Stream processing - part 2

Wojciech Rybak
RTB House

Agenda

- agenda
- intro
- advantages of stream processing
- basic concepts
- design patterns
- fault tolerance
- lambda architecture

What is stream processing?

- “[stream processing is] a type of data processing engine that is designed with infinite data sets in mind” - Tyler Akidau
- infinite == unbounded, always growing
- everything is a stream

Attributes of a data stream

Streams are:

- ordered
- immutable
- replayable

Request-response vs batch vs stream

Request-response:

- low latency
- often blocking
- also known as OLTP - online transaction processing

Batch:

- high latency
- high throughput
- great efficiency
- stale data

Stream:

- nonblocking
- reasonably low latency
- continuous and ongoing

Historical timeline

- 2002: “Models and issues in data stream systems”
- 2003: “TelegraphCQ: continuous dataflow processing”
- …
- Jan 2011: Kafka
- May 2011: Apache Flink
- Sep 2011: Apache Storm
- …
- 2013: Amazon Kinesis
- 2015: Google Cloud Pub/Sub + Dataflow
- 2015: Microsoft Azure Stream Analytics

Typical use cases

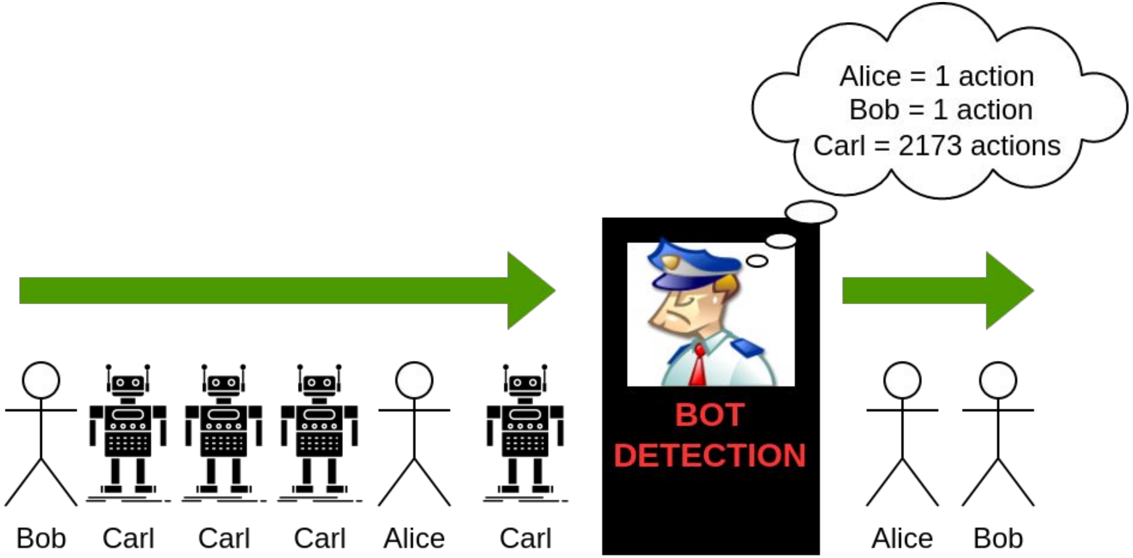
- complex event processing
- stream analytics
- maintaining materialized views
- search on streams

Real time analysis

- up to date information
- faster decision making
- faster reaction to market fluctuations
- easier detection and addressing of issues
- improved customer experience
- increased agility and optimization

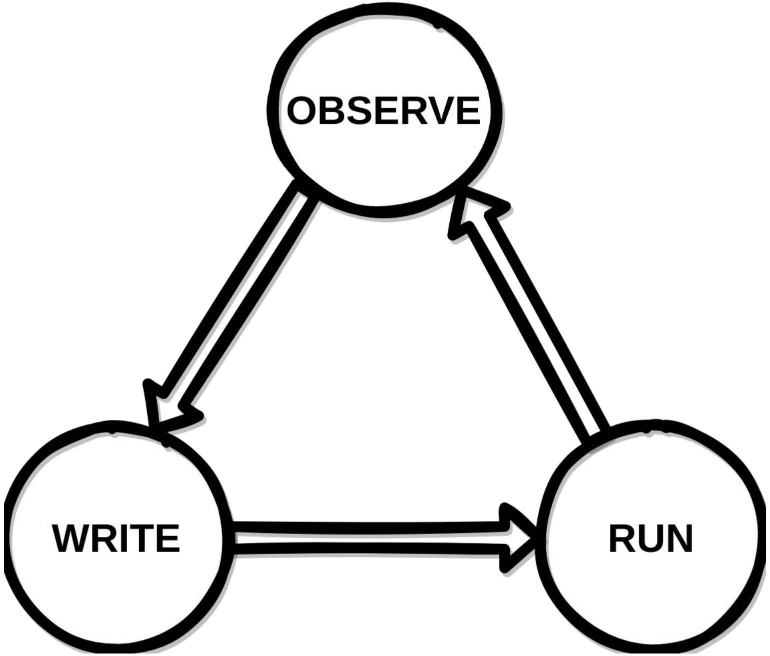
Practical example: bot detector

- certain bots are easy to detect (they make hundreds of actions per minute)
- but difficult to eliminate (they live only for several minutes)
- we can scan a stream of user actions and filter out users with too many actions



Short feedback loop when coding

- faster development
- better productivity



Is stream processing always the best approach?

- no

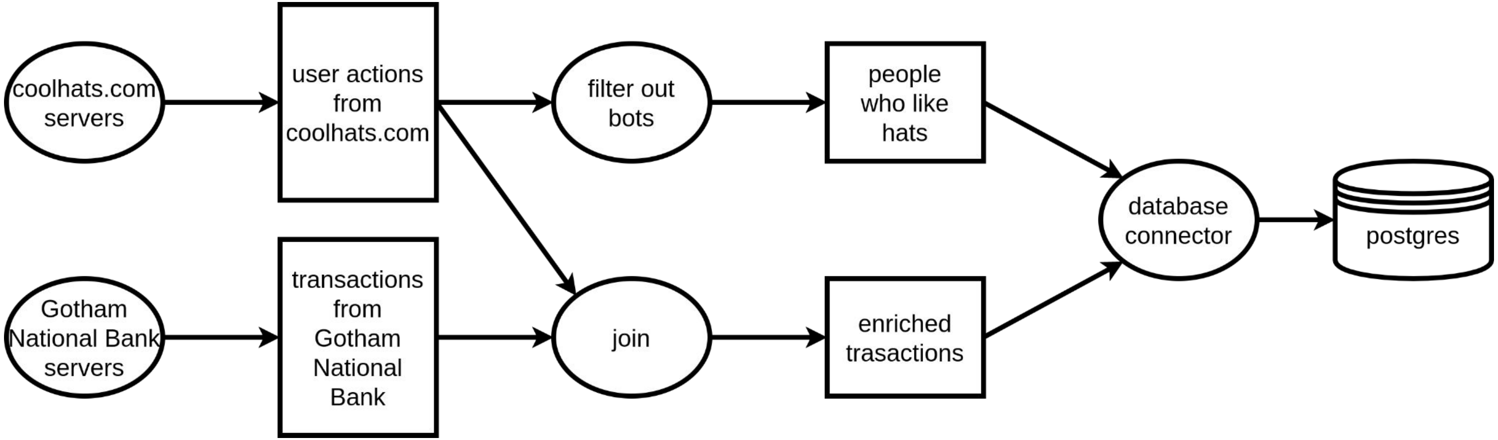
Stream processing concepts

A **stream** is a continuous body of surface water[1] flowing within the bed and banks of a channel. Depending on its location or certain characteristics, a stream may be referred to by a variety of local or regional names. Long large streams are usually called rivers, while smaller, less voluminous and more intermittent streams are known as streamlets, brooks or creeks, etc.

The flow of a stream is controlled by three inputs – surface runoffs (from precipitation or meltwater),

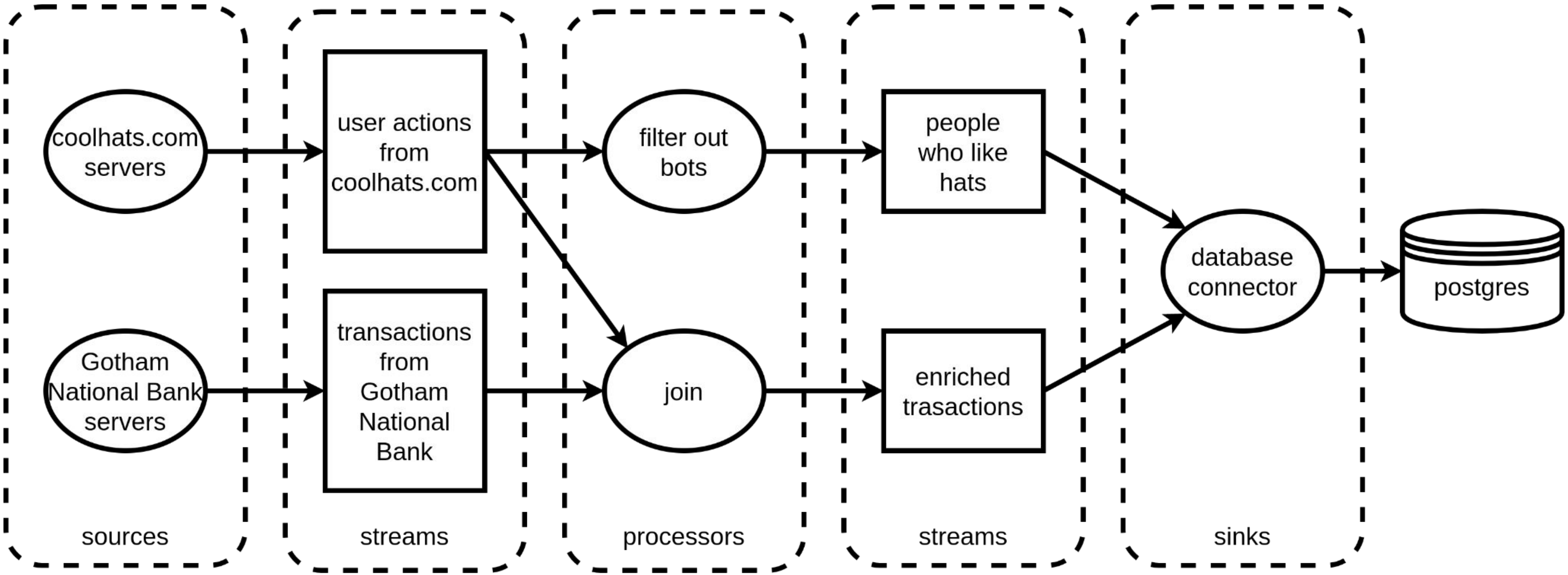
Aubach (Wiehl) in North Rhine-Westphalia, Germany

Topology



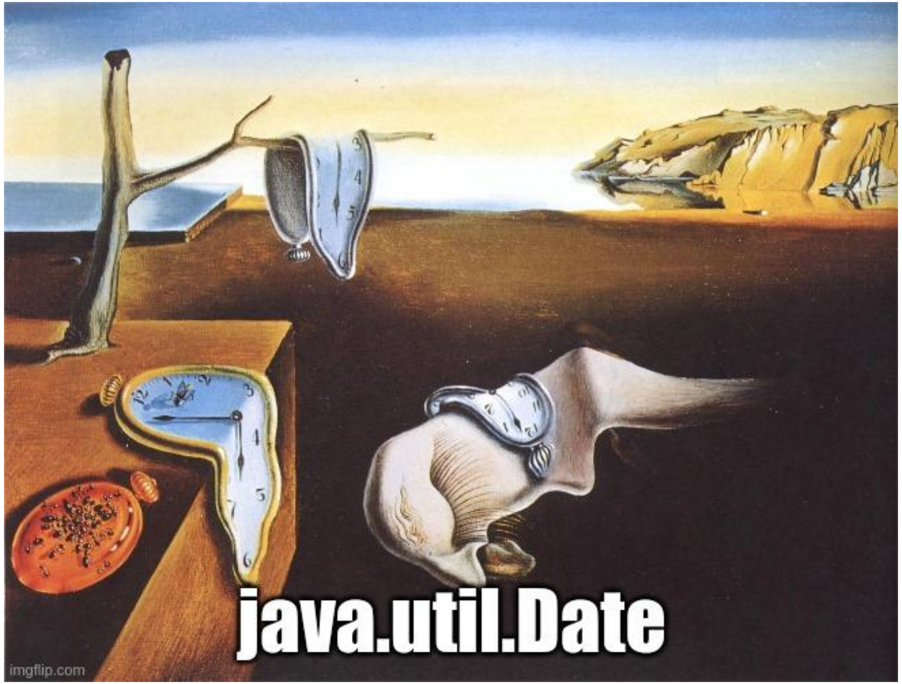
- logical abstraction for your stream processing code
- acyclic graph of sources, processors, and sinks

Topology


coolhats.com servers
Gotham National Bank servers
user actions from coolhats.com
transactions from Gotham National Bank
filter out bots
join
people who like hats
enriched trasactions
database connector
postgres
sources
streams
processors
streams
sinks

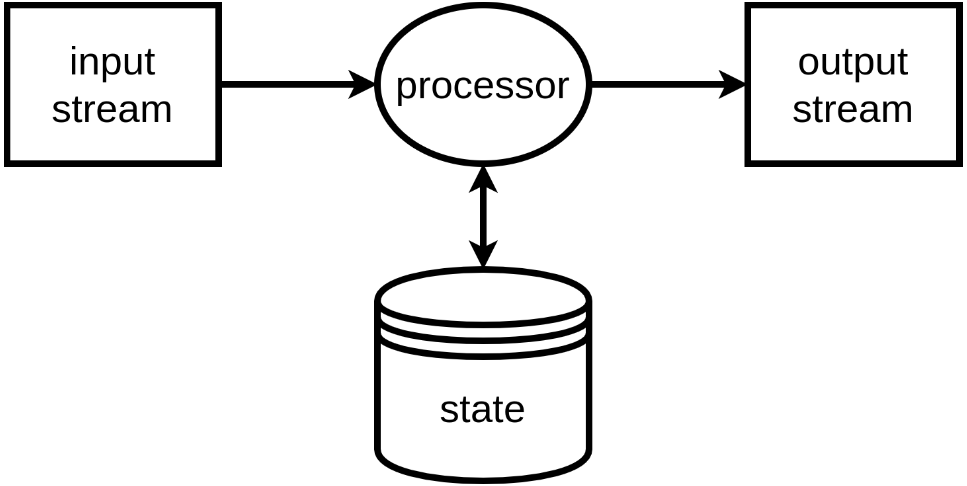
Time

- processing time
- ingestion time
- event time
- sometimes even more (our record: 20 timestamps in a single event)



State

- used to keep track of information about the events
- local: accessible only by a specific instance, within the application
- external: global; outside the application



Stream - table duality

TABLE

id	balance
101	1200
102	-300

UPDATE ... SET balance = 200 WHERE id = 101;
UPDATE ... SET balance = 3000 WHERE id = 101;
UPDATE ... SET balance = 700 WHERE id = 102;
UPDATE ... SET balance = 1200 WHERE id = 101;
UPDATE ... SET balance = -300 WHERE id = 102;

STREAM

101	200
101	3000
102	700
101	1200
102	-300

- a table is just a stream of updates
- a stream is just the change of a table over time

Stream - table duality

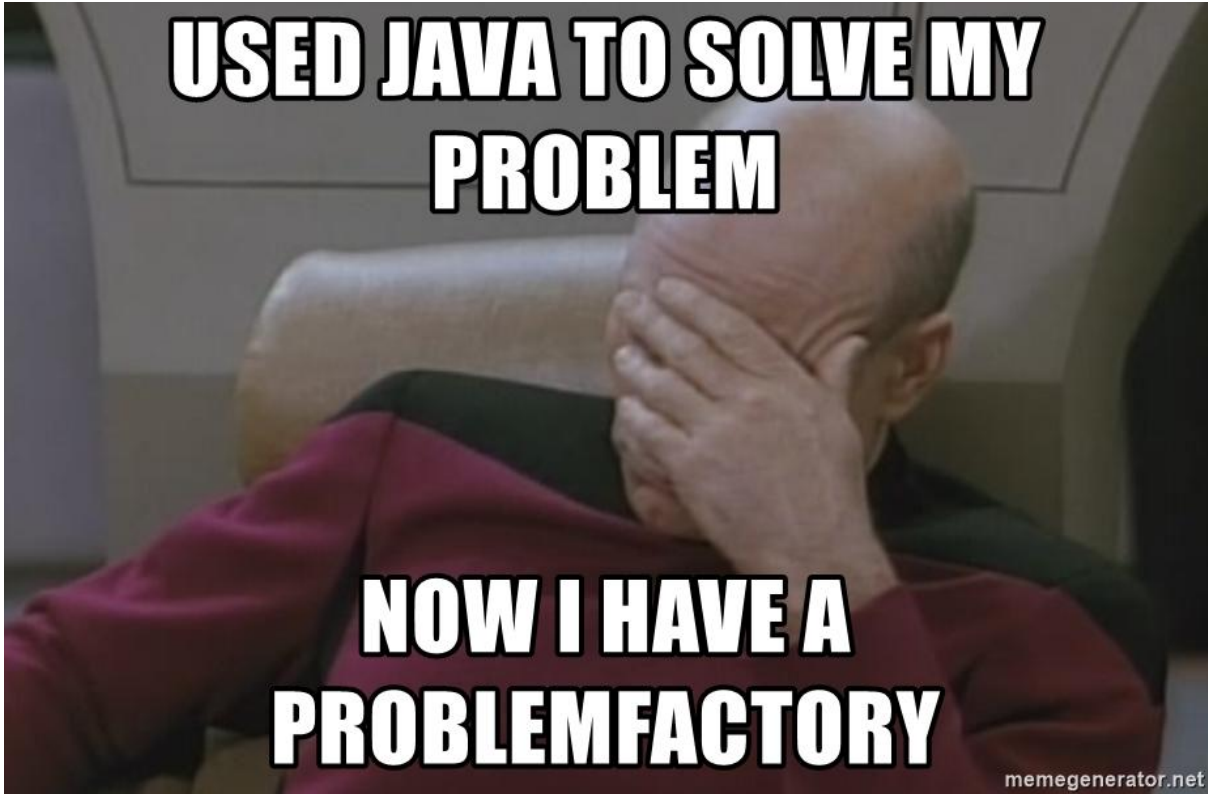
$$state(now) = \int_{t=0}^{now} stream(t)\ \mathrm{d}t$$

$$stream(t) = \frac{\mathrm{d}\, state(t)}{\mathrm{d}t}$$

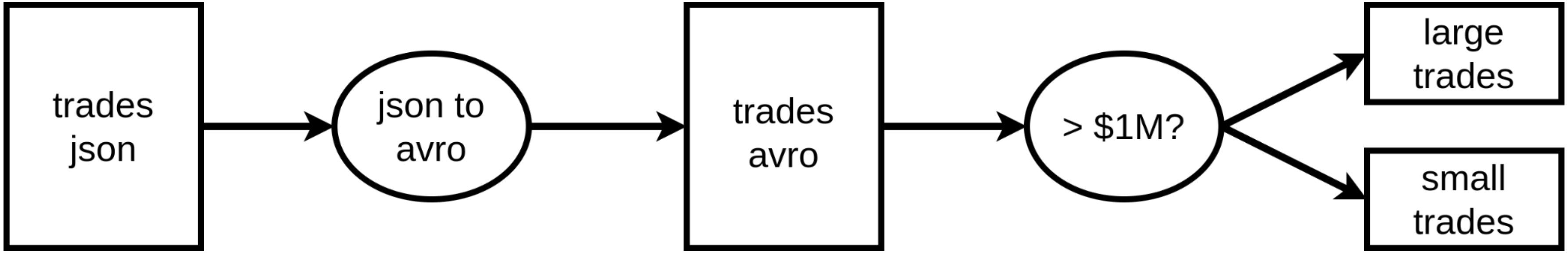
Processing guarantees

- at most once
- at least once
- exactly once

Design patterns

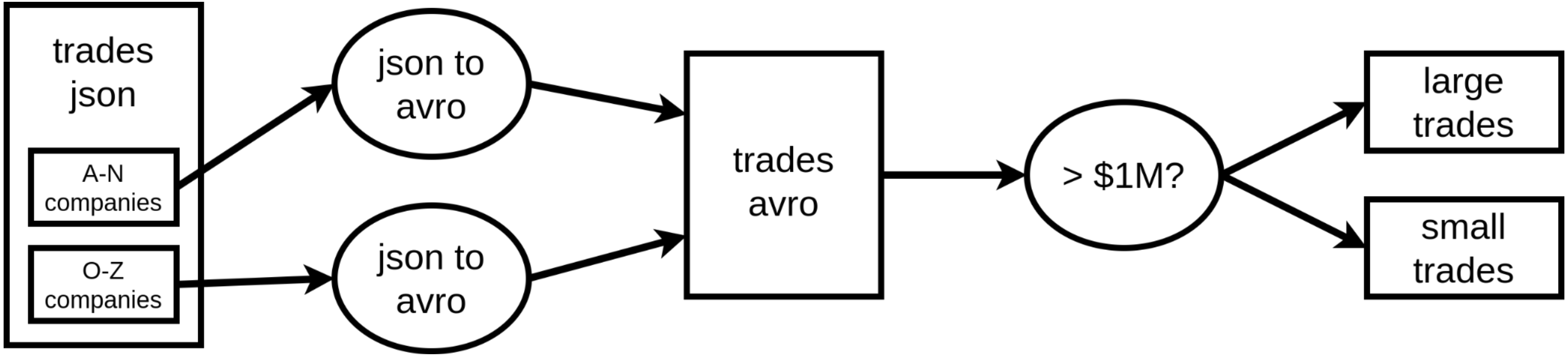


Single-event processing



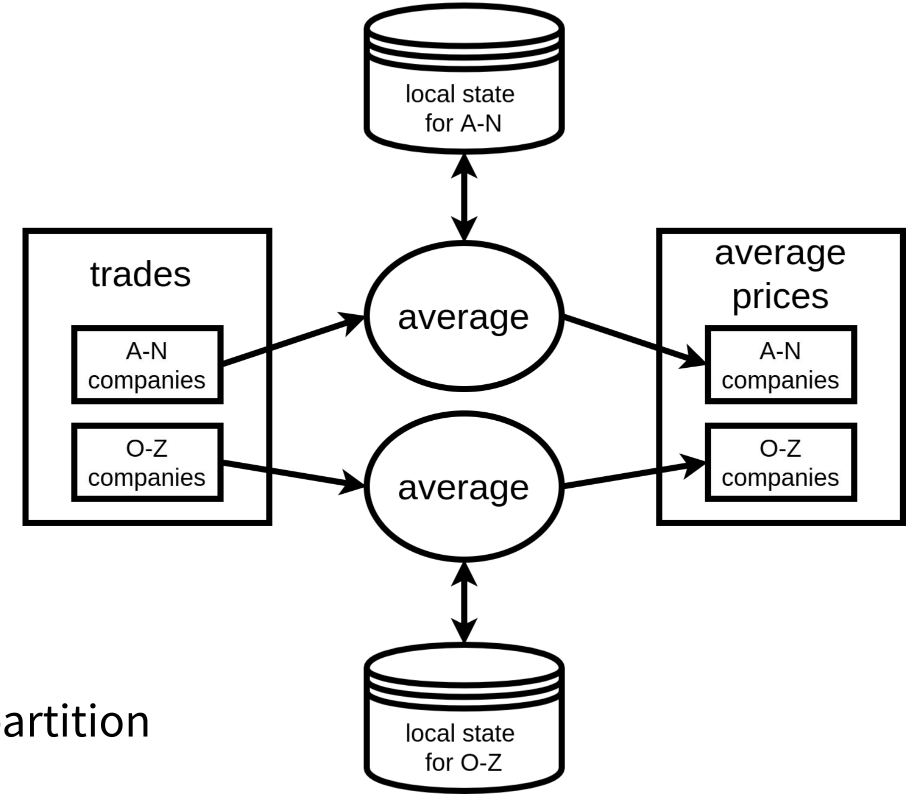
- aka map/filter pattern
- each event is handled independently
- no state == easy scaling & load balancing & failure recovery

Single-event processing



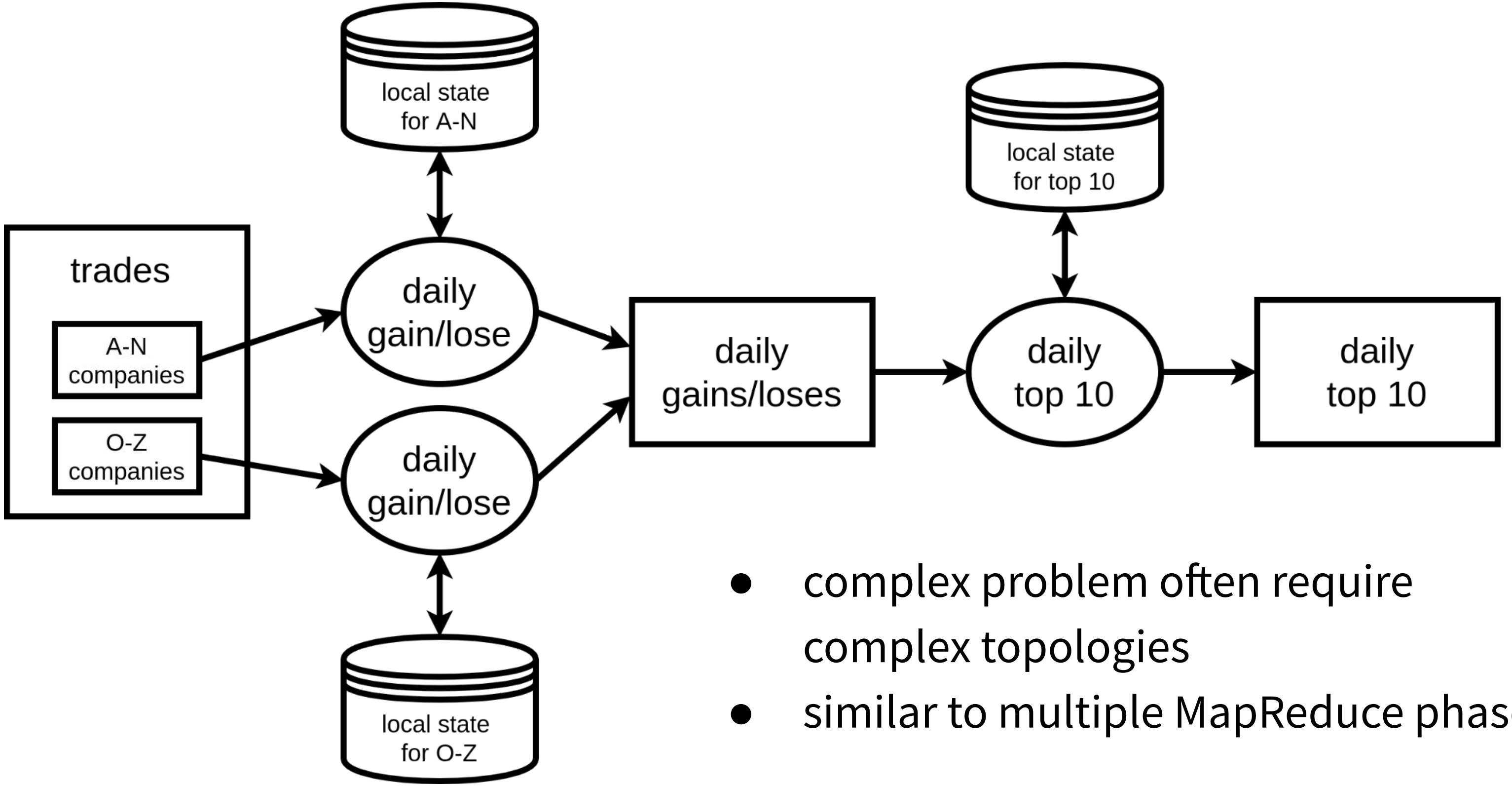
- aka map/filter pattern
- each event is handled independently
- no state == easy scaling & load balancing & failure recovery

Local state



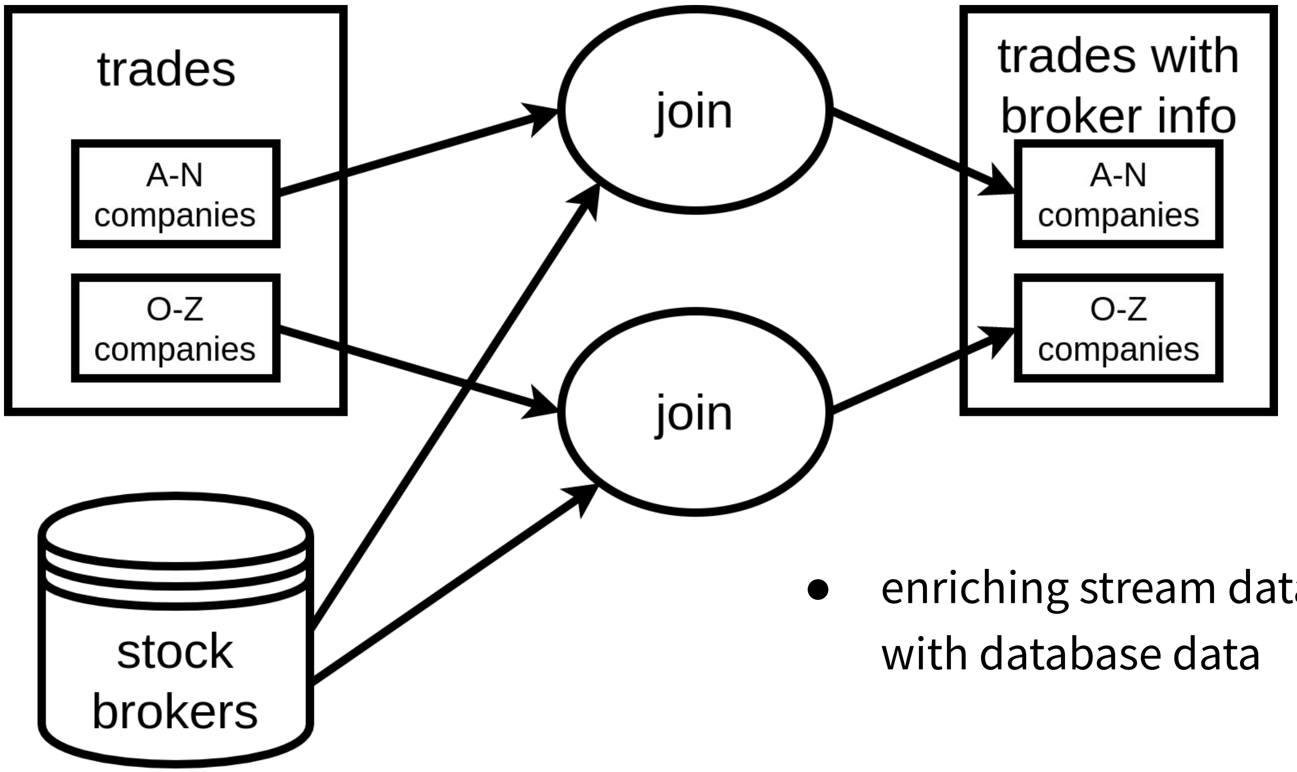
- used to aggregate information
- local state == aggregation per partition

Multiphase processing



- complex problem often require complex topologies
- similar to multiple MapReduce phases

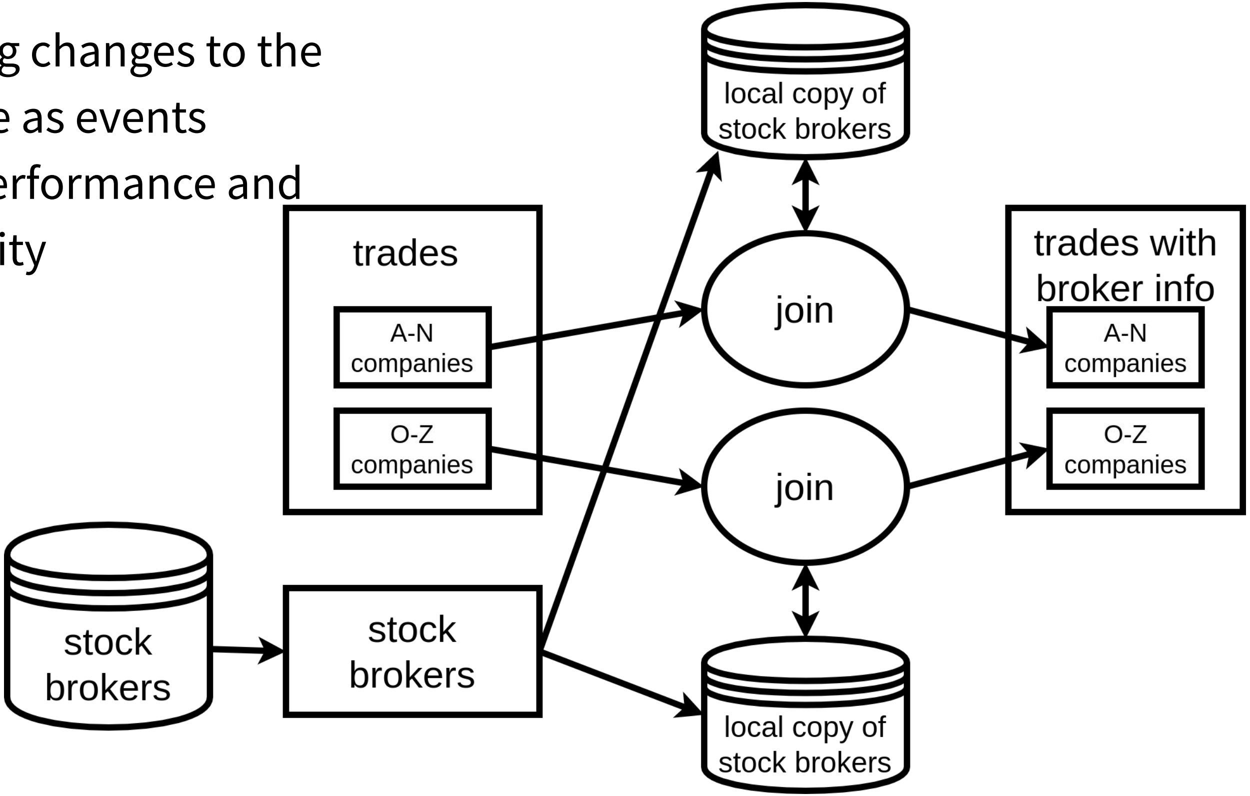
External lookup (stream - table join)



- enriching stream data with database data

External lookup (stream - table join)

- capturing changes to the database as events
- better performance and availability



Streaming join

- windowed join
- windows based on time

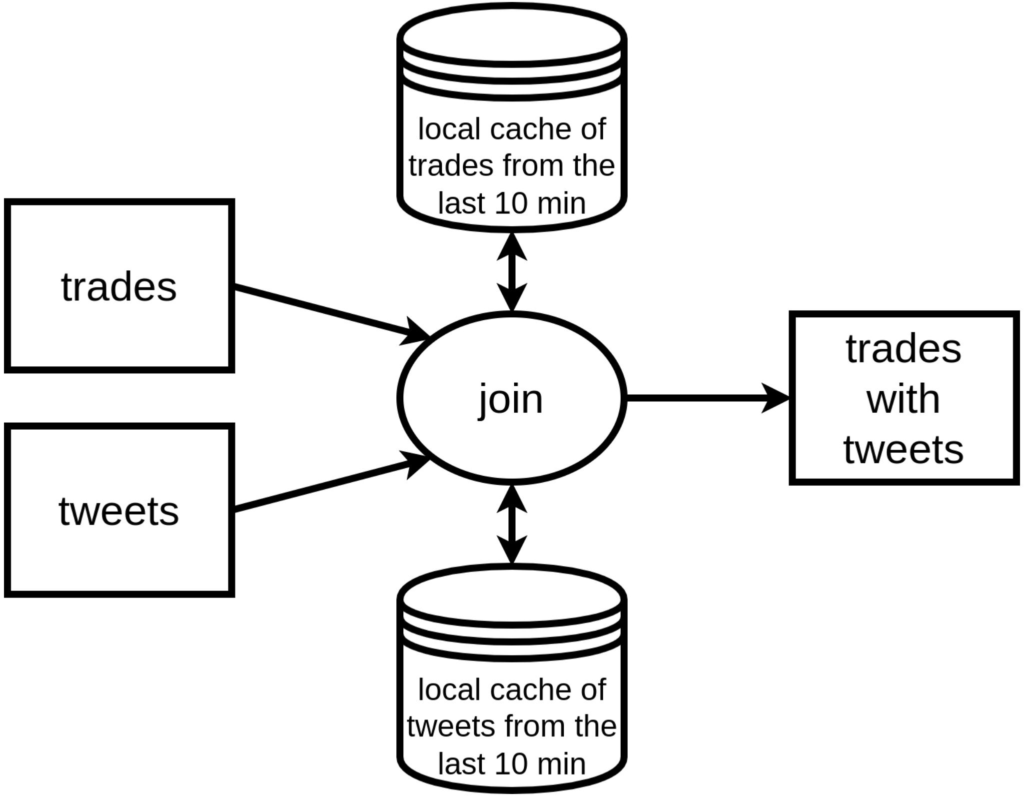


Table - table join

- nonwindowed join, we join the current state of both tables
- easy to scale
- makes stream processing feel "complete"

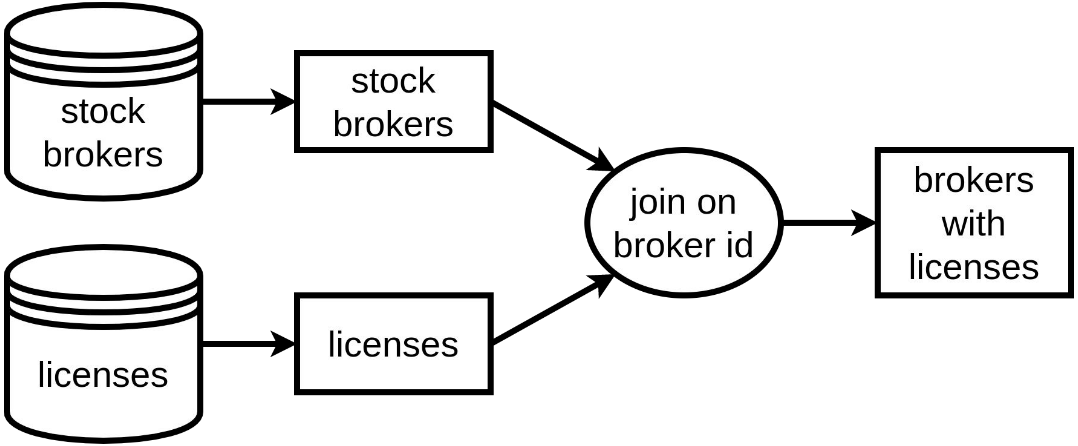
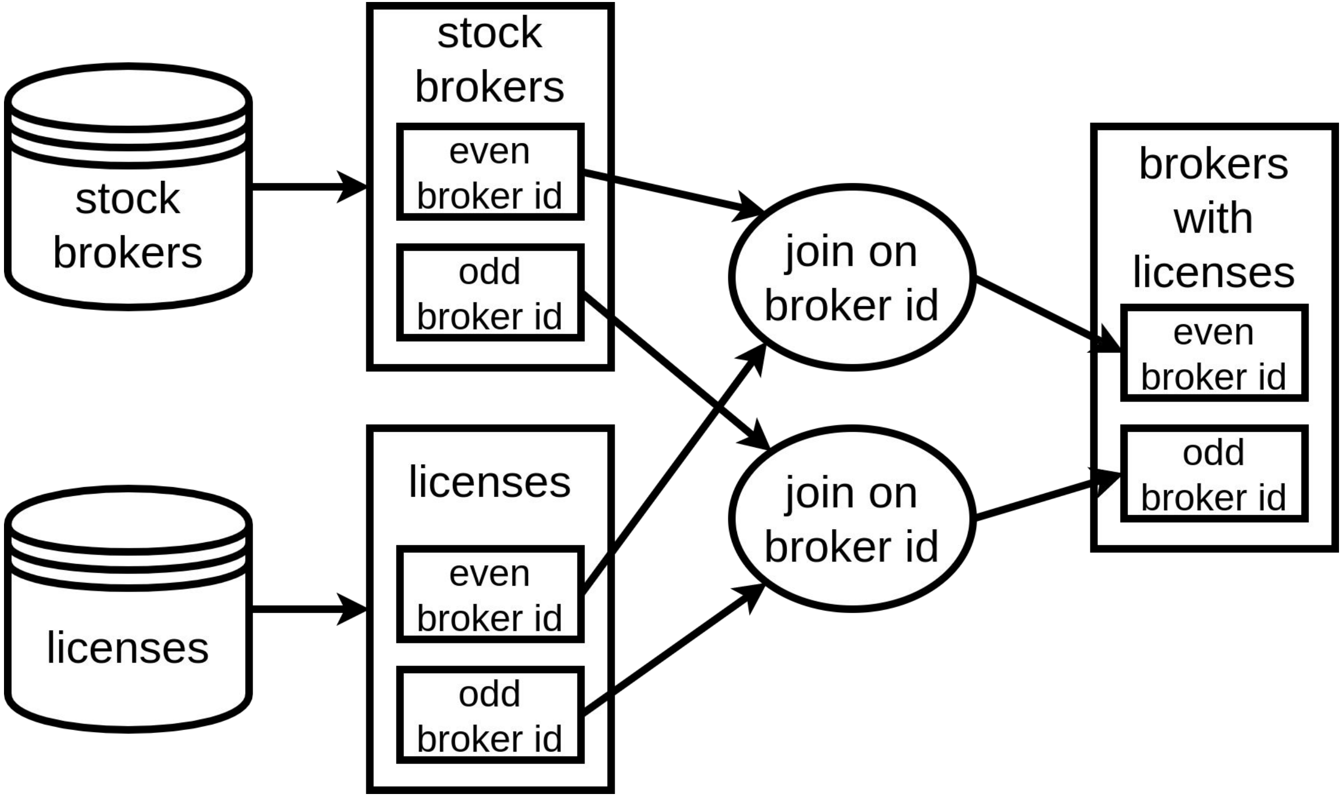
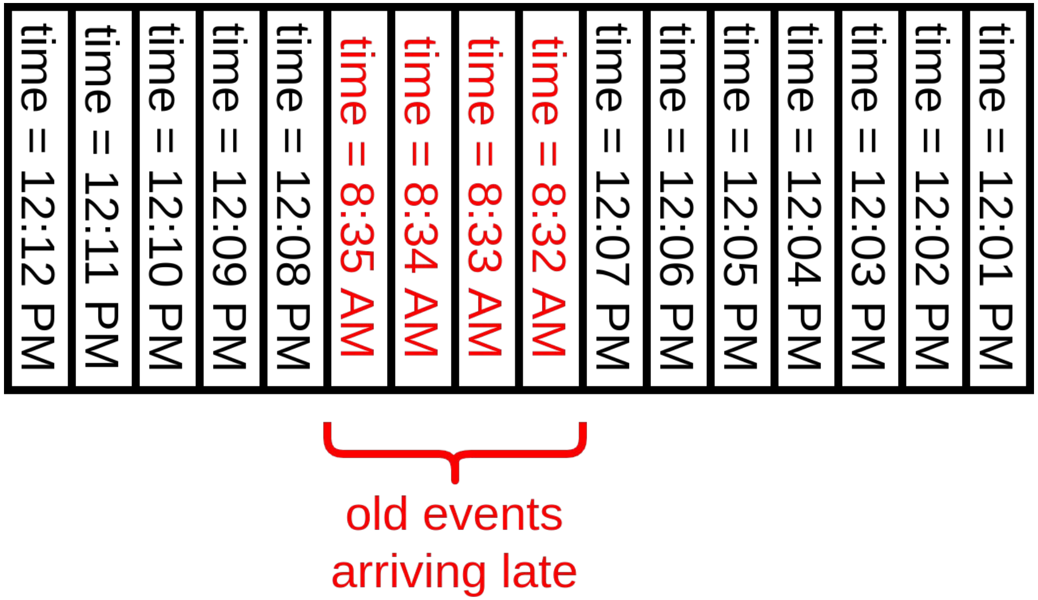


Table - table join

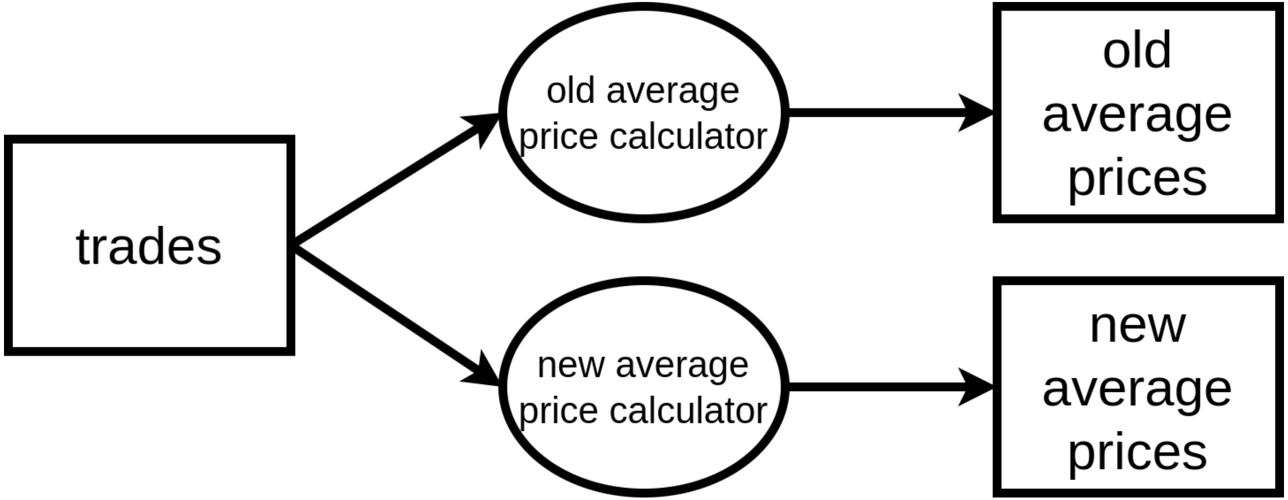


Out of sequence events

- example: mobile device reconnects after 5h in airplane mode
- no easy solution



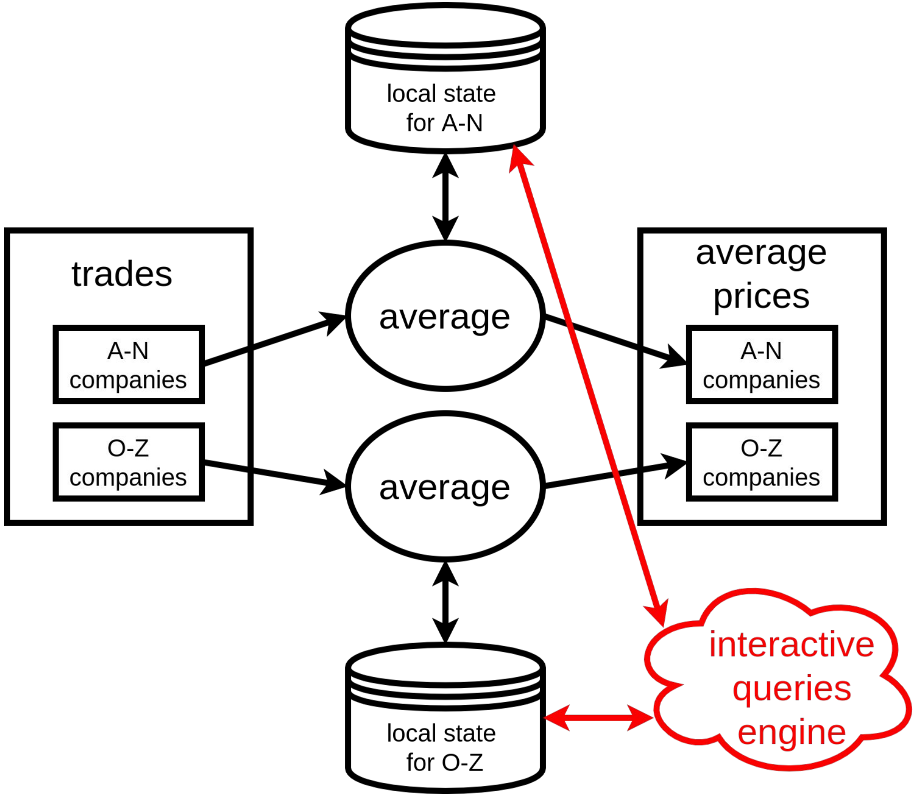
Reprocessing



- running two versions of the application along each other
- possible thanks to stream properties: replayability, immutability, and ordering

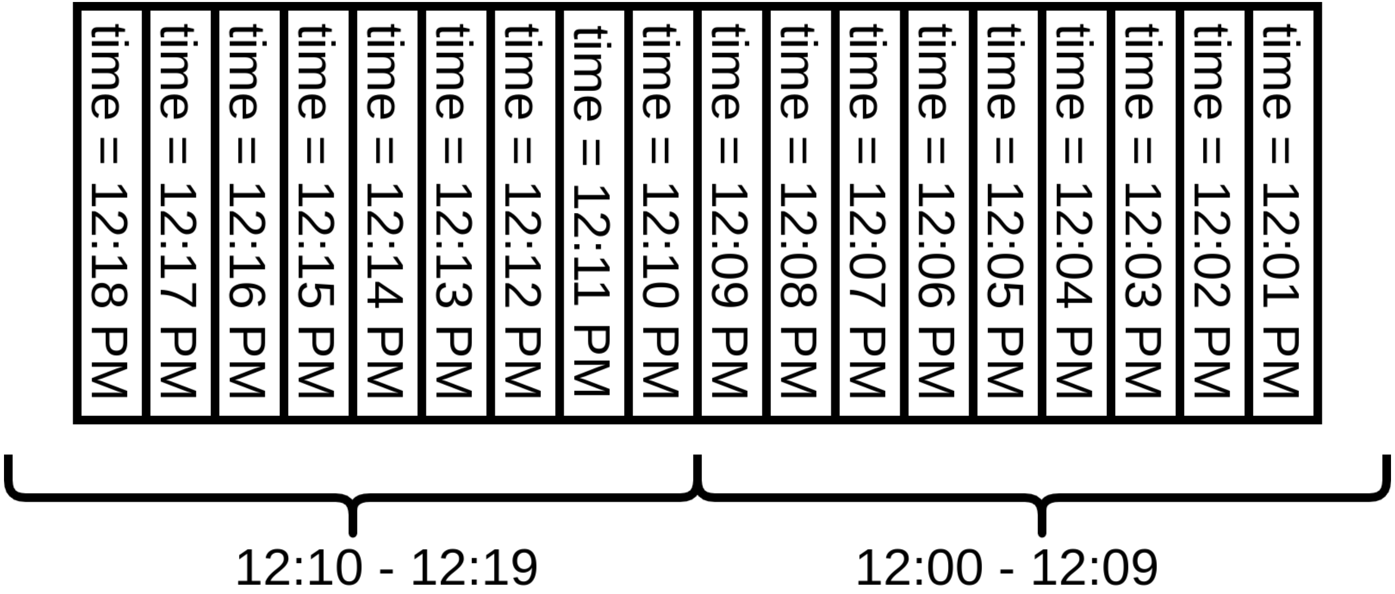
Interactive queries

- read the results directly from the state instead of querying the database or output topic

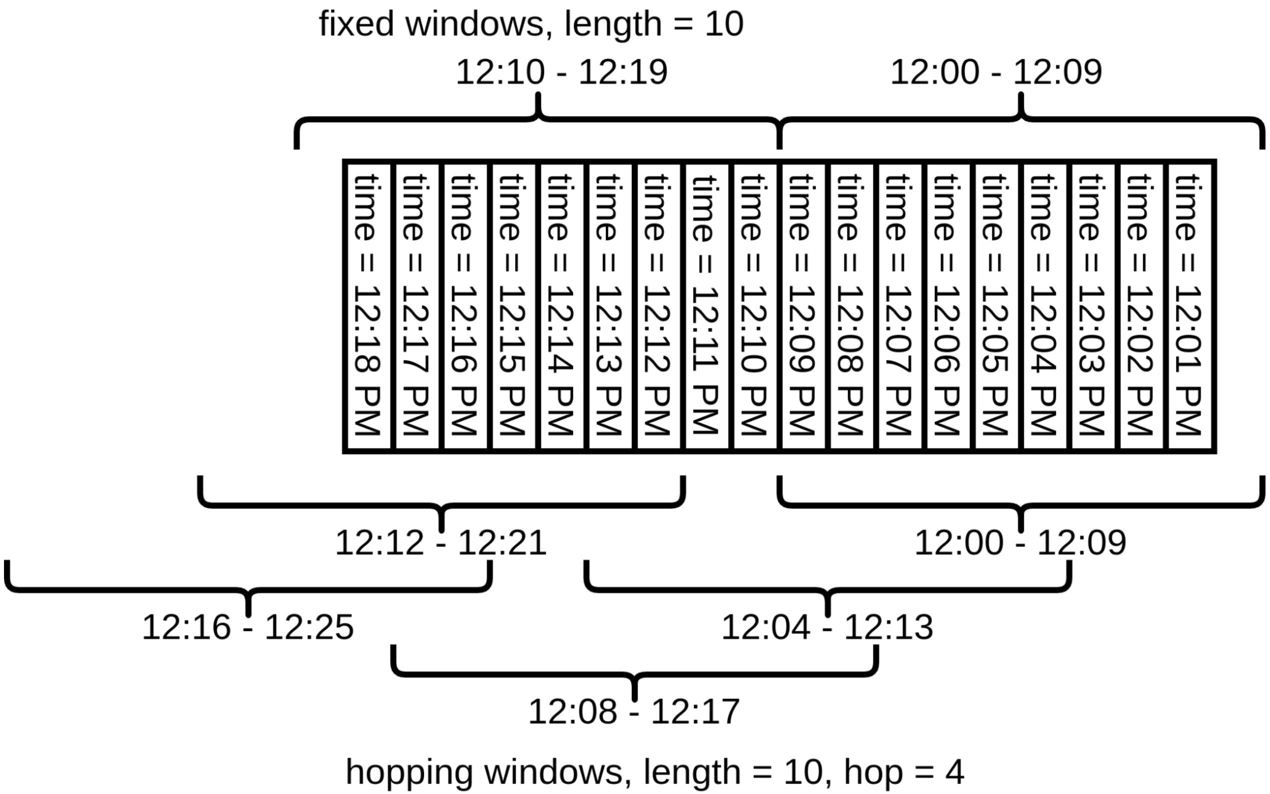


Time windows

Tumbling (fixed)

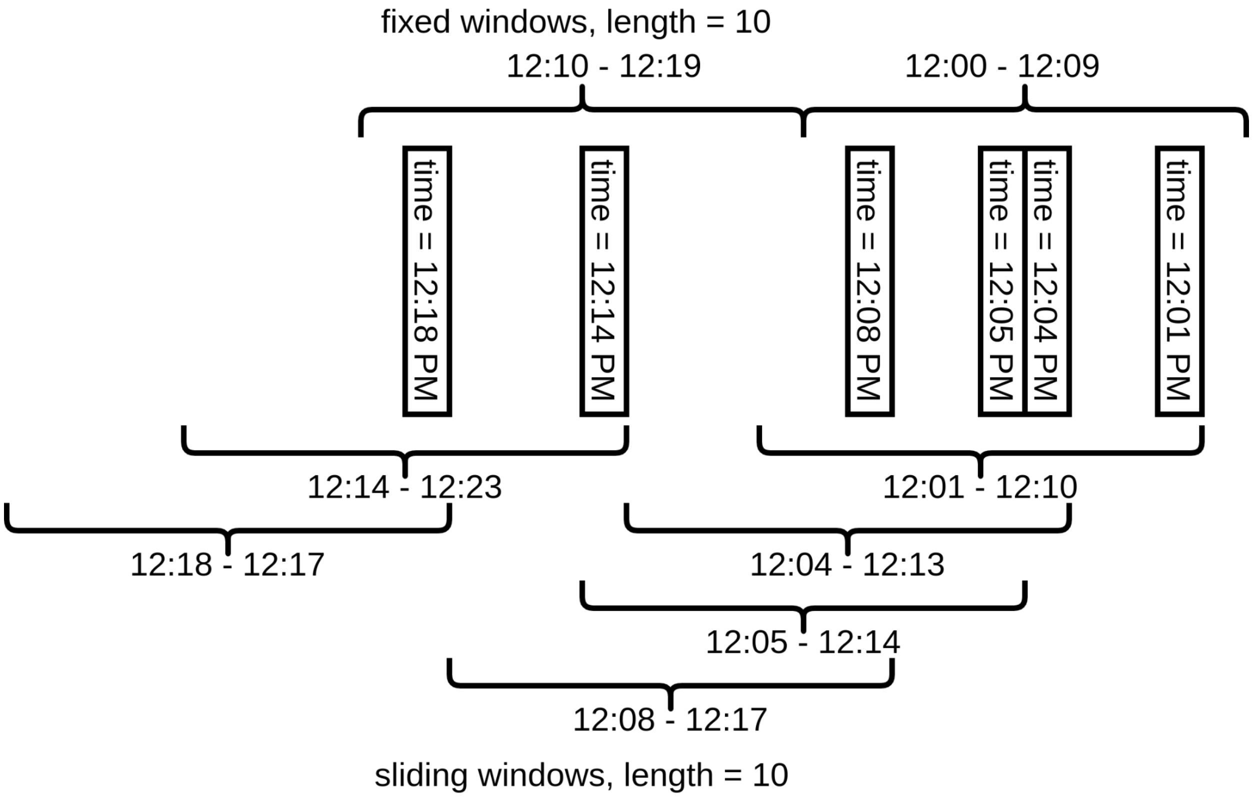


Hopping

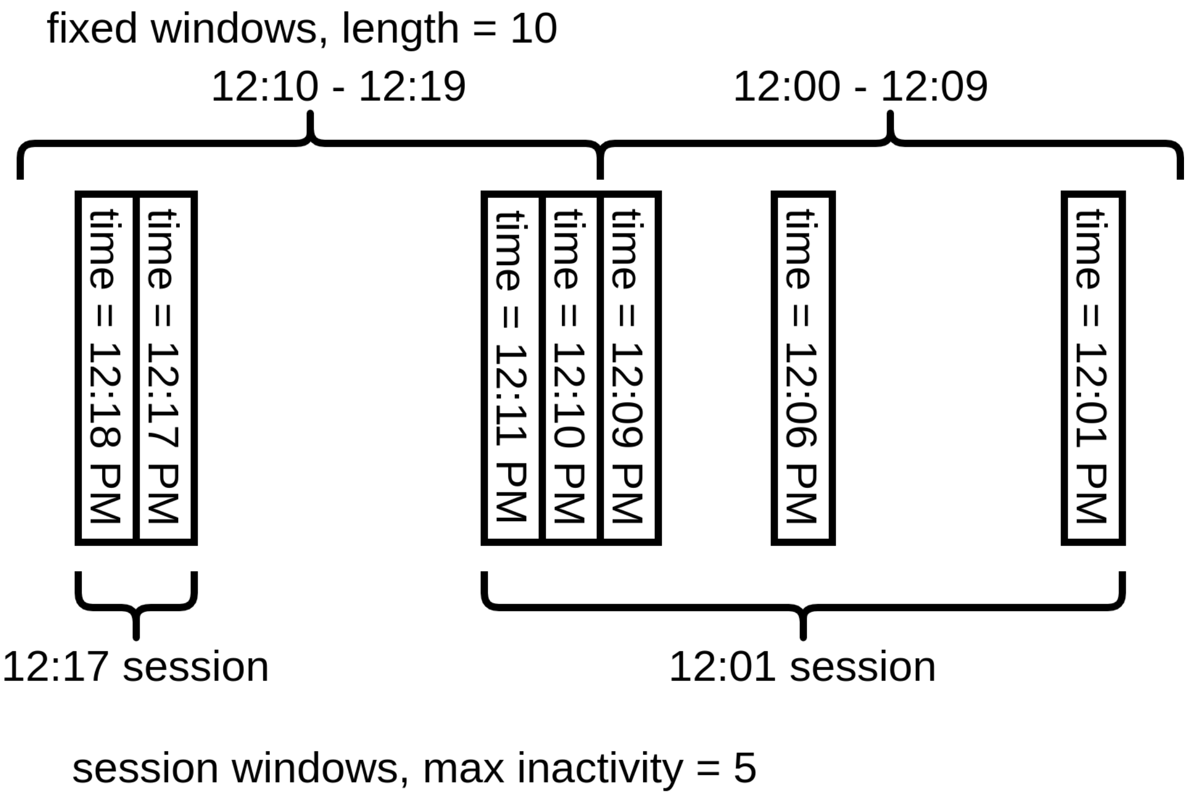


fixed windows, length = 10
12:10 - 12:19
12:00 - 12:09
time = 12:18 PM
time = 12:17 PM
time = 12:16 PM
time = 12:15 PM
time = 12:14 PM
time = 12:13 PM
time = 12:12 PM
time = 12:11 PM
time = 12:10 PM
time = 12:09 PM
time = 12:08 PM
time = 12:07 PM
time = 12:06 PM
time = 12:05 PM
time = 12:04 PM
time = 12:03 PM
time = 12:02 PM
time = 12:01 PM
12:12 - 12:21
12:00 - 12:09
12:16 - 12:25
12:04 - 12:13
12:08 - 12:17
hopping windows, length = 10, hop = 4

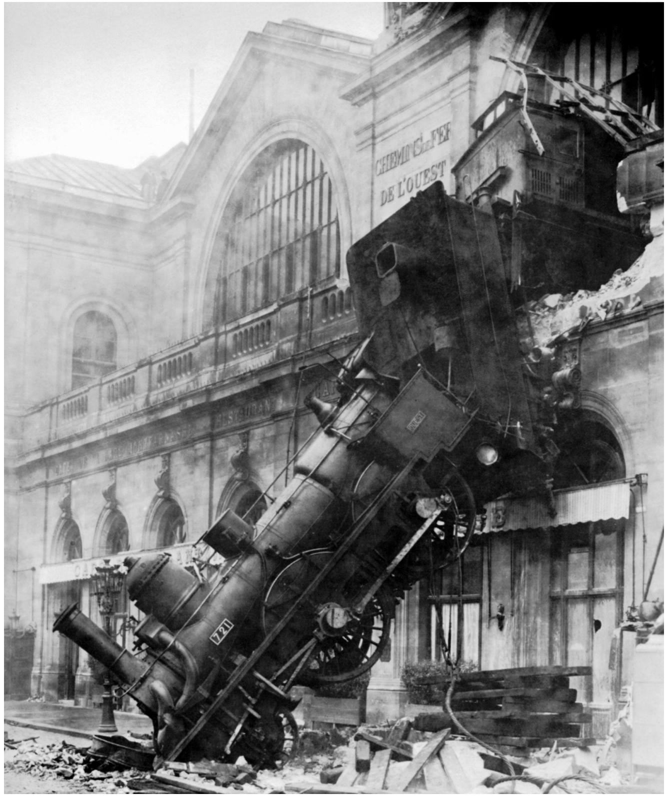
Sliding


fixed windows, length = 10
12:10 - 12:19
12:00 - 12:09
time = 12:18 PM
time = 12:14 PM
time = 12:08 PM
time = 12:05 PM
time = 12:04 PM
time = 12:01 PM
12:14 - 12:23
12:01 - 12:10
12:18 - 12:17
12:04 - 12:13
12:05 - 12:14
12:08 - 12:17
sliding windows, length = 10

Session

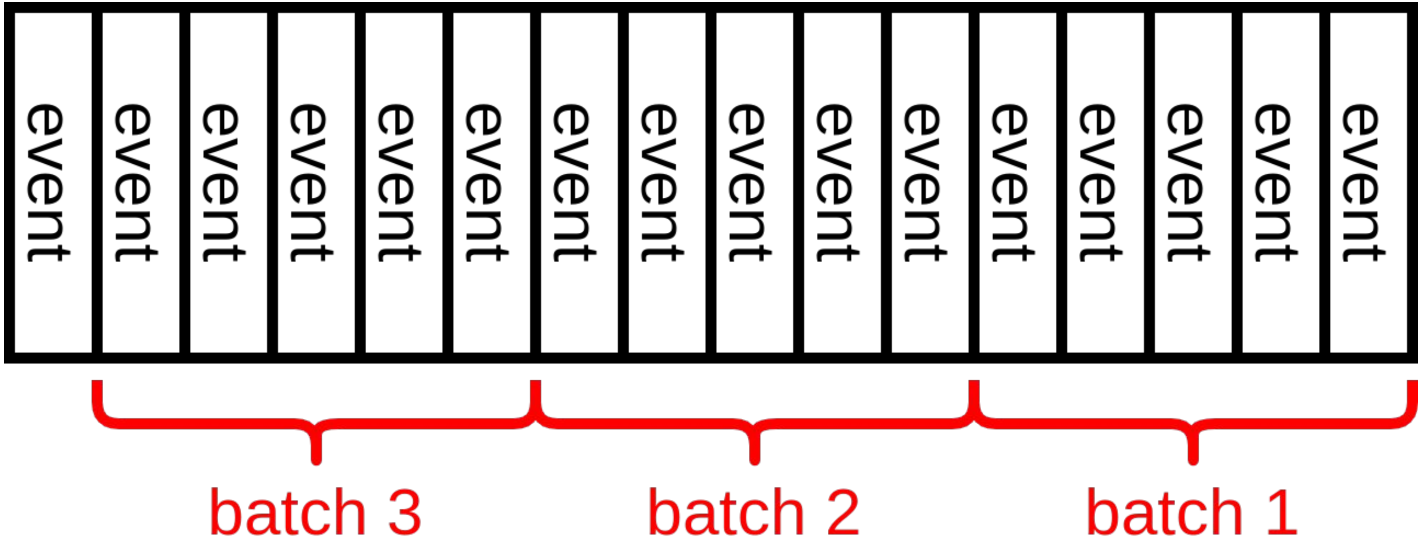

fixed windows, length = 10
12:10 - 12:19
12:00 - 12:09
time = 12:18 PM
time = 12:17 PM
time = 12:11 PM
time = 12:10 PM
time = 12:09 PM
time = 12:06 PM
time = 12:01 PM
12:17 session
12:01 session
session windows, max inactivity = 5

Fault tolerance



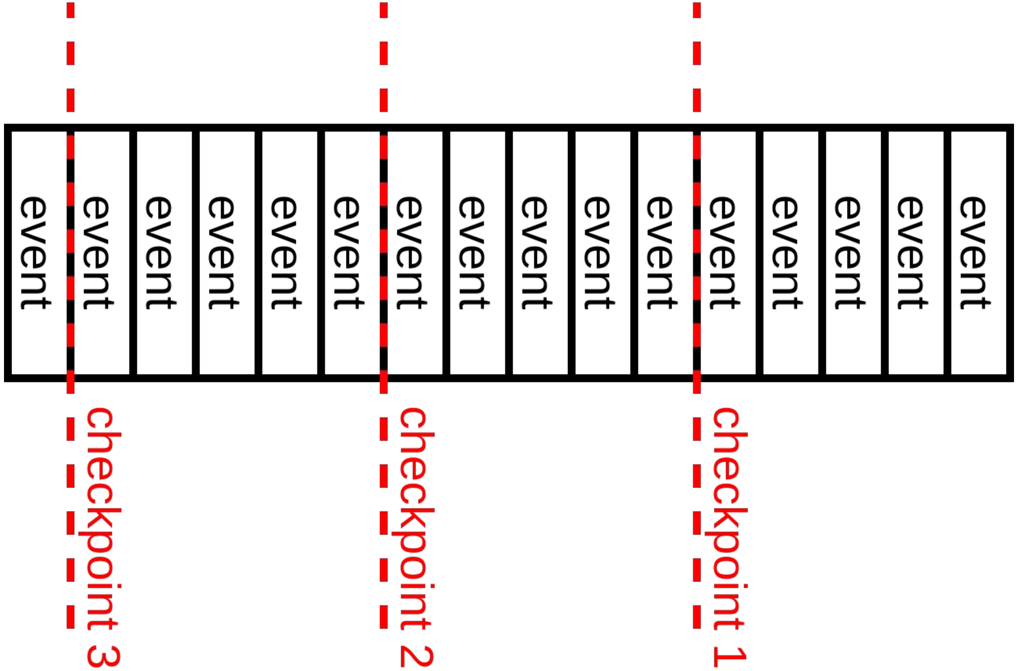
Microbatching

- break the stream into small blocks, and treat each block as a batch process



Checkpointing

- periodically generate a checkpoint of state and write it to persistent storage



Atomic commit

- Kafka transactions are of limited use
- X/Open XA (eXtended architecture): global transactions across heterogeneous components, based on two-phase commit protocol
- some large systems support transactions within their environment (e.g. Google Cloud Dataflow)

Idempotence

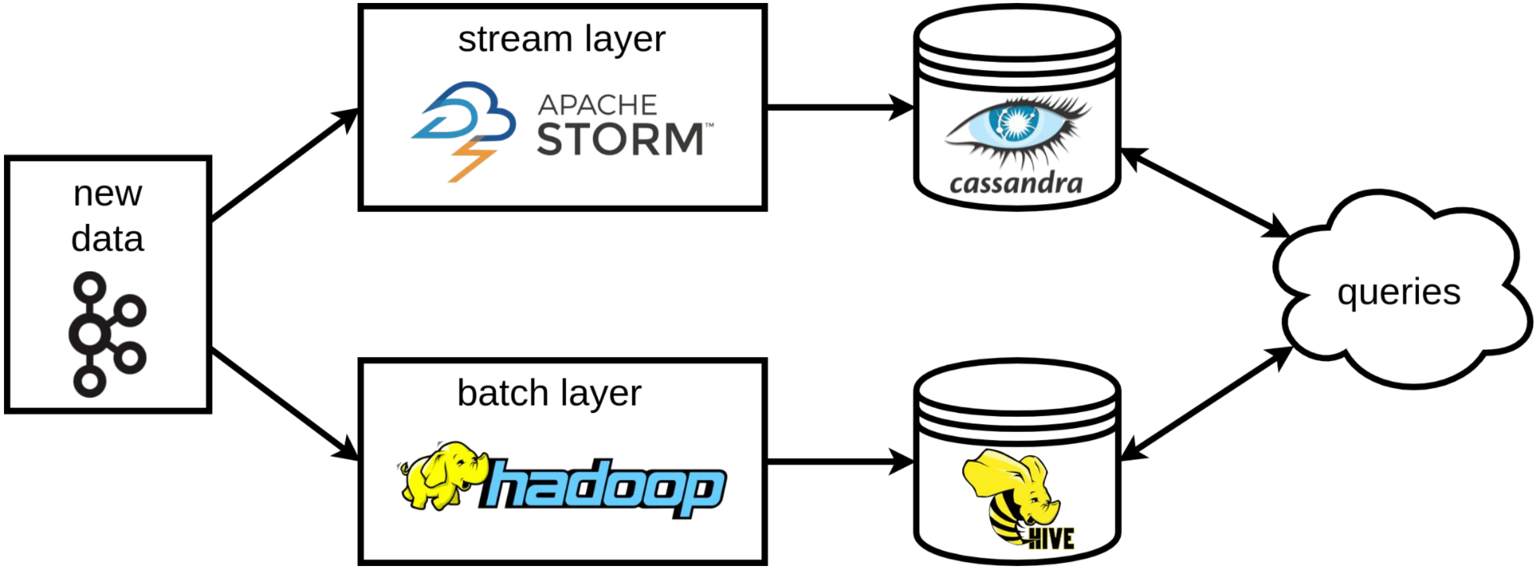
- idempotent operation performed multiple times has the same effect as performed once
- example: “UPDATE table SET processed = true WHERE event_id = 101”
- at least once + idempotent operations = effectively once

Rebuilding state

- applications must be able to restore the state after a failure
- option 1: keep the state in a remote database
- option 2: keep the state local, and replicate it periodically
- option 3: do nothing, and after a failure simply replay the input stream from start and rebuild the state

Lambda architecture

- “How to beat the CAP theorem” - Nathan Marz, 2011



Benefits & challenges

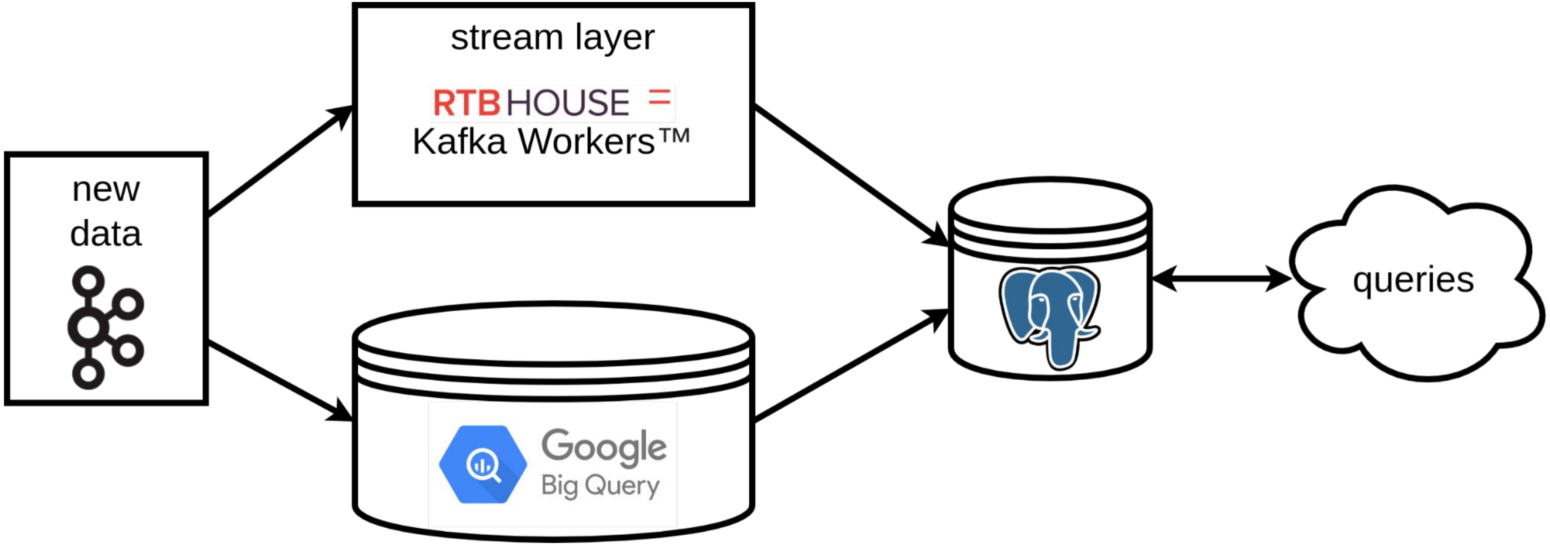
Positives:

- best of two worlds: low latency and possibility to process historical data
- algorithmic flexibility
- easy ad-hoc analysis

Negatives:

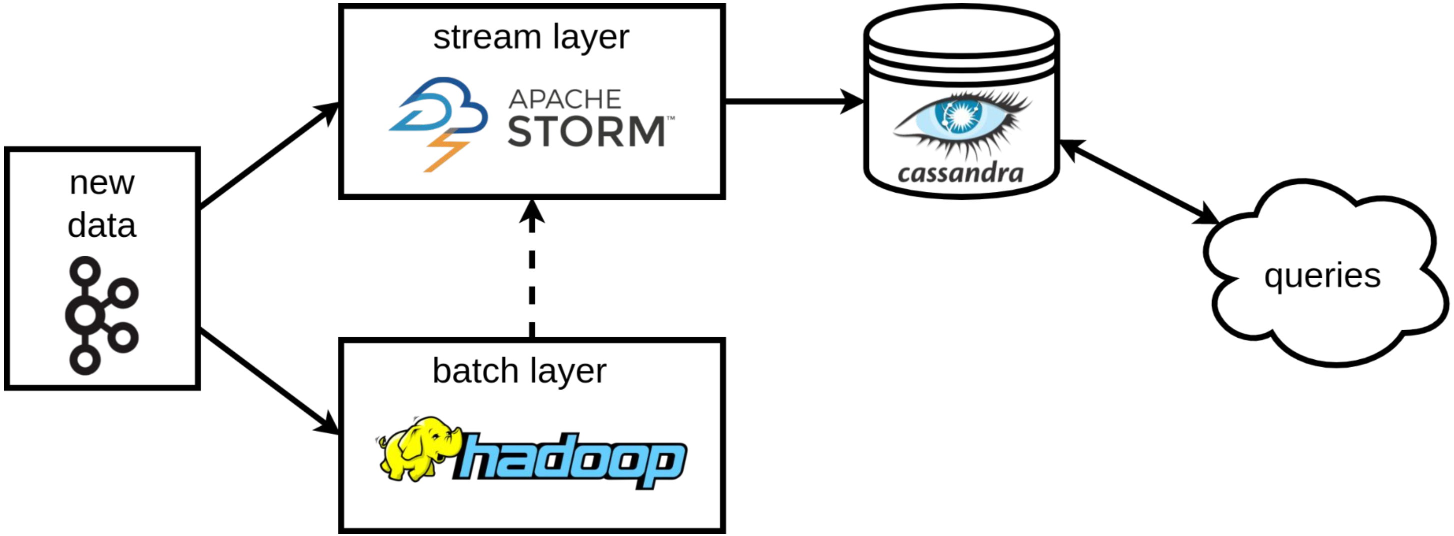
- inherent complexity
- maintaining two codebases
- synchronization between the layers is needed

Practical example: stats counter



Kappa architecture

- “Questioning the Lambda Architecture” - Jay Kreps, 2014



Summary

Two things to remember:

- strengths and weaknesses of data streams
- basic concepts and techniques