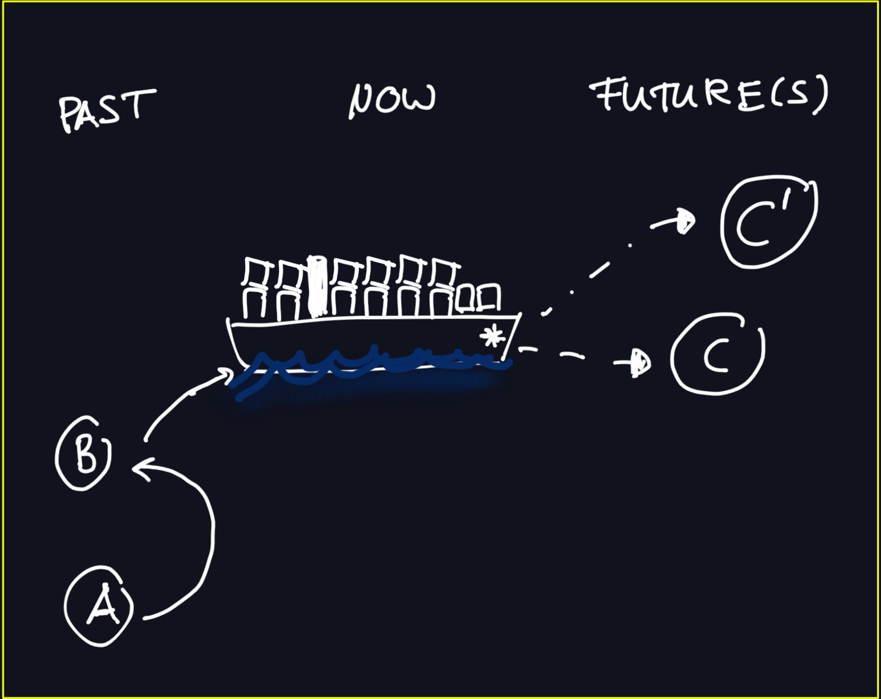
THE BEAUTY OF DELTA FOR POLYGLOT DATA AND ML WORKLOADS

Micha Kunze
Date 2024-06-13

CONTEXT

We ship data and ML

- Transported by Maersk
- 20+ ML products
- 1200+ datasets for analytics and operational data
- Mixed batch and streaming



DATA PLATFORM

Batch & Streaming

- Apache Spark for batch and Structured Streaming
- Delta Lake
- Pandas
- Apache Flink

Analytics & Operations

- All datasets available in a metastore
- Structured Streaming to feed operational stores

Decision Automation

- 20+ ML products
- Simulations using historical data
- Integration with operational apps/services

THE BEAUTY OF DELTA [METADATA]

DELTA LAKE

Self-described open table format

- No extra component needed (catalog)
- Protocol works with many engines
- Easy setup and testing

Rich metadata

- Transaction log + table history
- Change data feed
- Version control

THE BEAUTY OF DELTA [METADATA]

DELTA LAKE

Self-described open table format

- No extra component needed (catalog)
- Protocol works with many engines
- Easy setup and testing

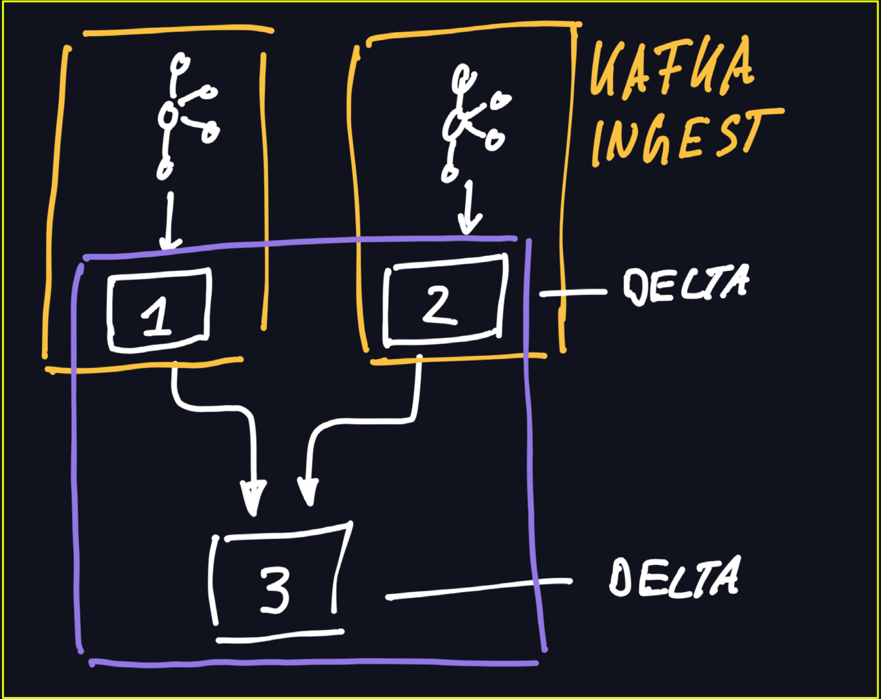
Rich metadata

- Transaction log + table history
- Change data feed
- Version control

RUN JOBS ONLY WHEN NEEDED

DELTA & ORCHESTRATION

Only kick off a job when data changes!



Without Delta Lake

- Reactive in-house scheduler
- Runs when upstream ran

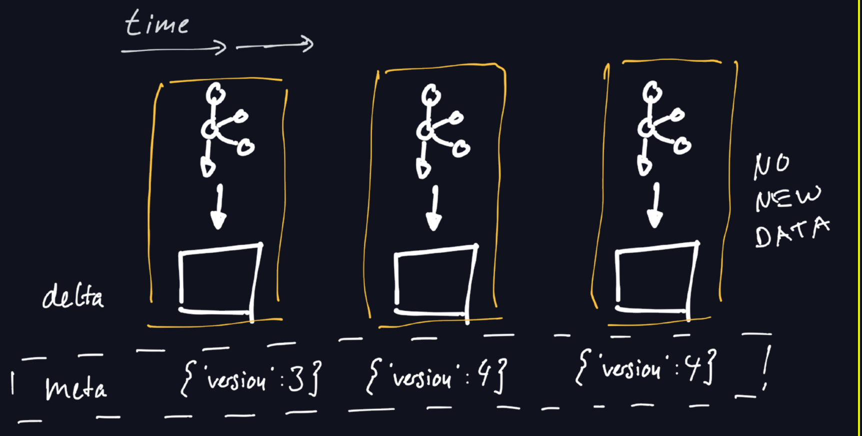
With Delta Lake

- Run only if delta log shows data changed

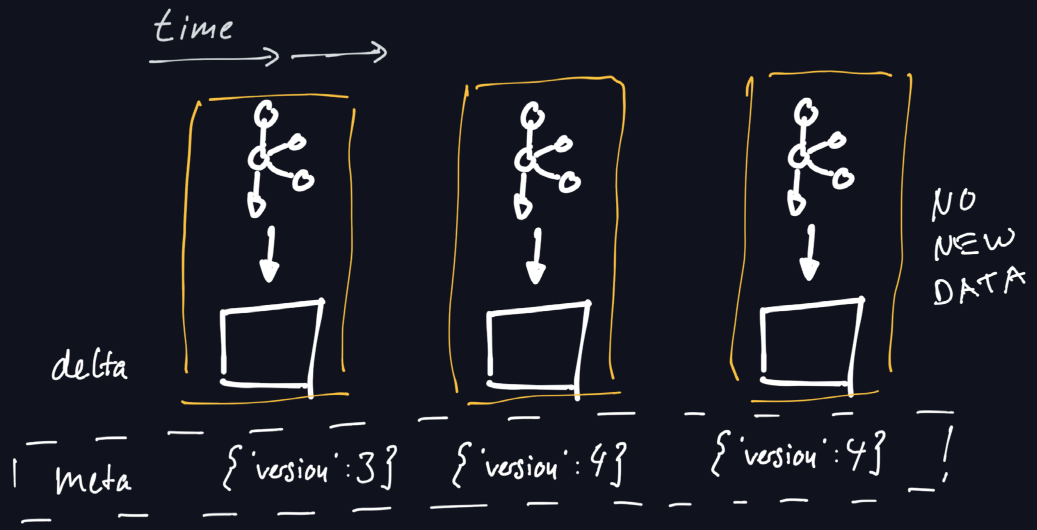
ONLY RUN ON DATA CHANGE

- Metadata at the destination which delta version the table has
- Downstream job uses that version and the version it used last

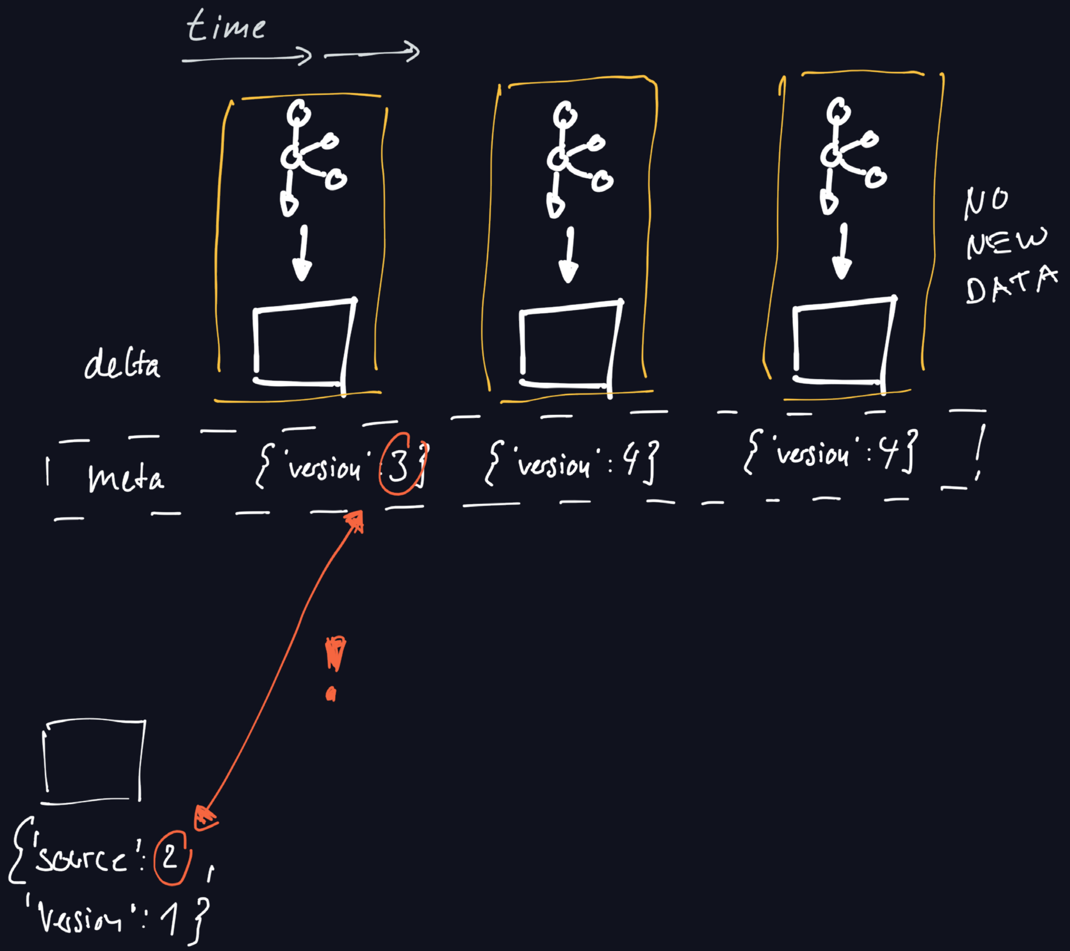
	version	timestamp	operation
1	55724	2024-05-01T19:40:15.000+00:00	WRITE
2	55723	2024-05-01T19:31:10.000+00:00	WRITE
3	55722	2024-05-01T19:30:26.000+00:00	OPTIMIZE
4	55721	2024-05-01T19:22:13.000+00:00	WRITE



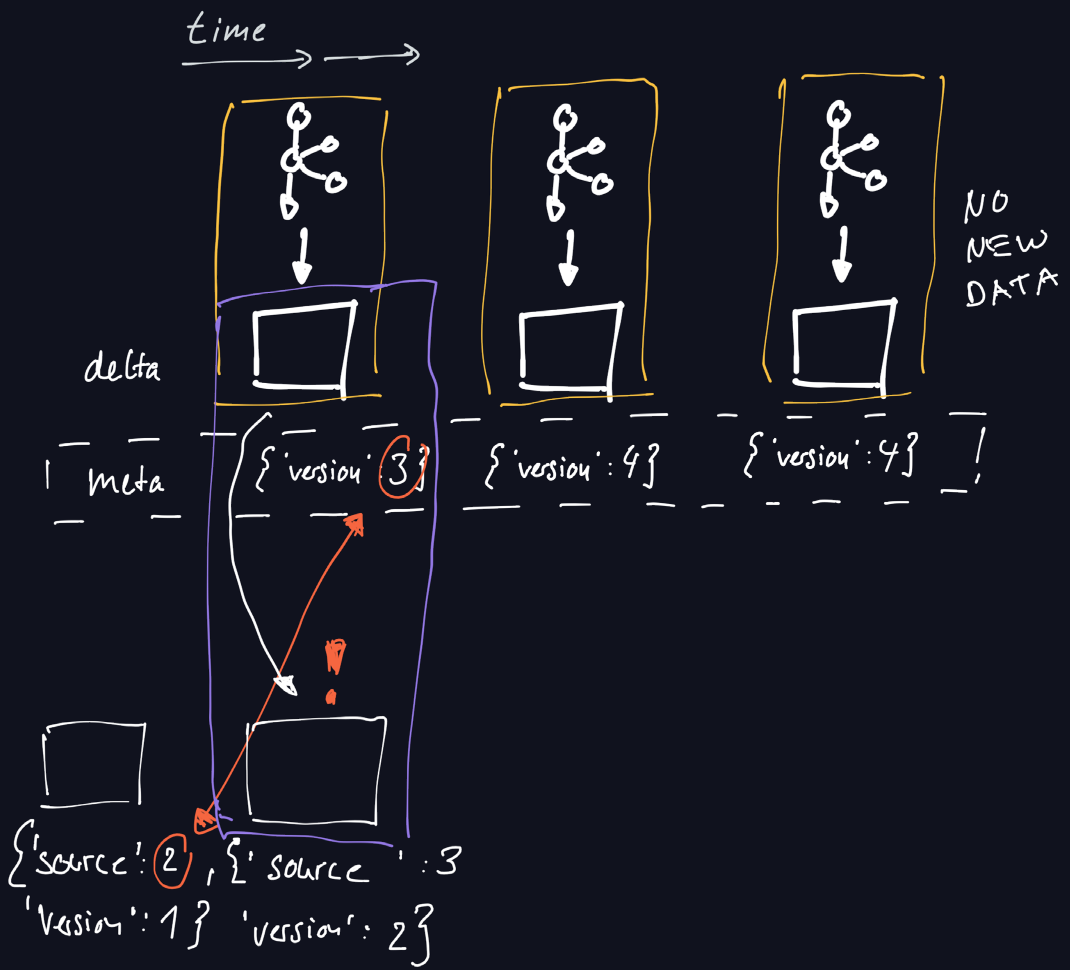
DETAILS



DETAILS

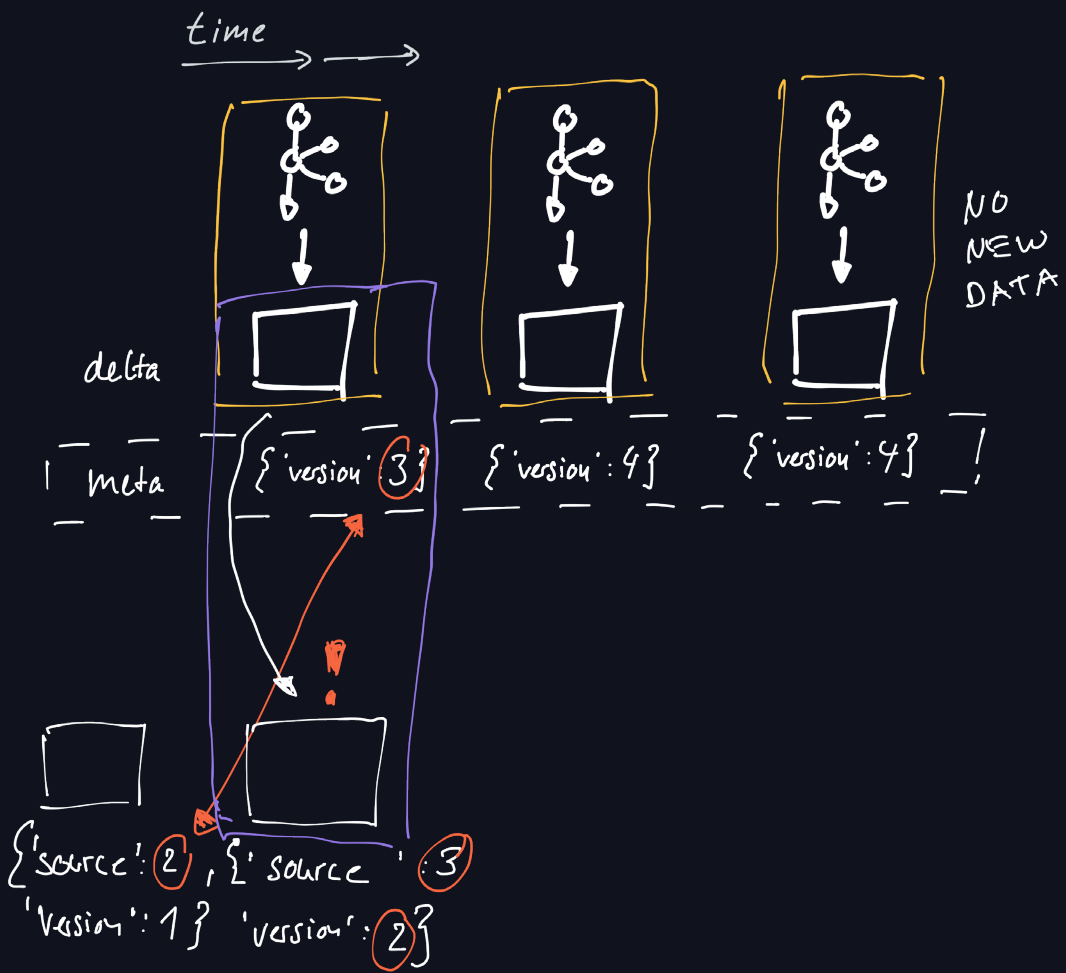

time
NO NEW DATA
delta
meta
{'version':3}
{'version':4}
{'version':4}
!
!
{'source':2,
'version':1}

DETAILS

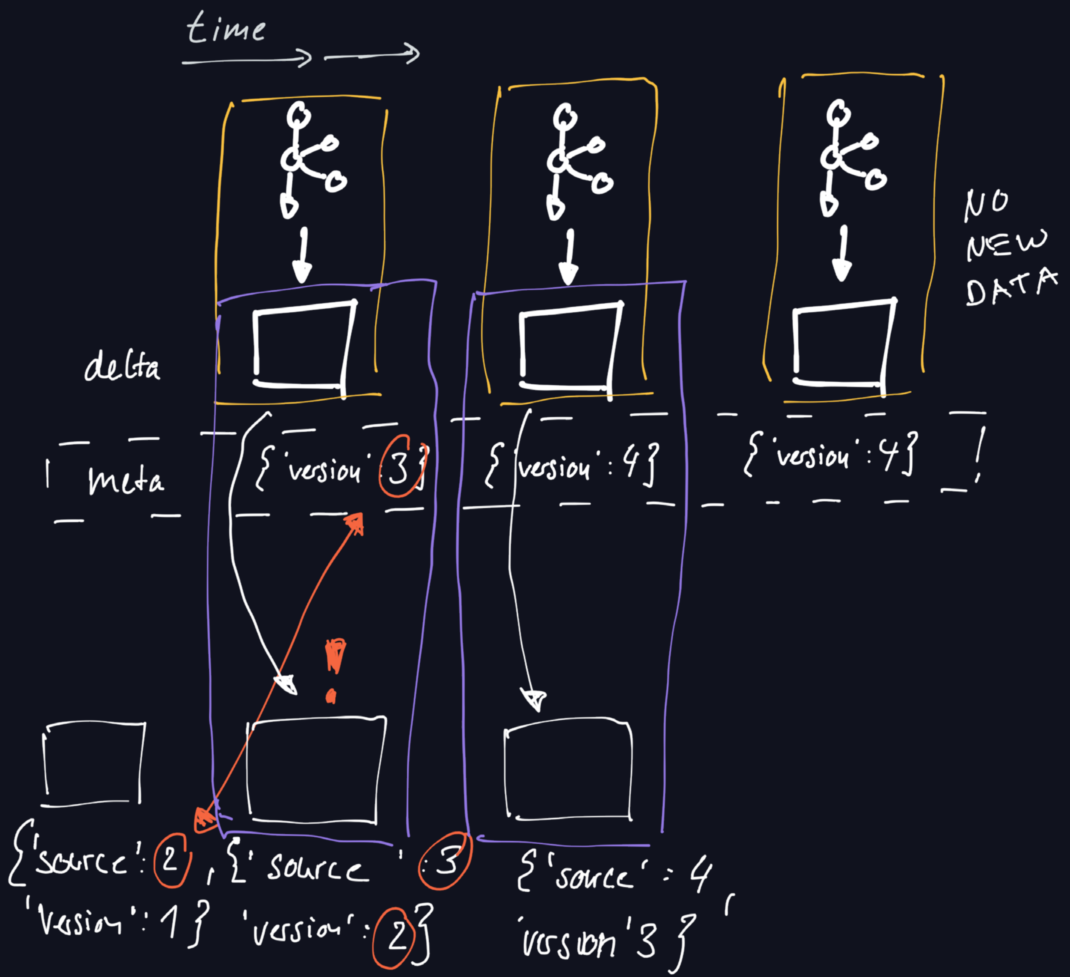


time
NO NEW DATA
delta
meta
{'version':3}
{'version':4}
{'version':4}
{'source':2, 'version':1}
{'source':3, 'version':2}

DETAILS

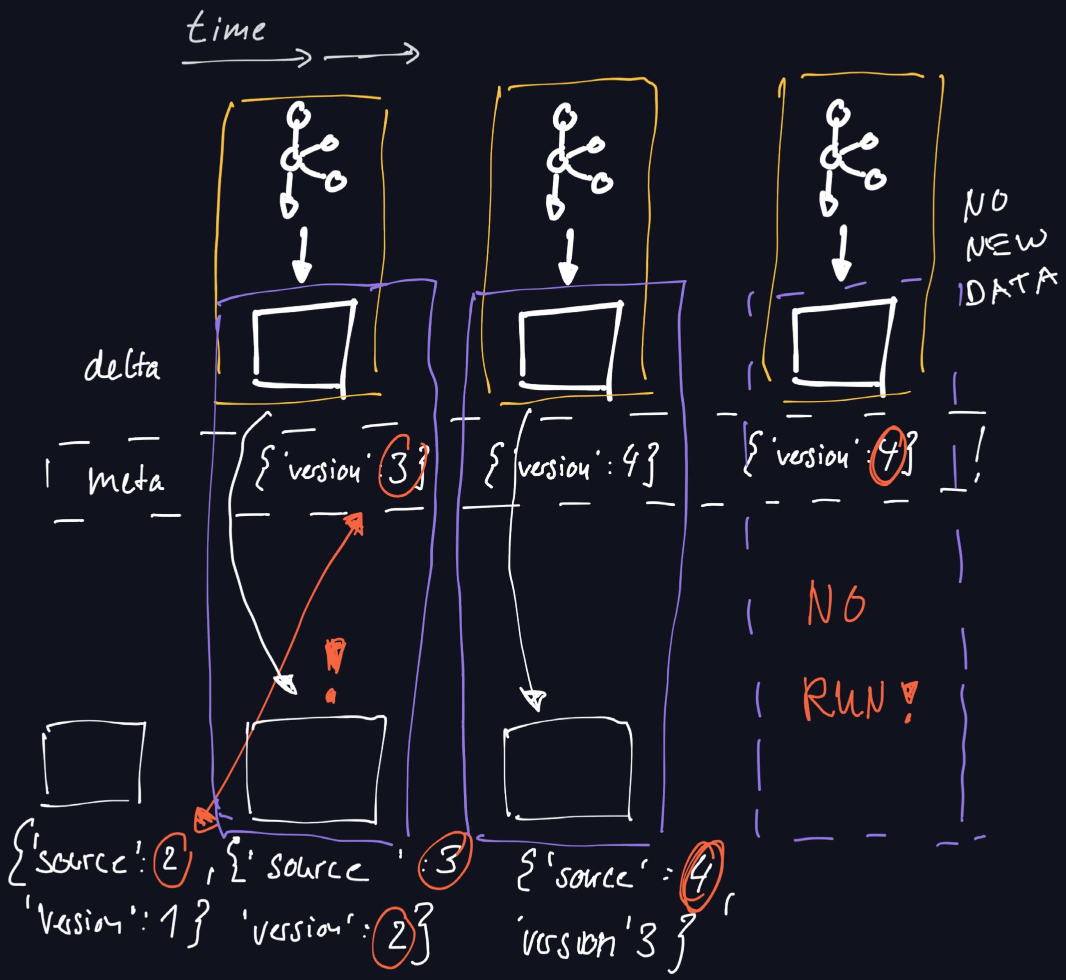

time
delta
meta
{'version': 3}
{'version': 4}
{'version': 4}
NO NEW DATA
{'source': 2, 'version': 1}
{'source': 3, 'version': 2}

DETAILS



time
NO
NEW
DATA
delta
meta
{'version':3}
{'version':4}
{'version':4}
{'source':2, 'version':1}
{'source':3, 'version':2}
{'source':4, 'version'3}

DETAILS



time
delta
meta
{ 'version' : 3 }
{ 'version' : 4 }
{ 'version' : 4 }
NO NEW DATA
NO RUN !
{ 'source' : 2 , 'version' : 1 }
{ 'source' : 3 'version' : 2 }
{ 'source' : 4 , 'version' 3 }

TL;DR

Opportunities

- Smarter scheduling of jobs: “only when needed”
- Reduced our daily job executions by >10%
- Great for Structured Streaming where data is not constantly being updated -> slow changing data

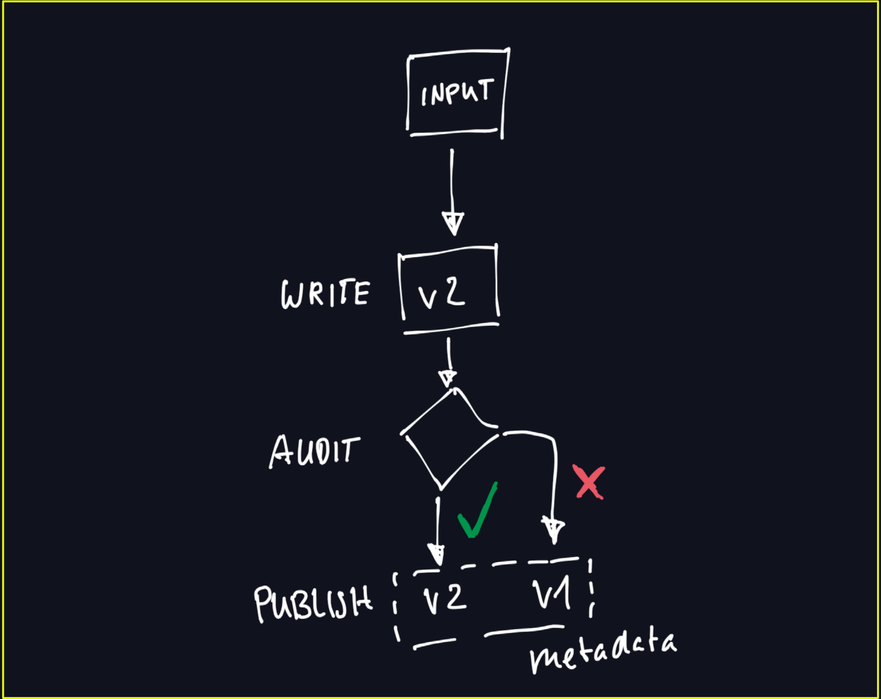
Pitfalls

- Need to take care of VACUUM/OPTIMIZE entries -> make sure streams can catch up

WRITE
AUDIT
PUBLISH

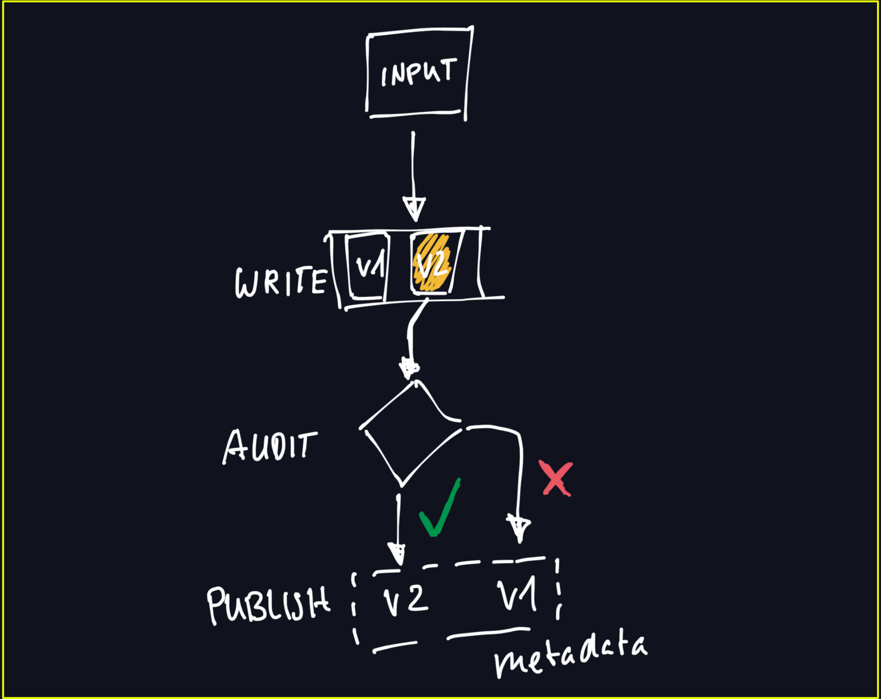
WAP - BATCH

- Run job and commit data
- Validate the entire dataset
    - ✅: save latest version in meta
    - ❌: keep previous version in meta
- Readers read version specified in metadata and downstream will not run (previous section)



WAP - STRUCTURED STREAMING

- Run job and commit data
- Validate the latest changes only
  - ✅: save latest version in meta
  - ❌: keep previous version in meta
- Readers read version specified in metadata



TIP - STRUCTURED STREAMING

Enable change data feed

PYTHON

```python
# function to enable change data feed if not yet enabled
def enable_change_feed_if_table_exists(self, spark: SparkSession, table_path: str):
    if DeltaTable.isDeltaTable(spark, table_path):

        dt = DeltaTable.forPath(spark, table_path)
        props: Dict[str, str] = dt.detail().select("properties").collect()[0][0]

        if props.get("delta.enableChangeDataFeed") != "true":
            # Enable change feed for future versions.
            spark.sql(f"ALTER TABLE delta.`{table_path}` SET TBLPROPERTIES (delta.enableChangeDataFeed = true)")
```

TIP - STRUCTURED STREAMING

Validate latest commit only

PYTHON

```python
def get_latest_append_batch(spark: SparkSession, filename: str) -> DataFrame:
    dt = DeltaTable.forPath(spark, filename)
    history = (
        dt.history(10)
        .filter(F.col("operation").isin(["WRITE", "STREAMING UPDATE"]))
        .select("version", "timestamp")
    )
    latest_version = history.collect()[0][0]
    return (
        spark.read.format("delta")
        .option("readChangeFeed", "true")
        .option("startingVersion", latest_version)
        .load(filename)
    )
```

TL;DR

Opportunities

- WAP is the best pattern to prevent bad data as early as possible
- Regular (~weekly) prevention of bad data/bad predictions being published
- Structured Streaming has limited WAP benefits, but freshness is often a great one

Pitfalls

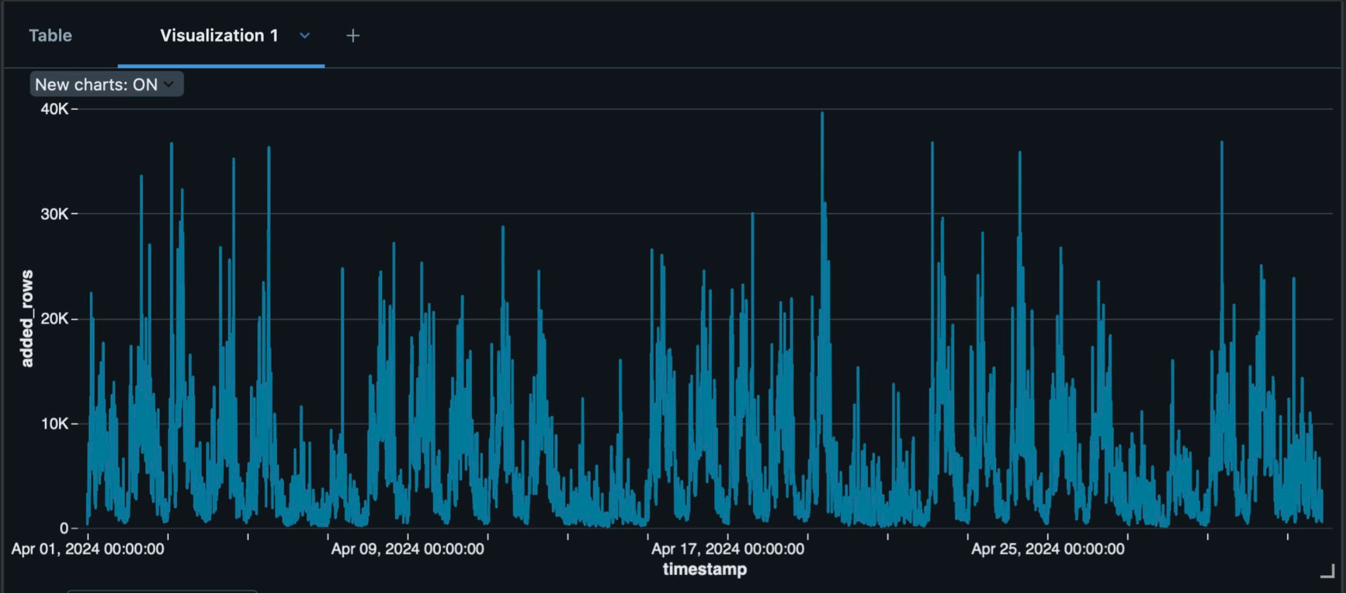
- The strength of WAP goes down with the size of the micro batch -> the closer you are to real time/small batches the less you can use WAP
- Implementing WAP with Delta Lake depends very much on your ability to integrate this with your tooling/orchestration

USE ML TO DETECT STREAMING ISSUES

VELOCITY OF DATA

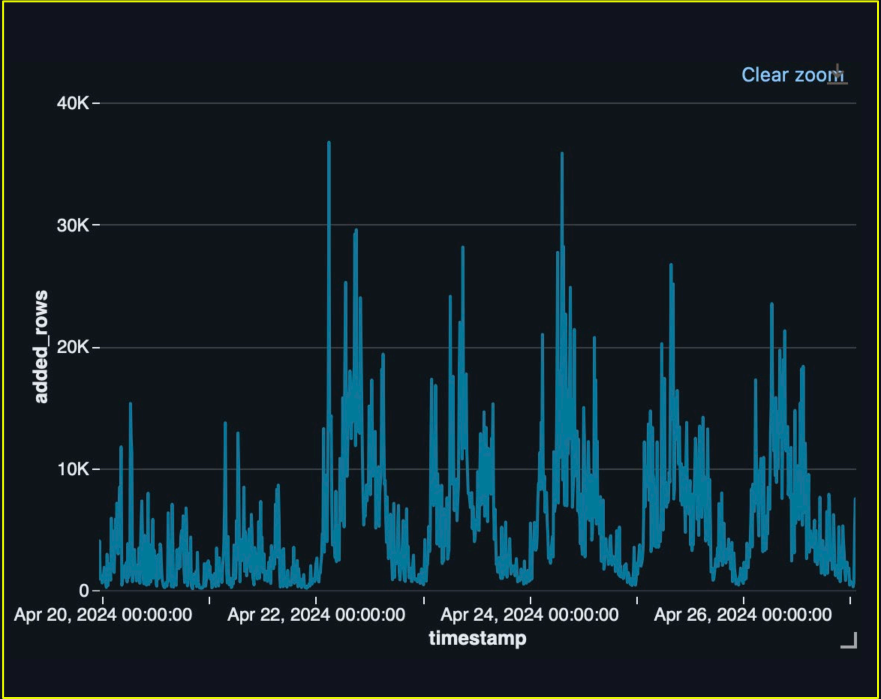
Delta log to the rescue

```
WITH HIST AS (DESCRIBE HISTORY [redacted])
SELECT timestamp, cast(operationMetrics.numOutputRows as integer) as added_rows
FROM HIST
WHERE operation = "WRITE"
```



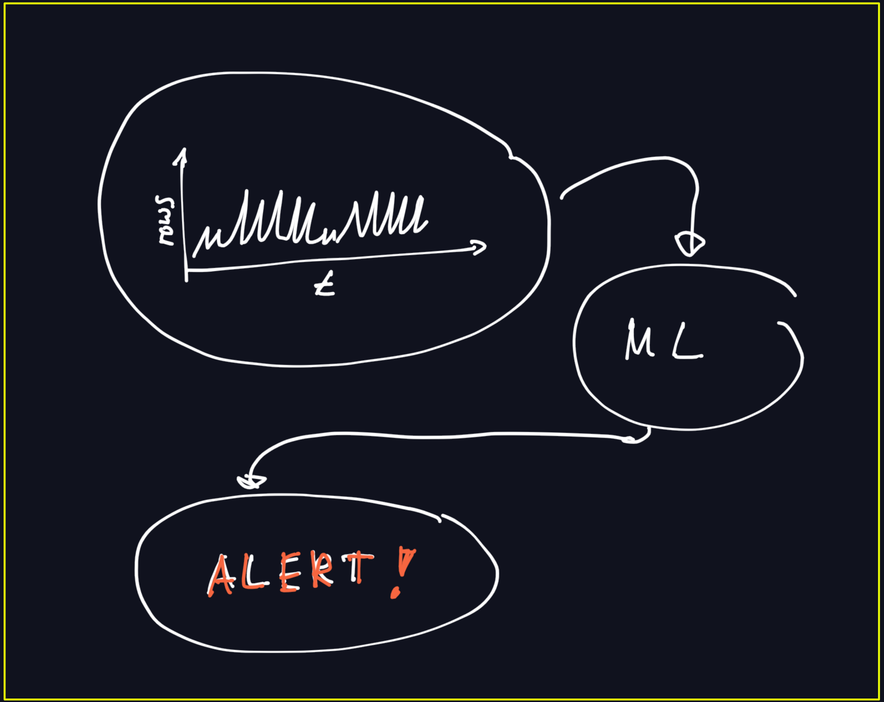
THRESHOLDS DO NOT WORK!

- How to set min and max?
- How to deal with daily/weekly trends?



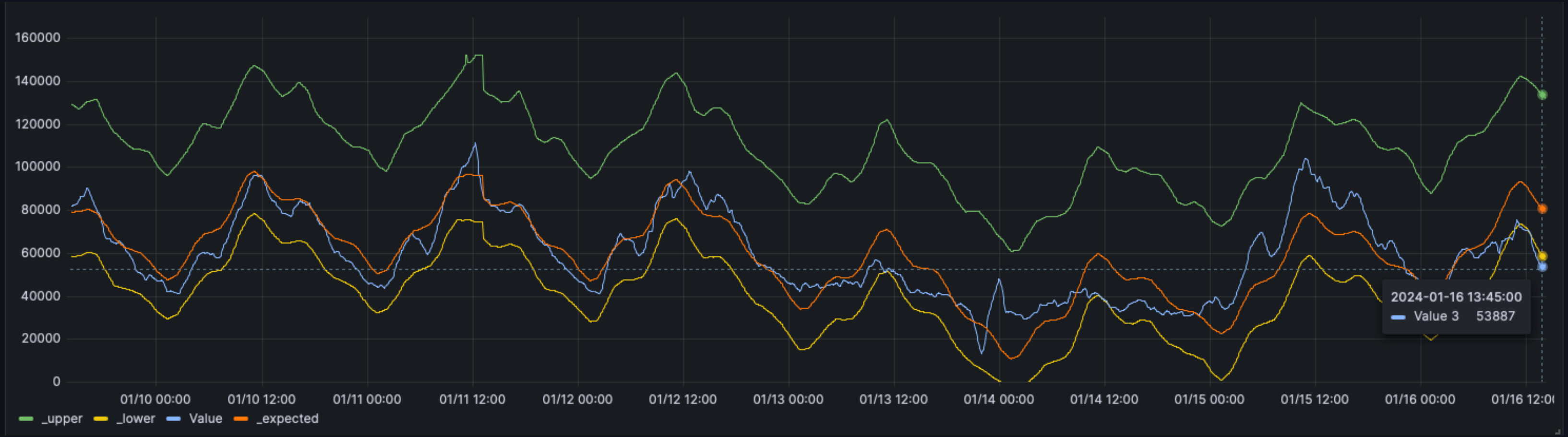
ML BASED ALERTING

- Get velocity data from delta log
- Train auto-ML model
- Alert when velocity is off on job runtime
    - Check added rows from latest commit/time vs expected velocity



ML BASED ALERTING

In prod



TL;DR

Opportunities

- Get observability/alerts on when data velocity is not as expected
- Great for regularly incoming data with strong week/day/season patterns
- Prophet was all we used

Pitfalls

- Low velocity data is impossible to predict well
- Need to smooth the data a little to not be too noisy
- Beware of Structured Streams that commit multiple times per micro batch -> e.g. Kafka tombstones