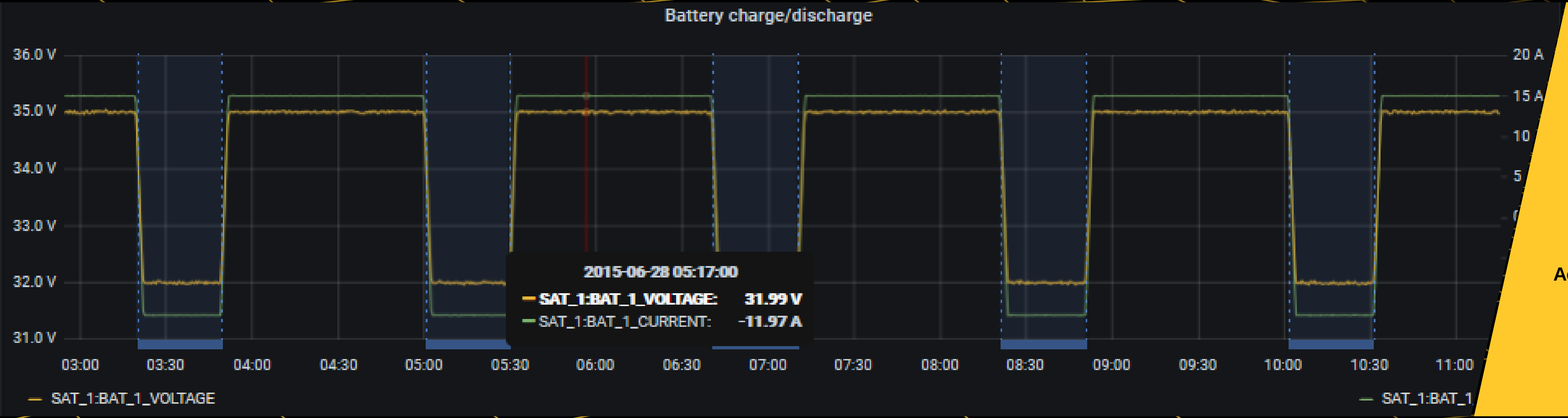
How to Keep all Telemetry from Your Constellation in One Basket

and not spend a fortune on it



1 February 2021

Petro Kazmirchuk, Rüdiger Gad

GSAW 2021

TERMA
ALLIES IN INNOVATION

Our challenges

- Spacecraft data archives are growing very fast in size and complexity
- Advent of satellite (mega) constellations multiplies this growth
  – insight from multiple satellites' data at once?
- Solutions might reflect data structure
  – but more often - team structure
  – federated solutions are especially complex
- Integrating and updating dependencies becomes a burden
  – commercial offers like Cloudera might help here
- Investments in such storage need to be justified
  → more projects and missions share one setup
  → more investment to accommodate size and varying needs
- Hardware requirements are very high
  – usually require a computing cluster
  – difficult to run on a laptop

Can we build a simple and generic tool that would cover most of use cases?

STAT Design Goals

- STAT – a data archiving and analysis tool for Terma Ground Segment Suite (TGSS)
- Core SQL model
  – unified access to data of various nature
  – common language for different engineers
  – public API through SQL views and functions
- Choose your own analytics tool
  – Jupyter notebooks, Grafana, Excel, Tableau etc
  – Grafana as a baseline
- Data sources:
  – CCS5: spacecraft checkout and mission control system
  – ORBIT: flight dynamics system
  – 3rd party software

STAT Design Goals

- Primary focus on numerical and discrete housekeeping TM
- Support both AIT and OPS

- Minimal dependencies; support Windows & Linux

Technology Stack

- PostgreSQL
  - Schemas help with separating public API from implementation details
  - Foreign-data wrapper (FDW) for importing data
  - Performance bottlenecks can be found early and easily with EXPLAIN
- TimescaleDB for time series
  - data stored in “hypertables”
  - same standard SQL, JOINs with other PostgreSQL tables
  - transparent compression
  - continuous aggregates
  - open-source, permissible license
  - excellent user manual
- Python
  - Dash & Plotly for highly customizable WebUI
  - Flask for the Web server and RESTful API
  - “wheel” installer for PIP

Workflow

ORBIT
(AoS, LoS etc)

CCS5 DB

TM, events, alarms

STAT DB

Web UI
REST
SQL
Python

MySQL or PostgreSQL

Optimised for CCS5 efficiency

One DB per session

Not convenient for data analytics

Full time range

Whole fleet

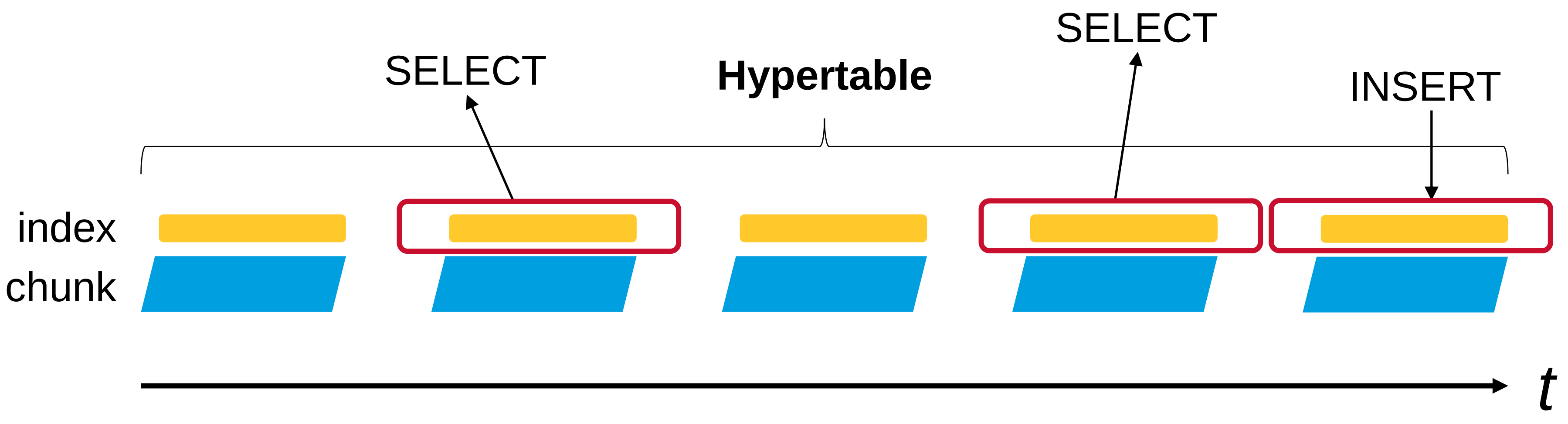
Efficient data analytics

Original frequency → Downsampling Level 1 → Downsampling Level 2 → …

Hypertables

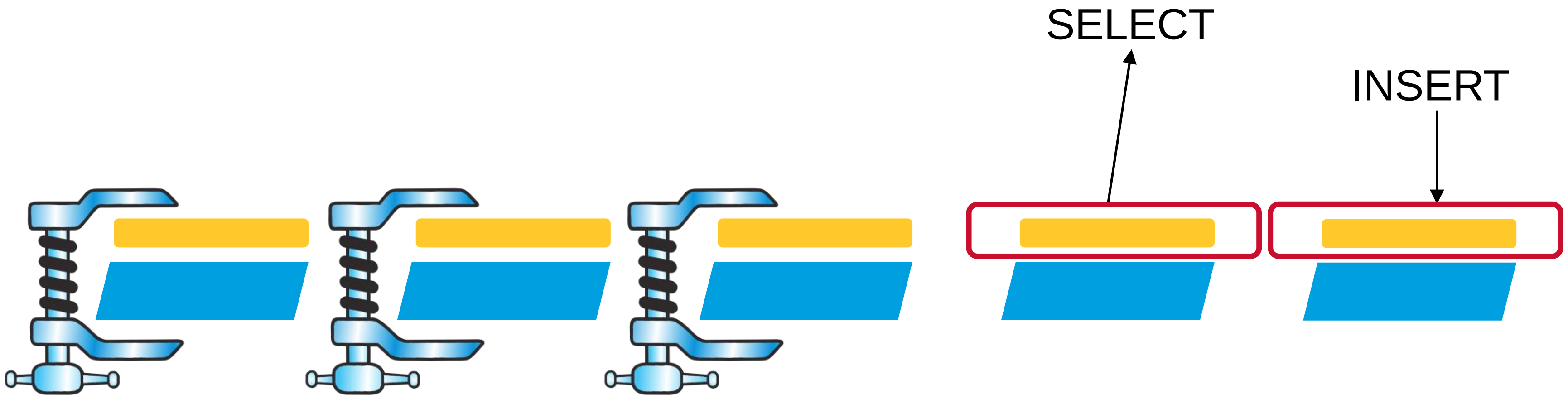
- “Hypertable” looks like a normal SQL table with a timestamp column
- In fact, it is an abstraction over a set of “chunks”
- Each chunk is a normal PostgreSQL table and corresponds to a configurable time interval
- TimescaleDB transparently distributes all queries depending on a requested time range
- Only active indexes need to be loaded in RAM

→ *performance boost*



Compression

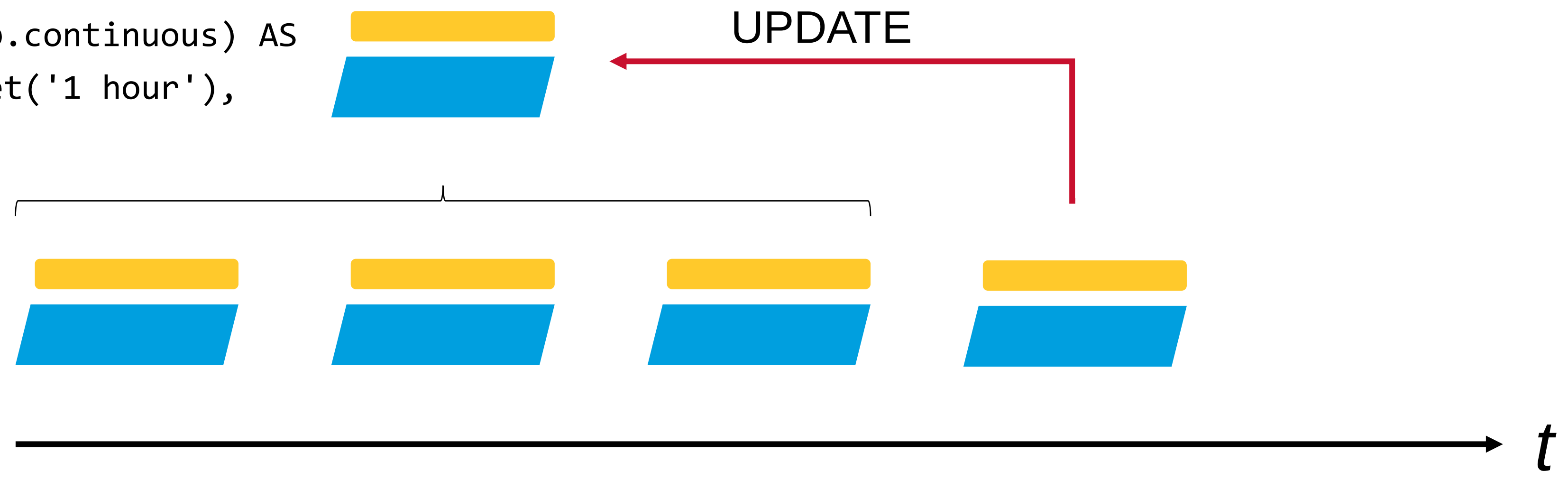
- "Cold" chunks are compressed automatically according to a specified policy
- When a SELECT query arrives, the chunk is decompressed on the fly
- INSERT/UPDATE queries are forbidden
- Compression algorithm depends on a column's data type:
  - – Gorilla compression for floats
  - – Delta-of-delta + Simple-8b for timestamps and other integer-like types
  - – Dictionary compression for columns with discrete values
  - – LZ-based array compression for all other types



Continuous Aggregates

- Materialised views that are updated incrementally in background
- Downsampled telemetry
  - efficient SELECTs at any level
  - numerical parameters: compute averages
  - alarms or discrete parameters: propagate depending on severity

```
CREATE MATERIALIZED VIEW ...
WITH (timescaledb.continuous) AS
SELECT time_bucket('1 hour'),
AVG(value)...
```



Continuous Aggregates - limitations

- Not possible to create CAGG on top of another CAGG
- Not possible to compress CAGG
- Not possible to specify chunk time interval
- Upcoming PostgreSQL 14 might have native support for incremental updates of materialised views:
  `CREATE [ INCREMENTAL ] MATERIALIZED VIEW`
- TimescaleDB 2.0 added Actions API that allows creating your own background jobs written in PL/pgSQL

Our dataset

- Each data point in the telemetry time series has:
  - timestamp (on-board time, microsecond precision)
  - raw value (as extracted from a TM packet)
  - engineering value (after calibration)
  - monitoring state (PostgreSQL enum: `alarm, warning, event, expired, ok` etc)
  - parameter ID
  - satellite ID
- Index with key fields: satellite ID, parameter ID, timestamp
- Source data: 28 parameters from 4 satellites sampled every 10 seconds over 6 years
  - 1.7 billion rows
  - uncompressed: 171 GB (including the index)
  - compressed: 17 GB → 90% compression rate
- Downsampled level #1: 30 min intervals
  - 9.6 million rows, 772 MB
- Downsampled level #2: 3h intervals
  - 1.6 million rows, 129 MB

Tips and tricks: minimizing data size

- Order of columns matters! Avoid or minimize padding

Row header		24 bytes
Timestamp	timestamptz	8 bytes
Raw value	double	8 bytes
Eng value	double	8 bytes
Param ID	smallint	2 bytes
padding		2 bytes
Mon state	enum	4 bytes
Sat ID	smallint	2 bytes
Total		**58 bytes**

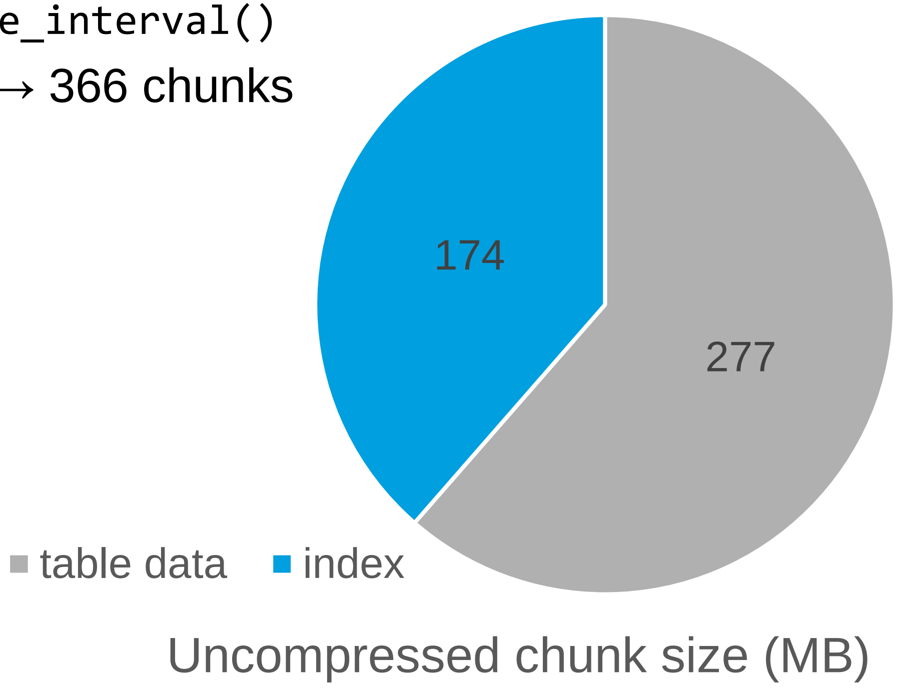
Tips and tricks: minimizing data size

- Order of columns matters! Avoid or minimize padding
  - list your columns from bigger to smaller data type
  - check your row size with `pg_column_size`
  - check alignment of types with `pg_type.typalign`
- Use the smallest type that serves your needs
- Postgres `ARRAY` type has 24-byte header too
- Don’t create unnecessary indexes
  - remember that having the `PRIMARY KEY` constraint implicitly creates an index

Row header		24 bytes
Timestamp	`timestamptz`	8 bytes
Raw value	`double`	8 bytes
Eng value	`double`	8 bytes
Monitoring state	`enum (oid)`	4 bytes
Parameter ID	`smallint`	2 bytes
Satellite ID	`smallint`	2 bytes
Total		**56 bytes**

Tips and tricks: choosing the right chunk interval

- General rule: active chunks (+ indexes) must fit into RAM
  - if a chunk doesn't fit, it will push other cached chunks out of memory
  - if chunks are too small, their total number will be very high → degraded query planning performance
- Not possible to specify chunk size instead of interval
  - assumes steady inflow of data
  - if it becomes a problem, it can be mitigated with `set_chunk_time_interval()`
- In our example dataset we configure the interval to 6 days → 366 chunks
  - 48 MB when compressed
  - 451 MB uncompressed



Tips and tricks: automatic switching between downsampling levels

- Write a SELECT for each table and combine them with UNION ALL
- In each SELECT add a WHERE condition that is true only for a table from which we want to select

```
SELECT time_bucket(bucket, time), avg(value)
FROM source_data
WHERE bucket < '30m'::interval
GROUP BY 1
UNION ALL
SELECT time_bucket(bucket, time), avg(value)
FROM downsample_1
WHERE bucket >= '30m'::interval AND bucket < '3h'::interval
GROUP BY 1
UNION ALL ...
```

→ less work for DB

→ less data to display

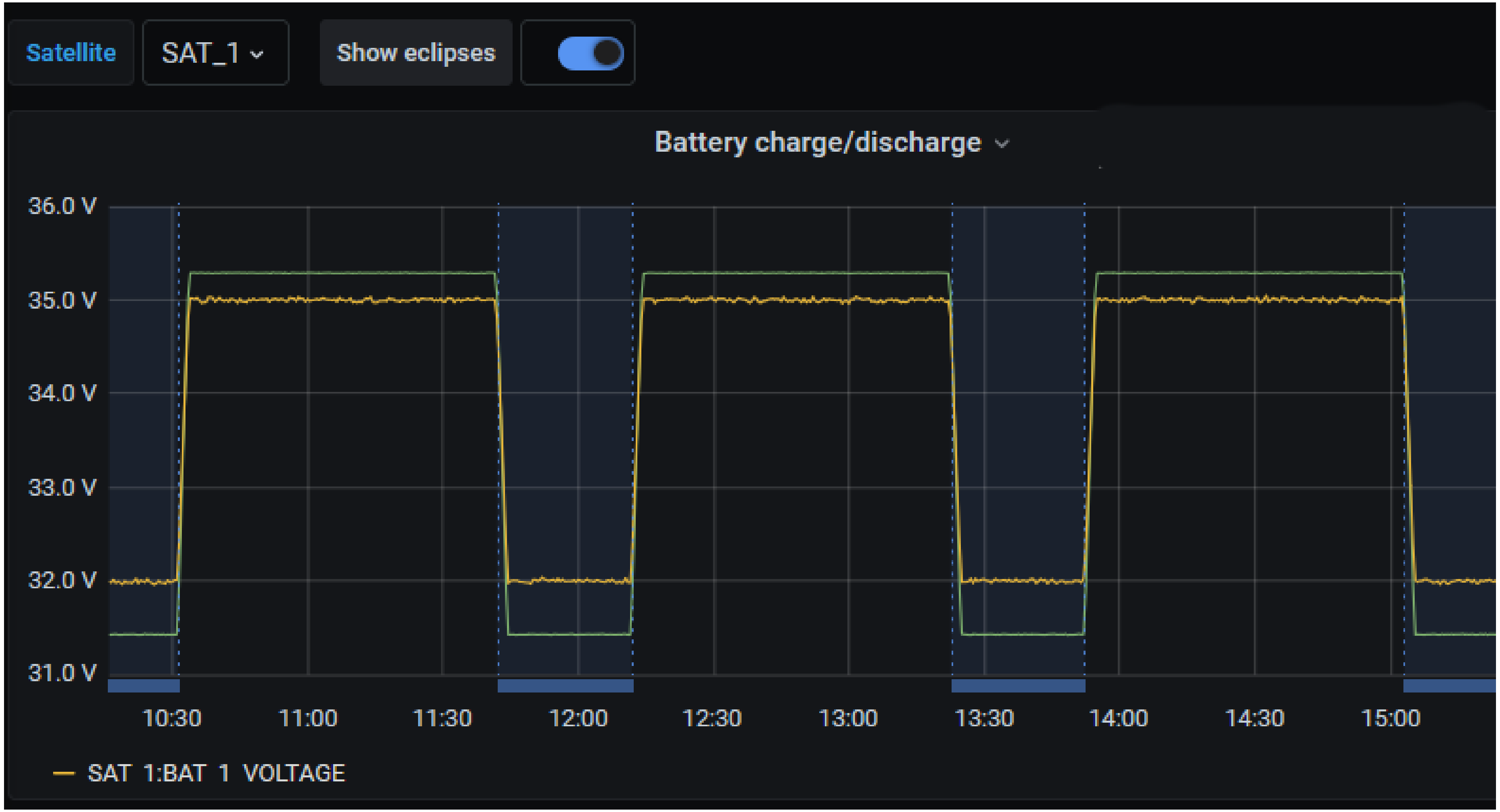
Tips and tricks: built-in variables in Grafana

- Grafana has a PostgreSQL data source with a special option to enable TimescaleDB features
- `$__timeFrom()` and `$__timeTo()` – time range of your graph
- `$__interval` – time interval for one point on the graph (our `bucket` argument)

```
SELECT * FROM stat_plot_raw_tm(
  '$__interval',
  $__timeFrom(),
  $__timeTo(),
  '$system_name',
  'BAT_1_TEMP'
);
```

Format as | Time series | Query Builder

Grafana


Satellite
SAT_1
Show eclipses
Battery charge/discharge
36.0 V
35.0 V
34.0 V
33.0 V
32.0 V
31.0 V
10:30
11:00
11:30
12:00
12:30
13:00
13:30
14:00
14:30
15:00
SAT 1:BAT 1 VOLTAGE

Conclusions

- You don’t need to spend a fortune on a data analytics platform if you set your goals clearly
- PostgreSQL & TimescaleDB can be deployed anywhere, from a laptop to a cluster, scaling to your needs
- “One size fits all” is not always optimal
  – each team can afford their own installation while sharing a repository with configuration and reference data
- Let your users choose their tools

Thank you!

References

1. Building columnar compression in a row-oriented database
2. Incremental materialized view maintenance for PostgreSQL 14
3. Grafana global variables
4. PostgreSQL Wiki: Inlining of SQL functions

Find out more…

tgss.terma.com

About Terma:

www.terma.com/press/newsletter
www.linkedin.com/company/terma-a-s
www.youtube.com/user/TermaTV

TERMA
ALLIES IN INNOVATION