.	8.	↓ 7.	Apache Druid	Multi-model	3.46	-0.04	+0.41
9.	9.	↑ 10.	TDengine	Time Series, Multi-model	2.58	+0.31	+0.91
10.	10.	↑ 15.	VictoriaMetrics	Time Series	2.01	+0.17	+0.66
11.	11.	↑ 14.	Apache IoTDB	Time Series	1.68	-0.04	+0.28
12.	12.	↓ 11.	RRDtool	Time Series	1.56	-0.02	+0.01
13.	↑ 15.	↓ 12.	Fauna	Multi-model	1.52	+0.06	+0.07
14.	↓ 13.	↓ 9.	GridDB	Time Series, Multi-model	1.52	+0.01	-0.48
15.	↓ 14.	↓ 13.	OpenTSDB	Time Series	1.46	-0.01	+0.05
16.	16.	16.	Amazon Timestream	Time Series	1.23	-0.03	+0.01
17.	17.		Arc	Time Series	0.98	+0.07	
18.	18.	18.	M3DB	Time Series	0.85	+0.01	+0.00
19.	↑ 20.	19.	CrateDB	Multi-model	0.60	+0.00	-0.04
20.	↓ 19.	20.	KairosDB	Time Series	0.53	-0.06	-0.01

https://db-engines.com/en/ranking/time+series+dbms

InfluxDB

- Natively built to manage time series data
- InfluxDB is part of an ecosystem that supports:
  - Collection: Telegraf
  - Storage: InfluxDB
  - Monitoring/Processing: Kapacitor
  - Visualization: Choronograf
  - Alerting



InfluxDB

- Data model:

```
measurement-name tag-set field-set timestamp
```

  - Measurement: string (indexed)
  - Tag-set: key-value pairs (indexed, only string allowed)
  - Field-set: key-value pairs (values can be of numbers, booleans or strings)
  - Timestamp: can have second, millisecond, microsecond, or nanosecond precision

Example:

```
cpu host=serverA,region=uswest idle=23,user=42,system=12 1464623548s
```

- Data compression depends on the level of precision
- Data storage (on disk):
  - Data is organized in a columnar style format
  - Contiguous blocks of time are set for the measurement, tagset, field.
  - Each field is organized sequentially on disk for blocks of time, which make calculating aggregates on a single field a very fast operation.
  - There is no limit to the number of tags and fields that can be used.

InfluxDB

- InfluxDB creates a shard for each block of time, whose size depends on the retention policy:
  - describes how long to keep data (retention)
  - how many copies (replication factor)
  - the time range covered by shard groups (shard group duration)

- Each shard maps to an underlying database with
  - A **WAL** file:
    - Write-optimized storage format (durable writes, but not easily queryable)
    - Writes to the WAL are appended to segments of a fixed size.
  - A **TSM** file:
    - Contains sorted, compressed series data.
    - Read-only files that are memory mapped.

InfluxDB: Query and Manipulation Language

- Flux is InfluxData's functional data scripting language designed for querying, analyzing, and acting on data

- Flux supports multiple data source types, including:
  - Time series databases (such as InfluxDB)
  - Relational SQL databases (such as MySQL and PostgreSQL)
  - CSV

- Flux unifies code for querying, processing, writing, and acting on data into a single syntax.

InfluxDB: Flux

Like treating water, a Flux query does the following:

1. Retrieves a specified amount of data from a source.
2. Filters data based on time or column values.
3. Processes and shapes data into expected results.
4. Returns the result.

```
from(bucket: "example-bucket")           // —— Source
  |> range(start: -1d)                   // —— Filter on time
  |> filter(fn: (r) => r._field == "foo") // —— Filter on column values
  |> group(columns: ["sensorID"])        // —— Shape
  |> mean()                              // —— Process
```

InfluxDB: Flux

```
from(bucket: "example-bucket")           // —— Source
  |> range(start: -1d)                   // —— Filter on time
  |> filter(fn: (r) => r._field == "foo") // —— Filter on column values
  |> group(columns: ["sensorID"])        // —— Shape
  |> mean()                              // —— Process
```

- from() to retrieve data from the data source.
- pipe-forward operator (|>) to send the output of each function to the next function as input.
- range(), filter(), or both to filter data based on column values.
- mean() to calculate the average of values returned from the data source.
- yield() to yield results to the user.

InfluxDB: The Flux Data Model

The Flux data model comprises the following:

- Stream of tables: returned by data sources
- Table: collection of columns partitioned by group key
- Column: collection of values of the same basic type that contains one value for each row.
- Row: a collection of associated column values.
- Group key: key-value pairs, where each key represents a column name and each value represents the column value included in the table.
  - All rows in a table contain the same values in group keys. All tables in a stream of tables have a unique group key
  - e.g., each group key represents a table containing data for a unique location

```
[_measurement: "production", facility: "us-midwest", _field: "apq"]
[_measurement: "production", facility: "eu-central", _field: "apq"]
[_measurement: "production", facility: "ap-east", _field: "apq"]
```

https://docs.influxdata.com/flux/v0.x/get-started/data-model

InfluxDB: Flux

- The majority of basic Flux queries include the following steps: Source, Filter, Shape, Process
- **Source**:
  - Flux input functions retrieve data from a data source.
  - All input functions return a stream of tables.
  - Flux supports multiple data sources including, time series databases (such as InfluxDB and Prometheus), relational databases (such as MySQL and PostgreSQL), CSV, and more.
- **Filter**:
  - Filter functions iterate over and evaluate each input row to see if it matches specified conditions.
  - range(): filter data based on time.
  - filter(): use a predicate function (fn) to filter data based on column values.

https://docs.influxdata.com/flux/v0.x/function-types/

https://docs.influxdata.com/flux/v0.x/get-started/syntax-basics/

InfluxDB: Flux

- **Shape** data:
  - Queries may require to change the structure of data
  - Functions that reshape data include:
    - group(): modify group keys
    - window(): modify _start and _stop values of rows to group data by time
    - pivot(): pivot column values into rows
    - drop(): drop specific columns
    - keep(): keep scientific columns and drop all others
- **Process**:
  - Aggregate data: into a single row (e.g., count(), mean(), sum(), quantile())
  - Select specific data points: return specific rows from each input table.
    - e.g., distinct(), first(), last(), min(), max(), limit(), top(), unique()
  - Rewrite rows: transform values (e.g., maths operations, process strings, add new columns)
    - e.g., map(), which by default drops not explicitly mapped columns, the with operator updates a column if it already exists and includes all existing columns:
      ```
      map(fn: (r) => ({ r with newColumn: r._value * 2 }))
      ```
  - Send notifications

Hands-on InfluxDB
(Docker image)

InfluxDB with Docker

- We use the official standalone InfluxDB image

```
$ docker pull influxdb:2.7
```

- We can now create an instance of InfluxDB

```
$ docker run -p 8086:8086 \
      -v $PWD:/var/lib/influxdb2 \
    influxdb:2.7
```

- Chronograf can be reached at http://localhost:8086

Loading Sample Data

```
import "influxdata/influxdb/sample"

sample.data(set: "airSensor")
    |> to(
        org: "example-org",
        bucket: "example-bucket"
    )
```

_time	_value	_field	_measurement	sensor_id
2022-04-09 17:05:49 GMT+2	0.5108571955231641	co	airSensors	TLM0100
2022-04-09 17:05:59 GMT+2	0.5164828607775828	co	airSensors	TLM0100
2022-04-09 17:06:09 GMT+2	0.5086872079247073	co	airSensors	TLM0100
2022-04-09 17:06:19 GMT+2	0.5019140187838028	co	airSensors	TLM0100
2022-04-09 17:06:29 GMT+2	0.5061828461616715	co	airSensors	TLM0100
2022-04-09 17:06:39 GMT+2	0.5156546512843109	co	airSensors	TLM0100
2022-04-09 17:06:49 GMT+2	0.5149306168772608	co	airSensors	TLM0100
2022-04-09 17:06:59 GMT+2	0.5254830148241758	co	airSensors	TLM0100
2022-04-09 17:07:09 GMT+2	0.519322216384614	co	airSensors	TLM0100

Sample Data: Air Sensors

- The air sensor sample data represents an "Internet of Things" use case;
- It simulates temperature, humidity, and carbon monoxide levels for multiple rooms in a building:
  - Multiple sensors (having ID starting with TLM0*)
  - Each collecting CO, humidity, and

Data structure:

```
_time, _value, _field, _measurement, sensor_id
```

example:

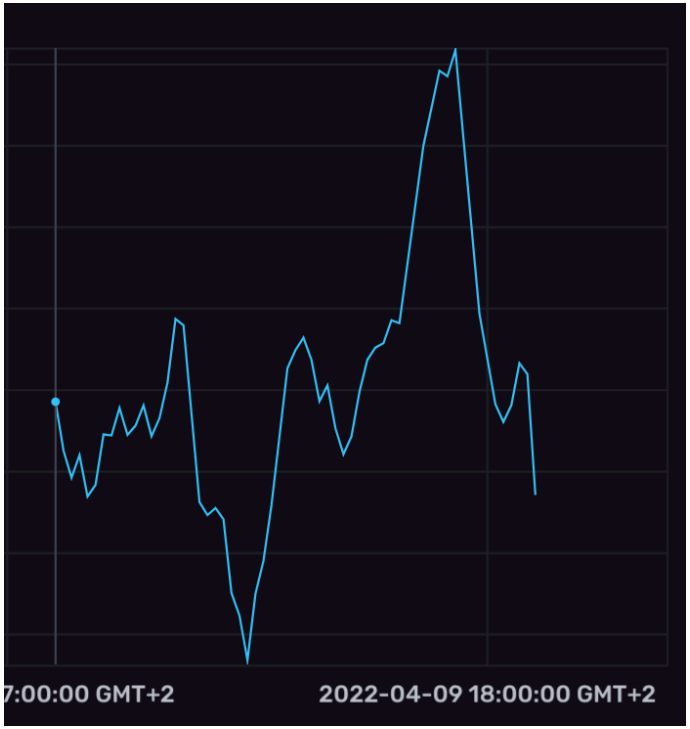
```
2025-04-11 02:59:01, 71.16, temperature, airSensors,
TLM0100
```

Get Raw Temperature Data

```
from(bucket: "example-bucket")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "airSensors")
|> filter(fn: (r) => r["_field"] == "temperature")
|> yield(name: "raw values")
```

Get Sensor's Temperature Data

```
from(bucket: "example-bucket")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "airSensors")
|> filter(fn: (r) => r["_field"] == “temperature")
|> filter(fn: (r) => r["sensor_id"] == "TLM0100")
|> yield(name: "raw values")
```



Get Average Sensor’s Temperature Data

```
from(bucket: "example-bucket")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "airSensors")
|> filter(fn: (r) => r["_field"] == “temperature")
|> filter(fn: (r) => r["sensor_id"] == “TLM0100”)
|> mean()
```

_start	_stop	_value
2022-04-09 12:26:52 GMT+2	2022-04-09 18:26:52 GMT+2	71.20310898758906

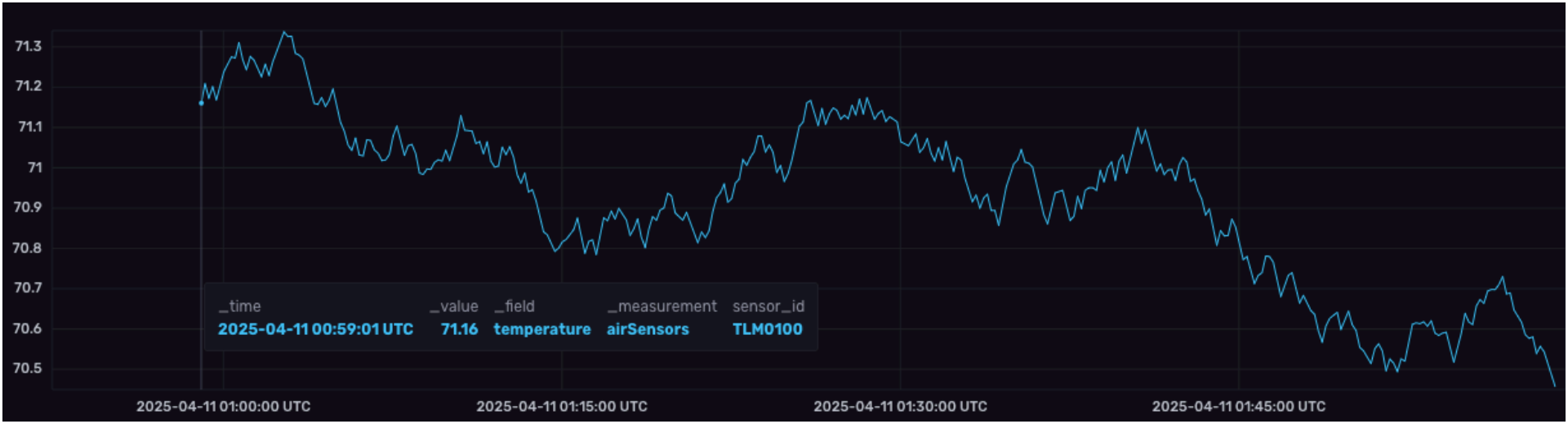
Get the Derivative of Temperature Data

- We first aggregate values in a windows of 10 minutes;
- Then, we compute the derivative using 10 minutes as unit (recall the definition of derivative)

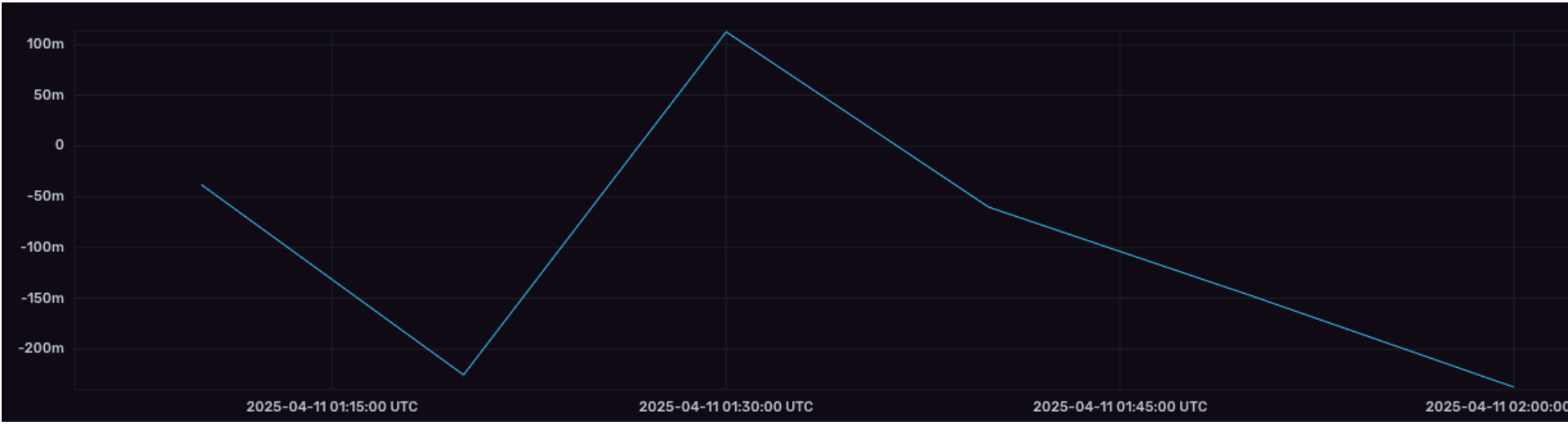
```
from(bucket: "example-bucket")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "airSensors")
|> filter(fn: (r) => r["_field"] == “temperature")
|> filter(fn: (r) => r["sensor_id"] == “TLM0100")
|> aggregateWindow(every: 10m, fn: mean, createEmpty: false)
|> derivative(unit: 10m, nonNegative: false)
|> yield(name: "derivative")
```

Get the Derivative of Temperature Data

- Raw data before the derivative



- Result of derivative operation



InfluxDB 3: From TSDB to Analytics Engine

- Shift from **specialized TSBD** to **columnar analytics engine with SQL**
- **Architecture**:
  - Columnar storage (Parquet): efficient compression, fast scans on large datasets
  - In-memory processing (Apache Arrow): vectorized processing, high-performance analytics
  - SQL query engine (DataFusion): decoupled storage and processing, enabling better integration with object storage (e.g., AWS S3)
  - Overall, scales better for large, historical, analytical workloads
- Why **SQL**?
  - Standard, widely known language;
  - Supports: join, window functions, complex analytics
- **Trade-off**:
  - Fewer built-in time series-related functions
  - But more flexibility (e.g., while influxDB 2 let you analyze one time series at a time, with InfluxDB3 you can relate multiple time series together)

https://docs.influxdata.com/influxdb3/core/reference/