Stream Processing with Apache Flink

QCon London,
March 7, 2016

Robert Metzger
@rmetzger_
rmetzger@apache.org

Talk overview

- My take on the stream processing space, and how it changes the way we think about data
- Discussion of unique building blocks of Flink
- Benchmarking Flink, by extending a benchmark from Yahoo!

Apache Flink

- Apache Flink is an open source stream processing framework
  - Low latency
  - High throughput
  - Stateful
  - Distributed
- Developed at the Apache Software Foundation, 1.0.0 release available soon, used in production

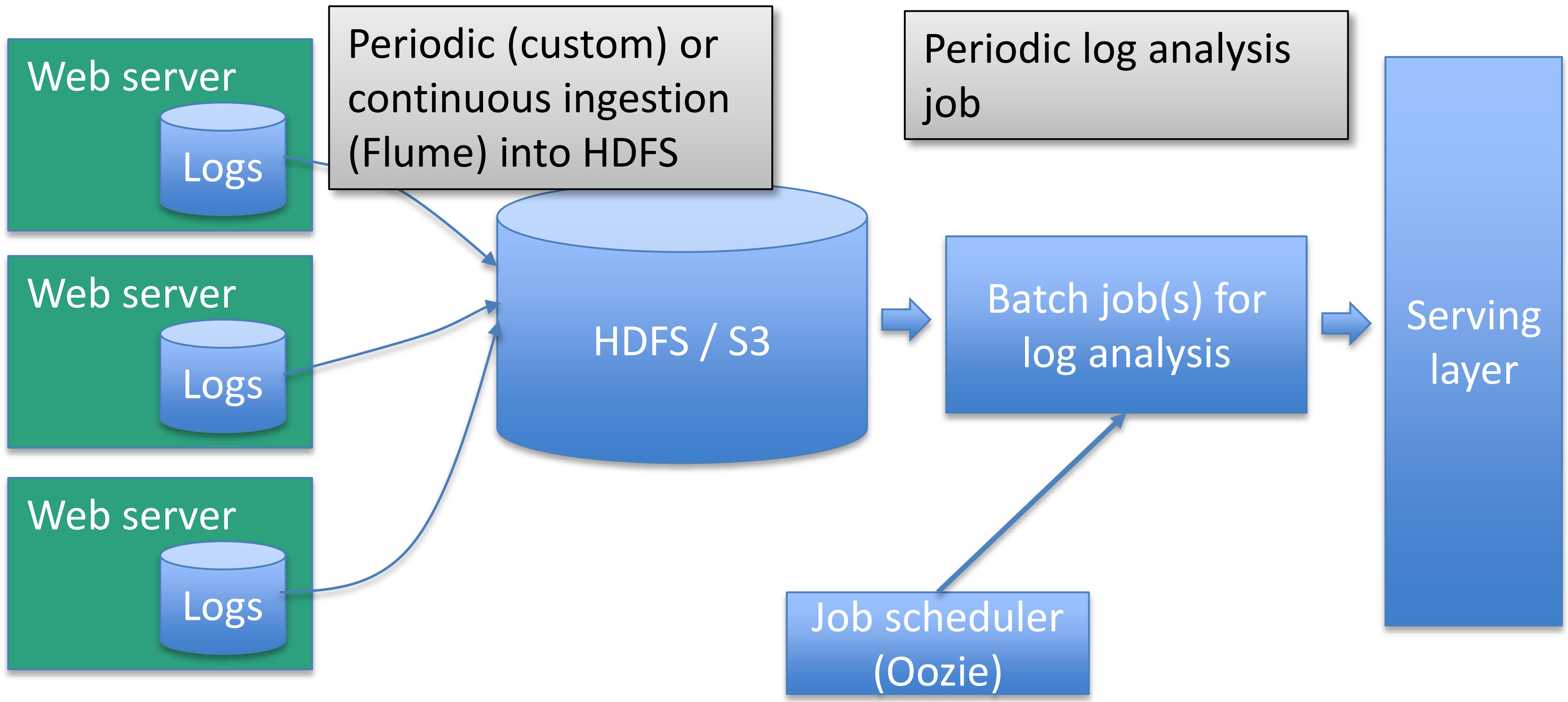
Entering the streaming era

Streaming is the biggest change in data infrastructure since Hadoop

1. Radically simplified infrastructure
2. Do more with your data, faster
3. Can completely subsume batch

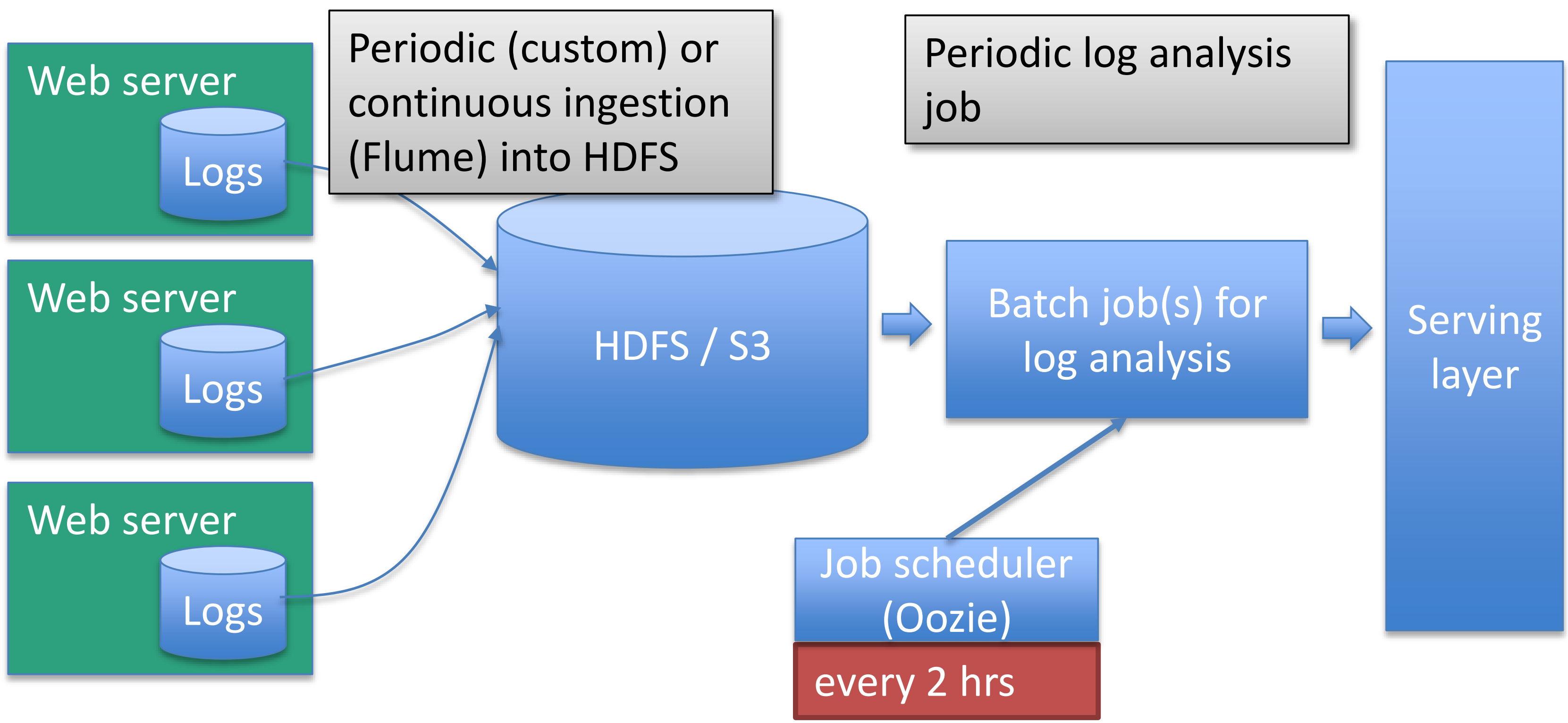
Traditional data processing

- Log analysis example using a batch processor



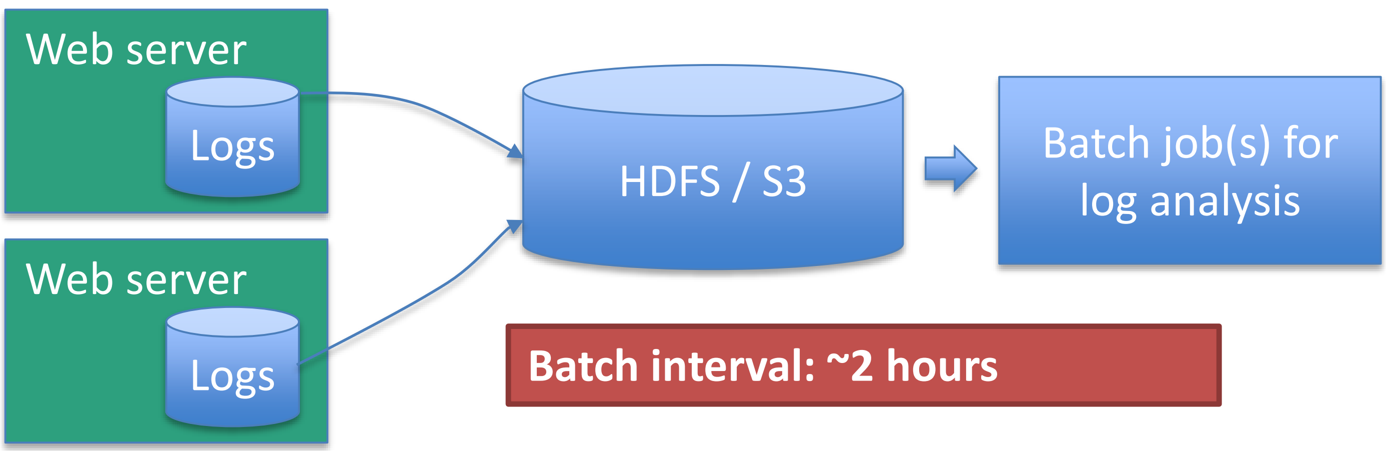
Traditional data processing

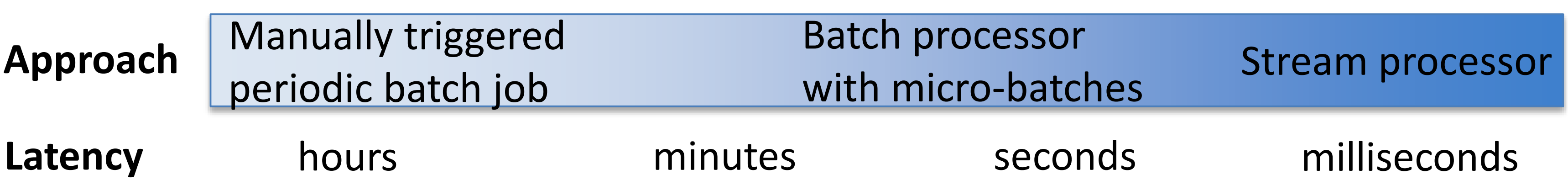
- **Latency** from log event to serving layer usually in the **range of hours**



Data processing without stream processor

- This architecture is a hand-crafted micro-batch model





Downsides of stream processing with a batch engine

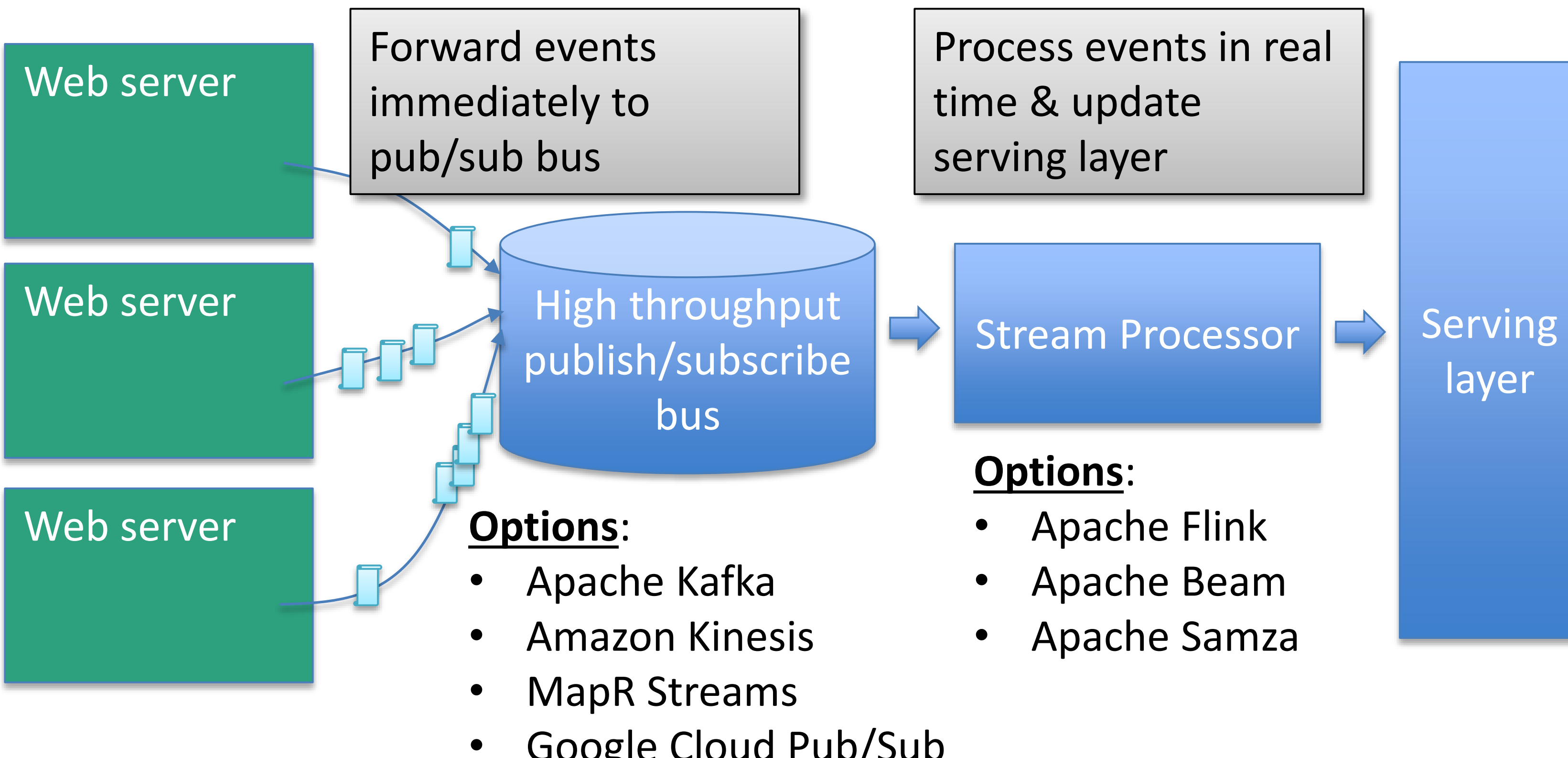
- Very high latency (hours)
- Complex architecture required:
  - Periodic job scheduler (e.g. Oozie)
  - Data loading into HDFS (e.g. Flume)
  - Batch processor
  - (When using the “lambda architecture”: a stream processor)

→ All these components need to be implemented and maintained

- Backpressure: How does the pipeline handle load spikes?

Log event analysis using a stream processor

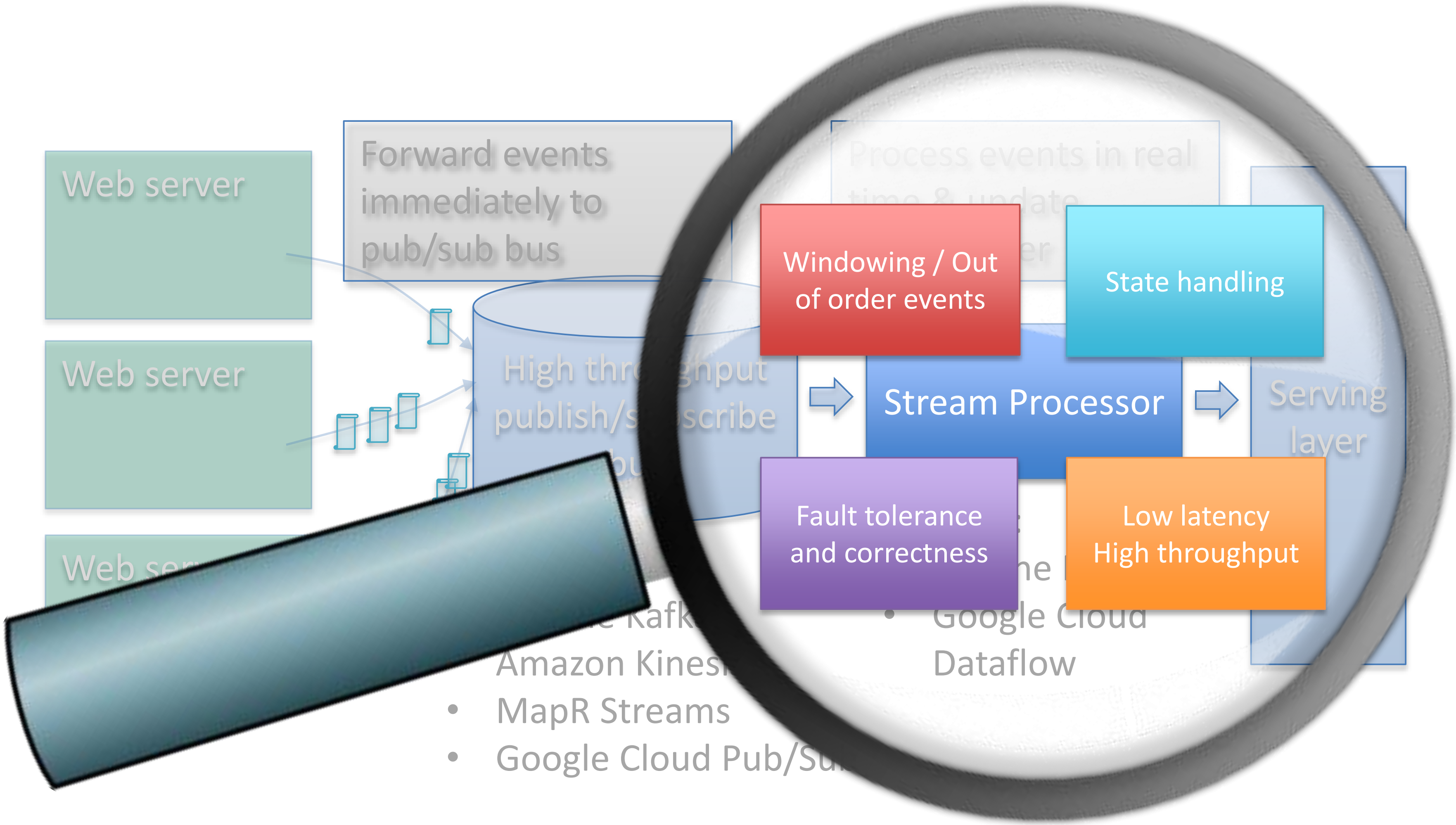
- Stream processors allow to analyze events with **sub-second latency**.



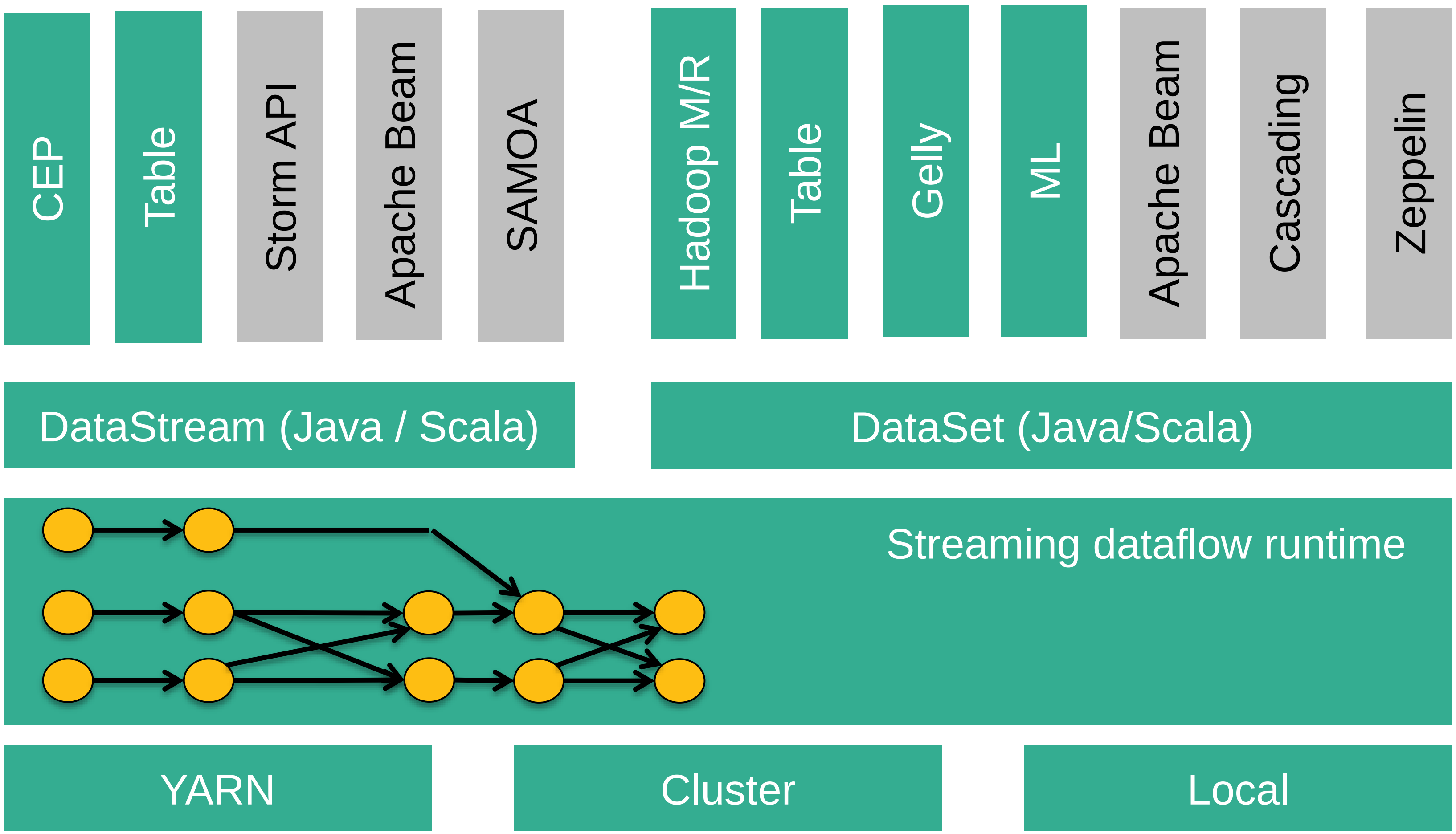
Real-world data is produced in a continuous fashion.

New systems like Flink and Kafka embrace streaming nature of data.

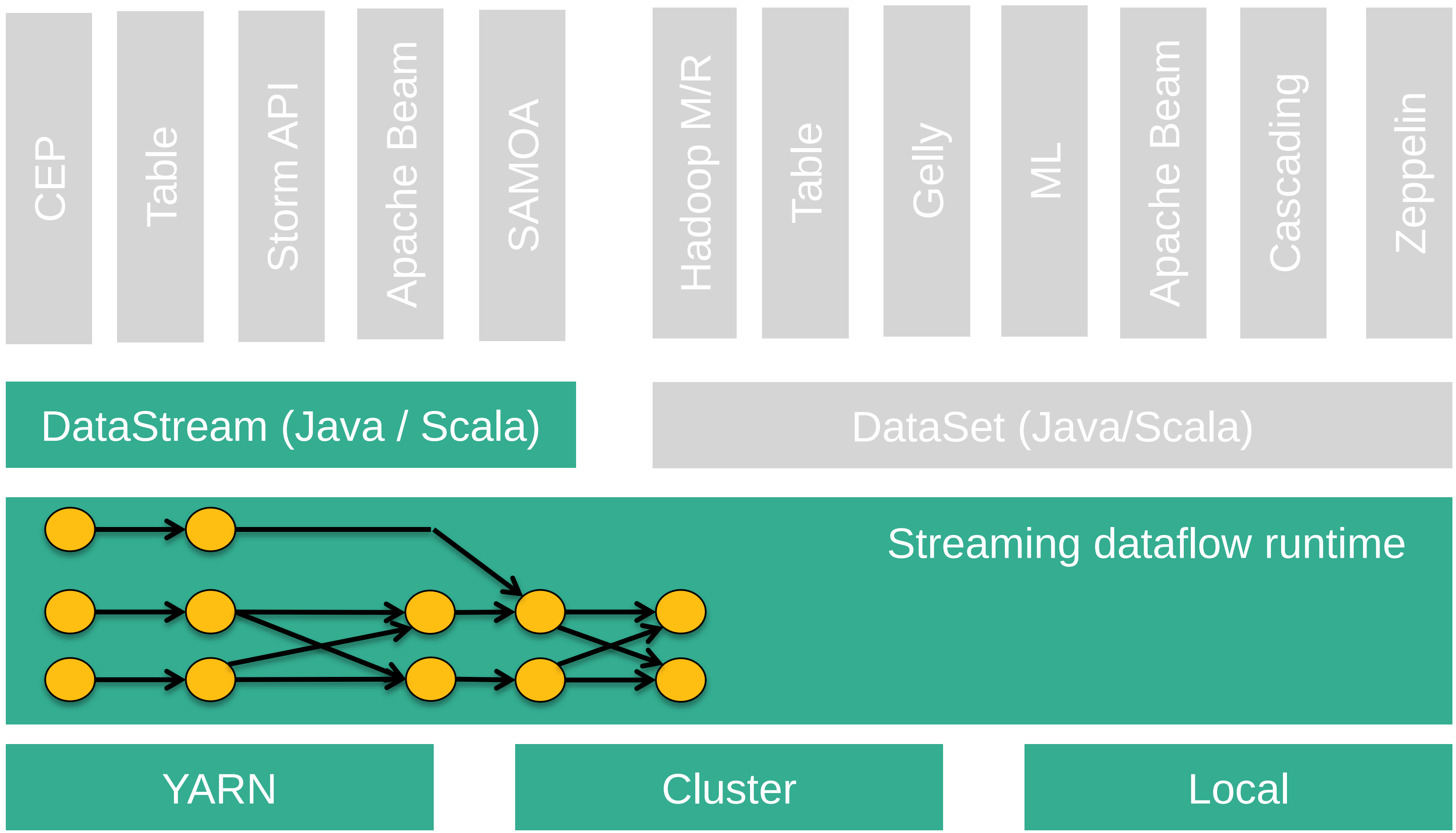
What do we need for replacing the “batch stack”?



Apache Flink stack



Needed for the use case



Windowing / Out of order events

Windowing / Out of order events

State handling

Fault tolerance and correctness

Low latency
High throughput

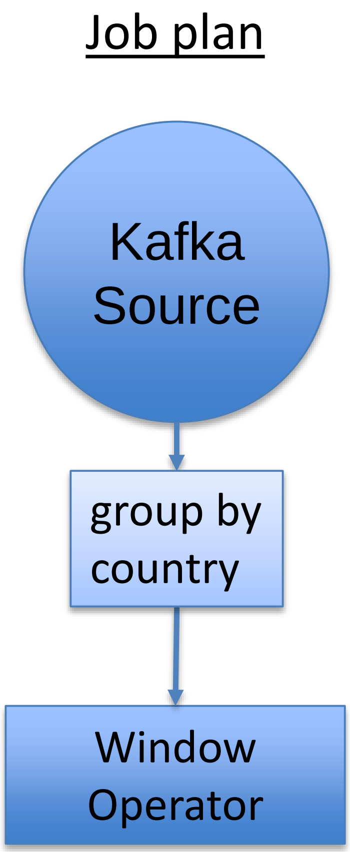
Building windows from a stream

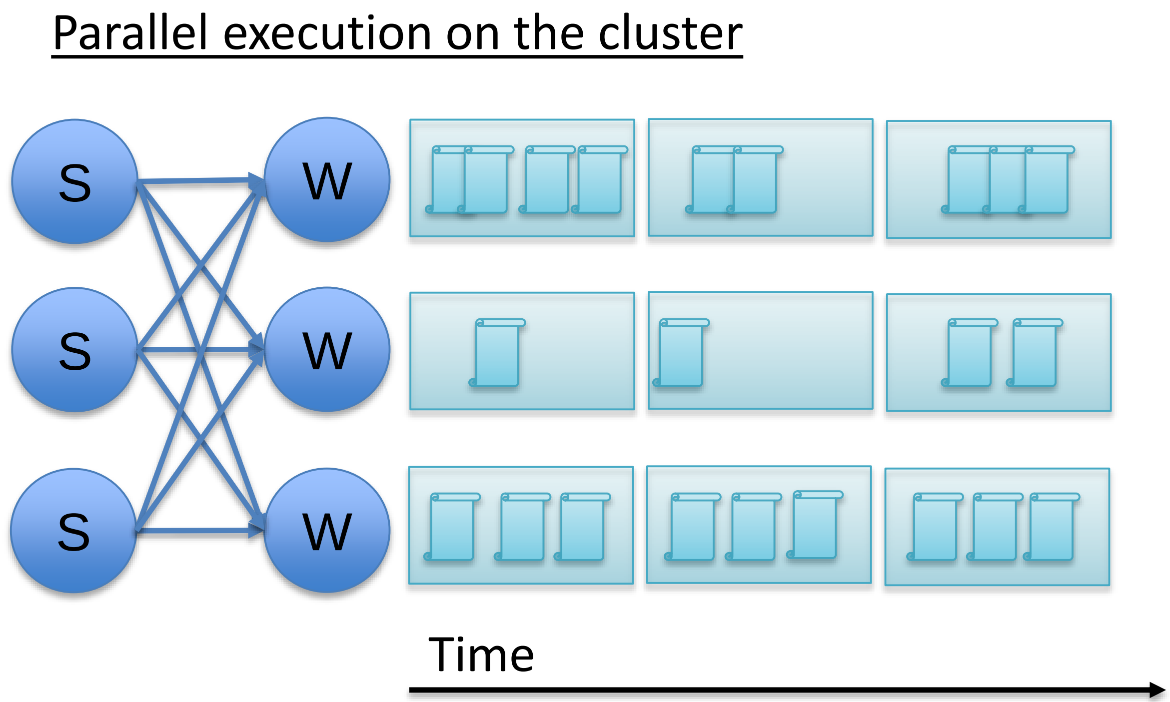
- *“Number of visitors in the last 5 minutes per country”*

```
// create stream from Kafka source
DataStream<LogEvent> stream = env.addSource(new KafkaConsumer());
// group by country
DataStream<LogEvent> keyedStream = stream.keyBy(“country“);
// window of size 5 minutes
keyedStream.timeWindow(Time.minutes(5))
// do operations per window
.apply(new CountPerWindowFunction());
```

Building windows: Execution

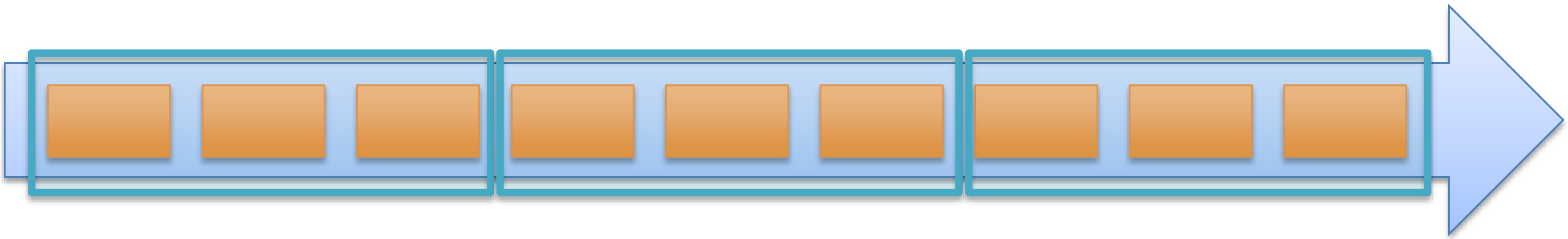
```
// window of size 5 minutes
keyedStream.timeWindow(Time.minutes(5));
```



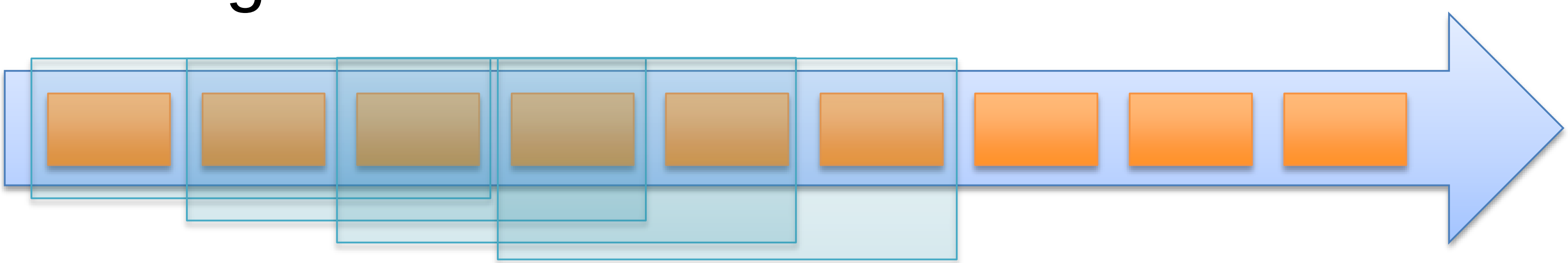


Window types in Flink

- Tumbling windows



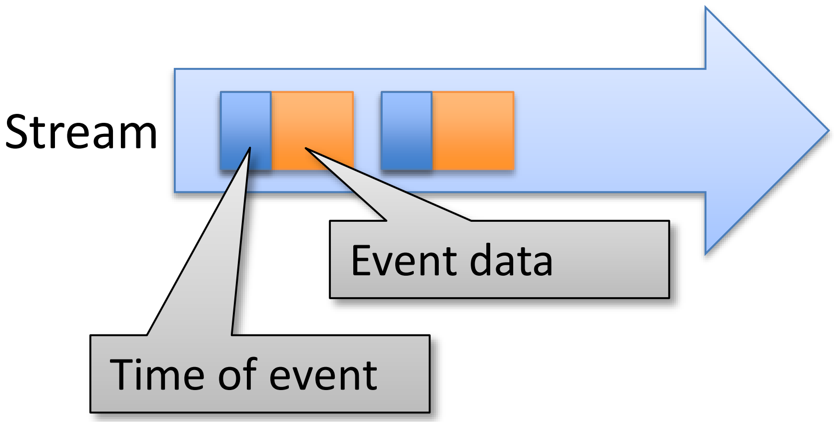
- Sliding windows



- Custom windows with window assigners, triggers and evictors

Further reading: http://flink.apache.org/news/2015/12/04/Introducing-windows.html

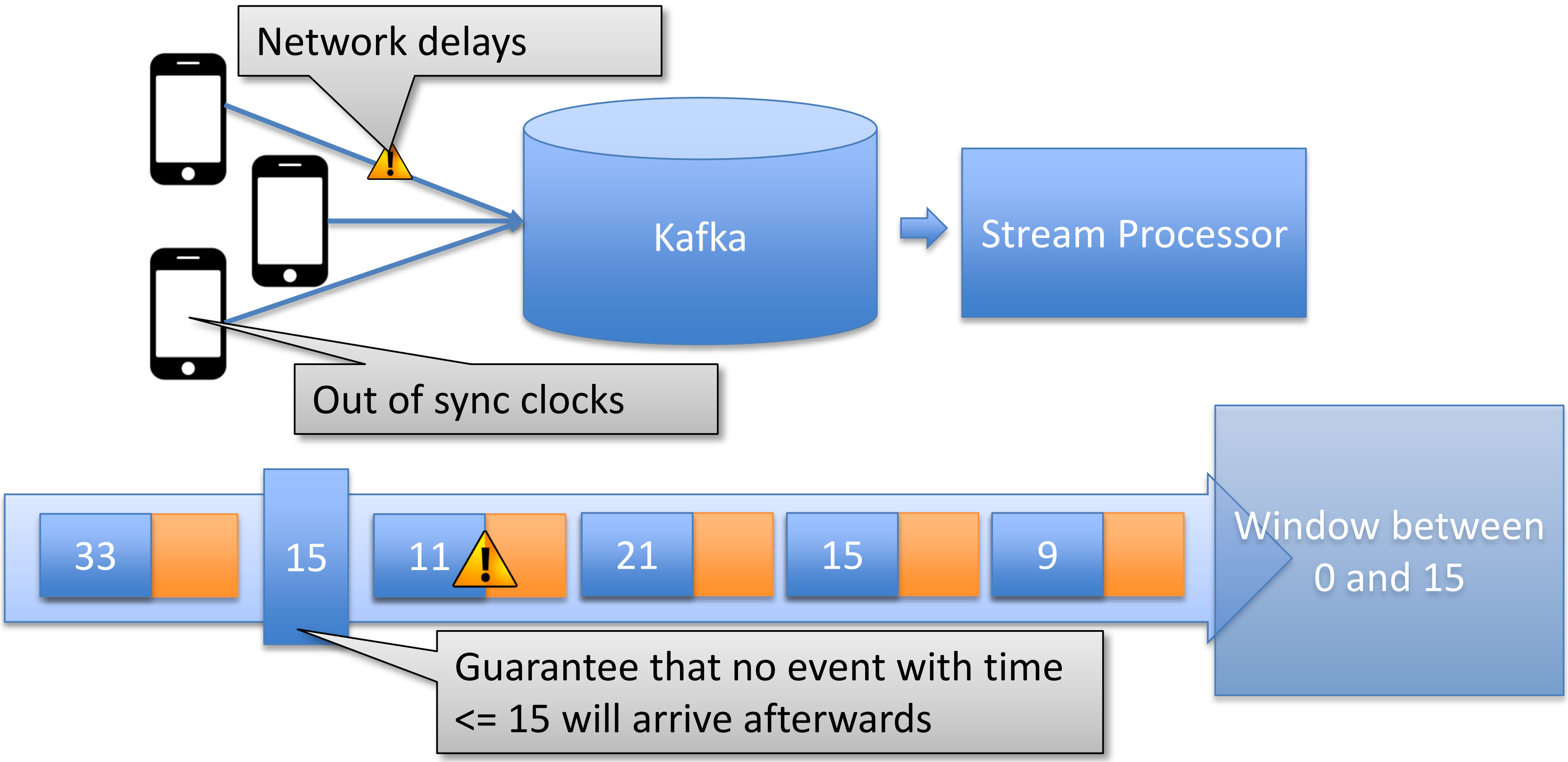
Time-based windows




```
{
    “accessTime”: “1457002134”,
    “userId”: “1337”,
    “userLocation”: “UK”
}
```

→ Windows are created based on the real world time when the event occurred

A look at the reality of time



- Events arrive **out of order** in the system
- Use-case specific **low watermarks** for time tracking

Time characteristics in Apache Flink

- Event Time
  - Users have to specify an event-time extractor + watermark emitter
  - Results are deterministic, but with latency
- Processing Time
  - System time is used when evaluating windows
  - low latency
- Ingestion Time
  - Flink assigns current system time at the sources
- Pluggable, without window code changes

State handling

Windowing / Out of order events

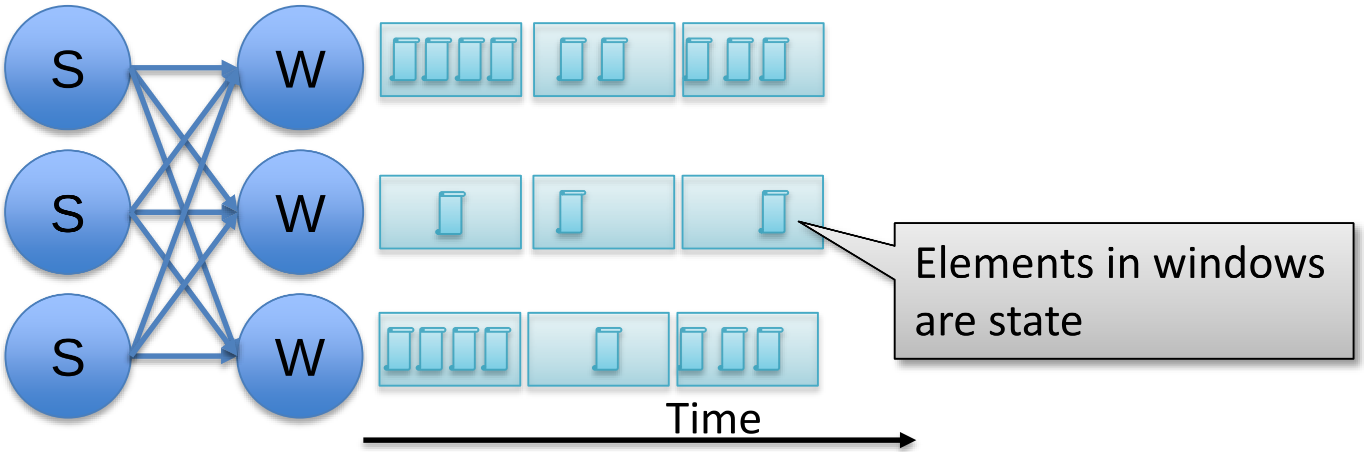
State handling

Fault tolerance and correctness

Low latency High throughput

State in streaming

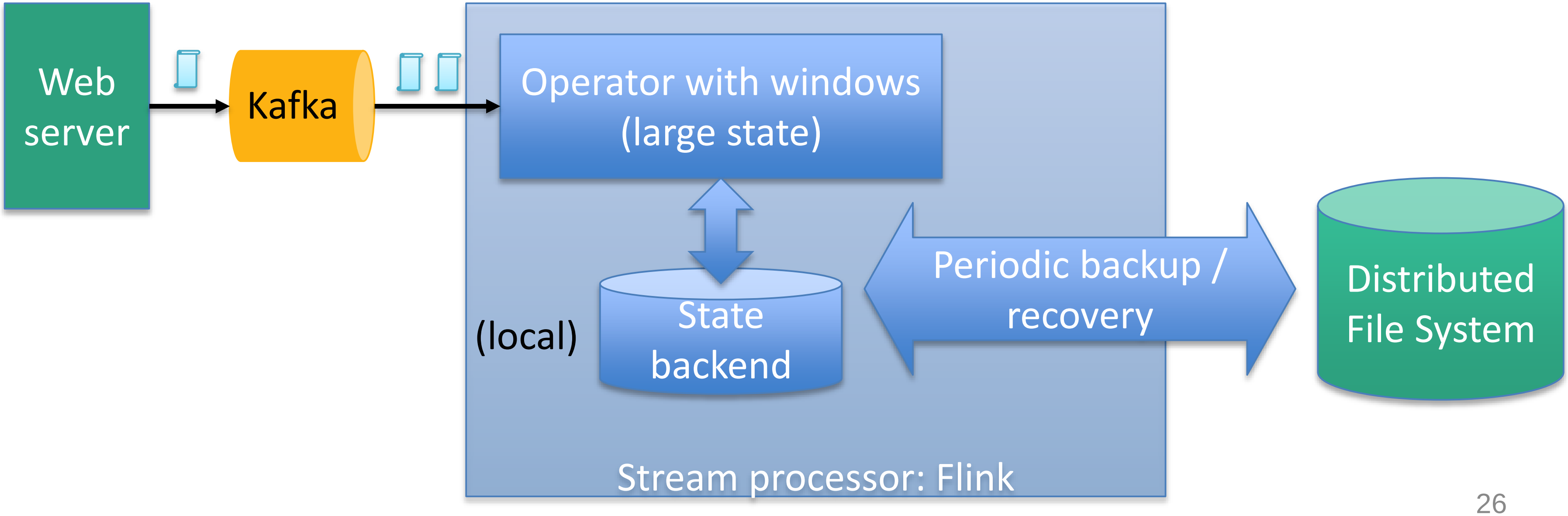
- Where do we store the elements from our windows?



- In stateless systems, an external state store (e.g. Redis) is needed.

Managed state in Flink

- Flink automatically backups and restores state
- State can be larger than the available memory
- State backends: (embedded) RocksDB, Heap memory



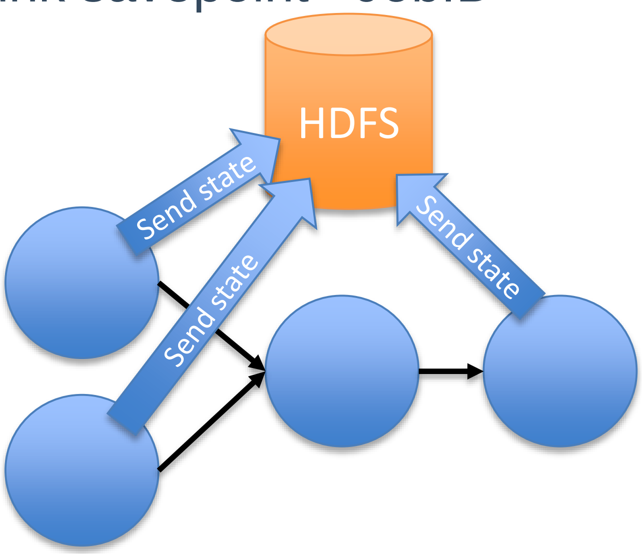
Managing the state

- How can we operate such a pipeline 24x7?
- Losing state (by stopping the system) would require a replay of past events
- We need a way to store the state somewhere!

Savepoints: Versioning state

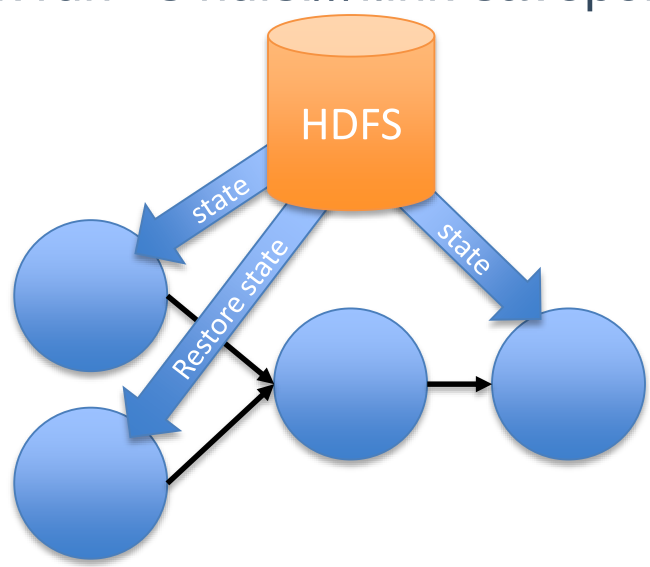
- Savepoint: Create an addressable copy of a job’s current state.
- Restart a job from any savepoint.

> flink savepoint <JobID>



> hdfs:///flink-savepoints/2

> flink run –s hdfs:///flink-savepoints/2 <jar>



Further reading: http://data-artisans.com/how-apache-flink-enables-new-streaming-applications/

Fault tolerance and correctness

Windowing / Out of order events

State handling

Fault tolerance and correctness

Low latency High throughput

Fault tolerance in streaming

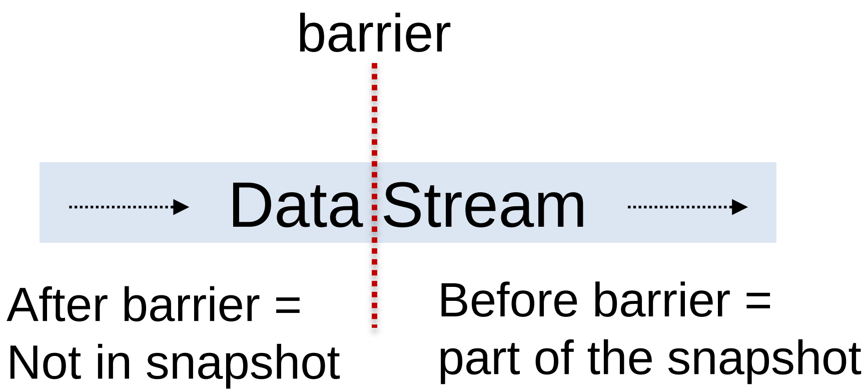
- How do we ensure the results (number of visitors) are always correct?
- Failures should not lead to data loss or incorrect results

Fault tolerance in streaming

- **at least once:** ensure all operators see all events
  - Storm: Replay stream in failure case (acking of individual records)
- **Exactly once:** ensure that operators do not perform duplicate updates to their state
  - **Flink: Distributed Snapshots**
  - Spark: Micro-batches on batch runtime

Flink's Distributed Snapshots

- Lightweight approach of storing the state of all operators without pausing the execution
- Implemented using barriers flowing through the topology



Further reading: http://blog.acolyer.org/2015/08/19/asynchronous-distributed-snapshots-for-distributed-dataflows/

Wrap-up: Log processing example

- How to do something with the data? **Windowing**
- How does the system handle large windows? **Managed state**
- How do operate such a system 24x7? **Safepoints**
- How to ensure correct results across failures? **Checkpoints, Master HA**

Performance:
Low Latency & High Throughput

Windowing / Out of order events

State handling

Fault tolerance and correctness

Low latency
High throughput

Performance: Introduction

- Performance always depends on your own use cases, so test it yourself!
- We based our experiments on a recent benchmark published by Yahoo!
- They benchmarked Storm, Spark Streaming and Flink with a production use-case (counting ad impressions)

Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Yahoo! Benchmark

- Count ad impressions grouped by campaign
- Compute aggregates over a 10 second window
- Emit current value of window aggregates to Redis every second for query

Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Yahoo's Results

"Storm […] and Flink […] show sub-second latencies at relatively high throughputs with Storm having the lowest 99th percentile latency. Spark streaming 1.5.1 supports high throughputs, but at a relatively higher latency."

(Quote from the blog post's executive summary)

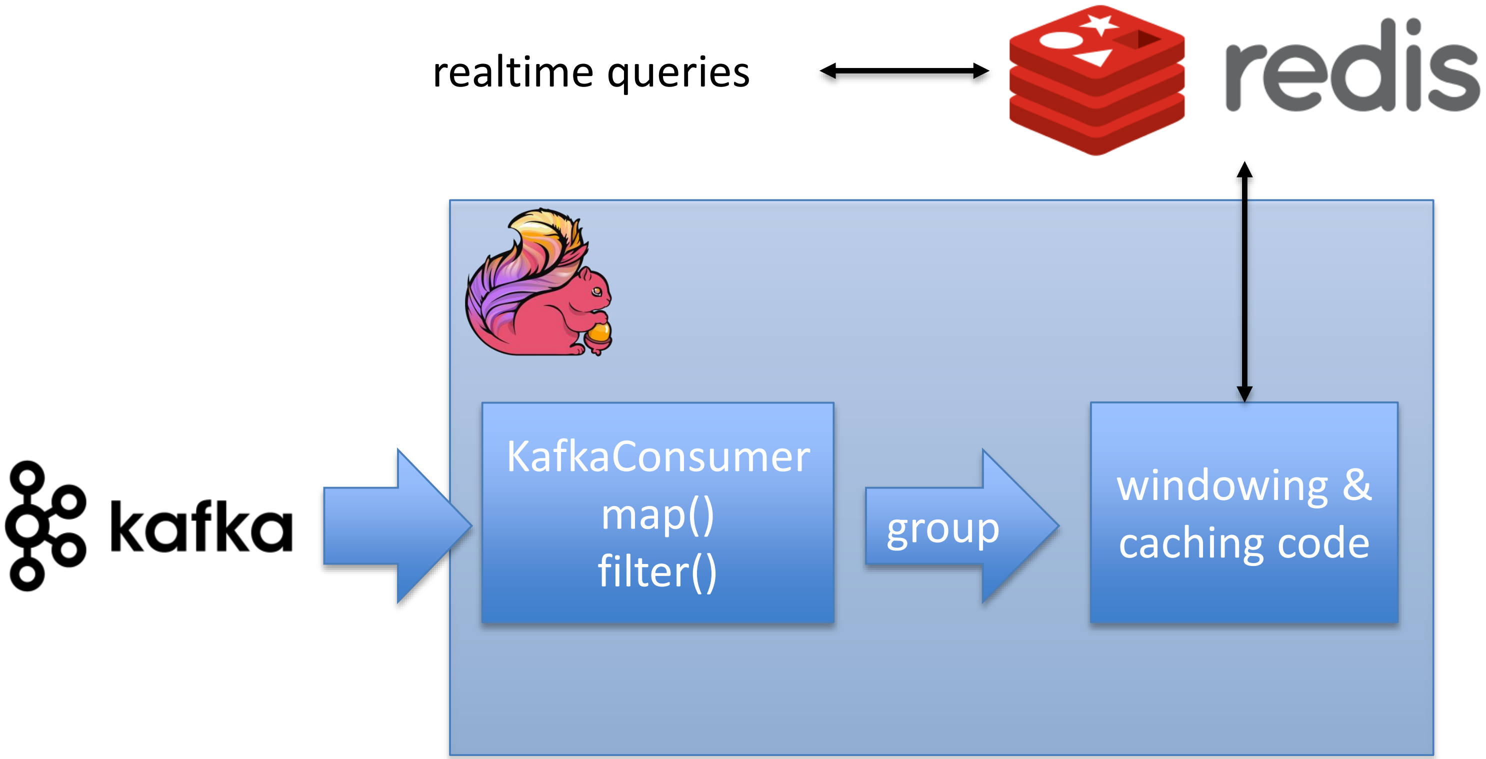
Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Extending the benchmark

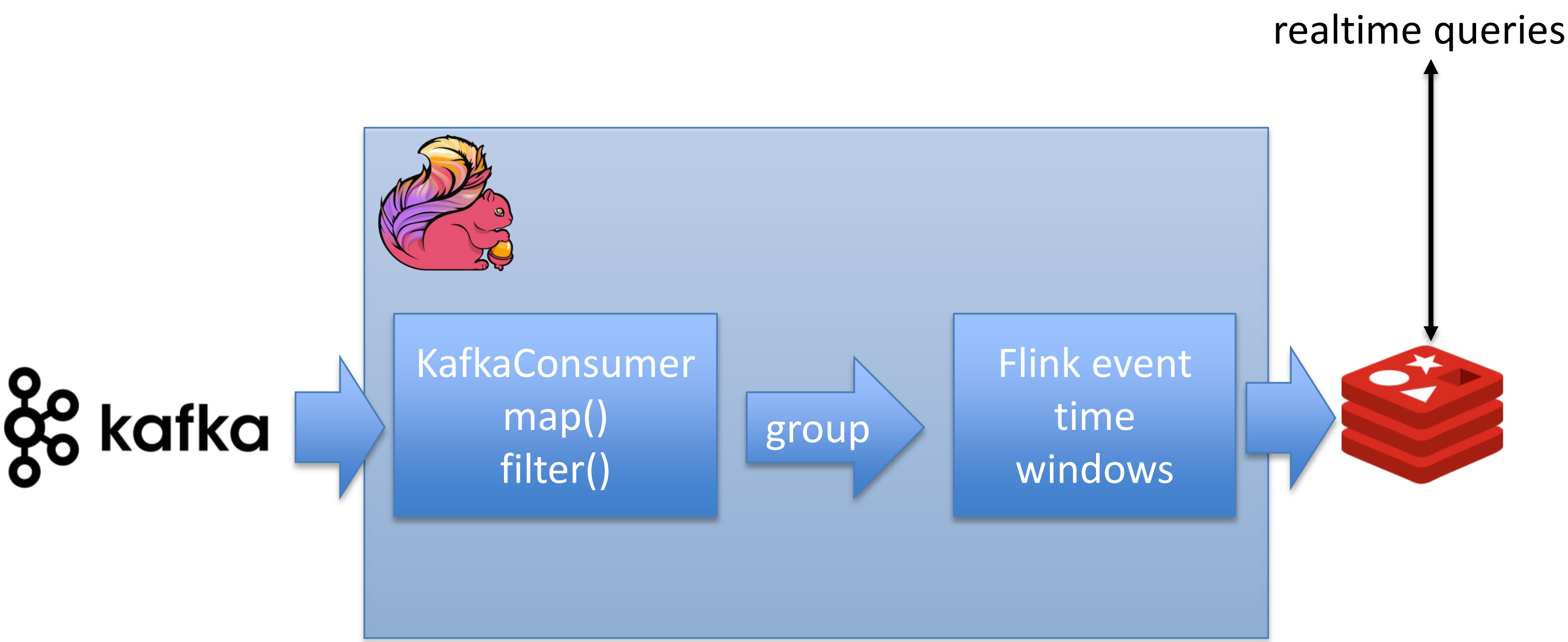
- Benchmark stops at Storm’s throughput limits. **Where is Flink’s limit?**
- How will Flink’s own window implementation perform compared to Yahoo’s “state in redis windowing” approach?

Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

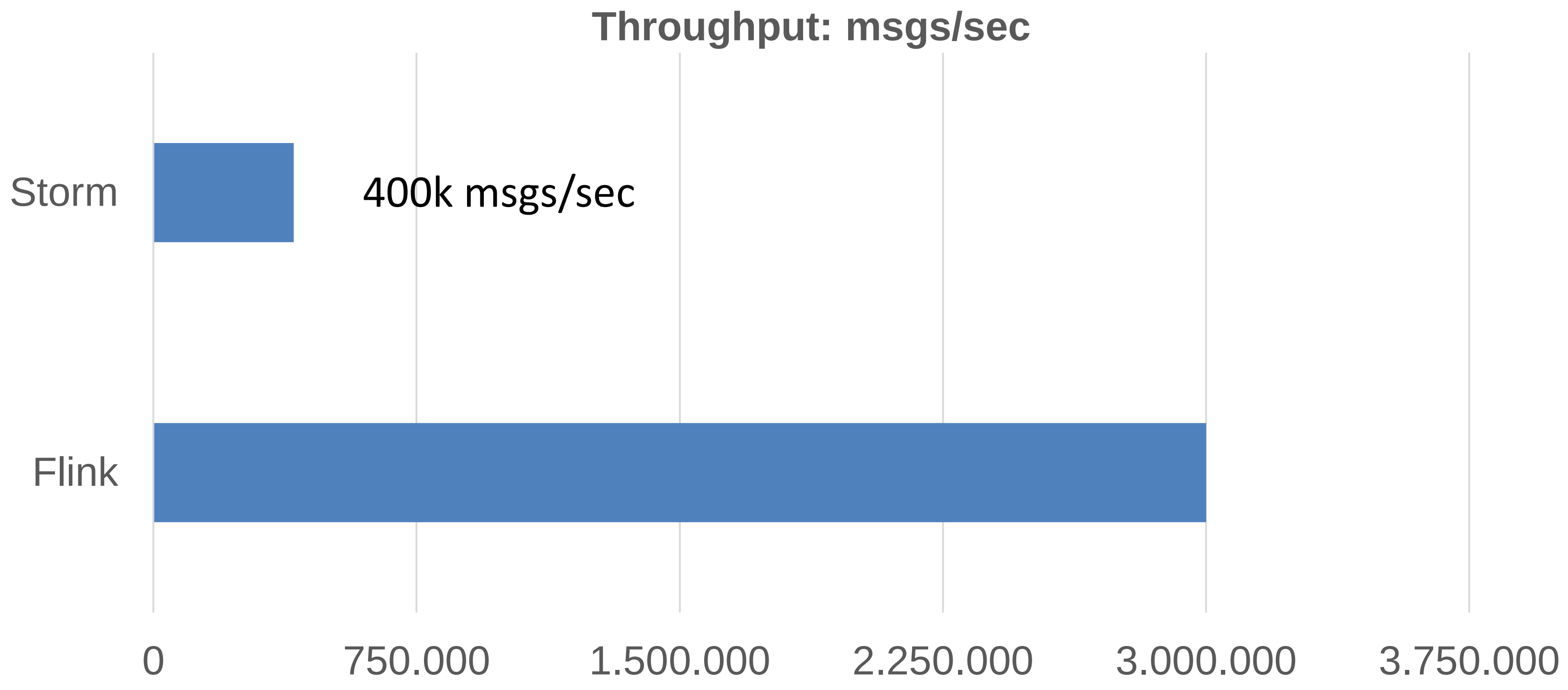
Windowing with state in Redis



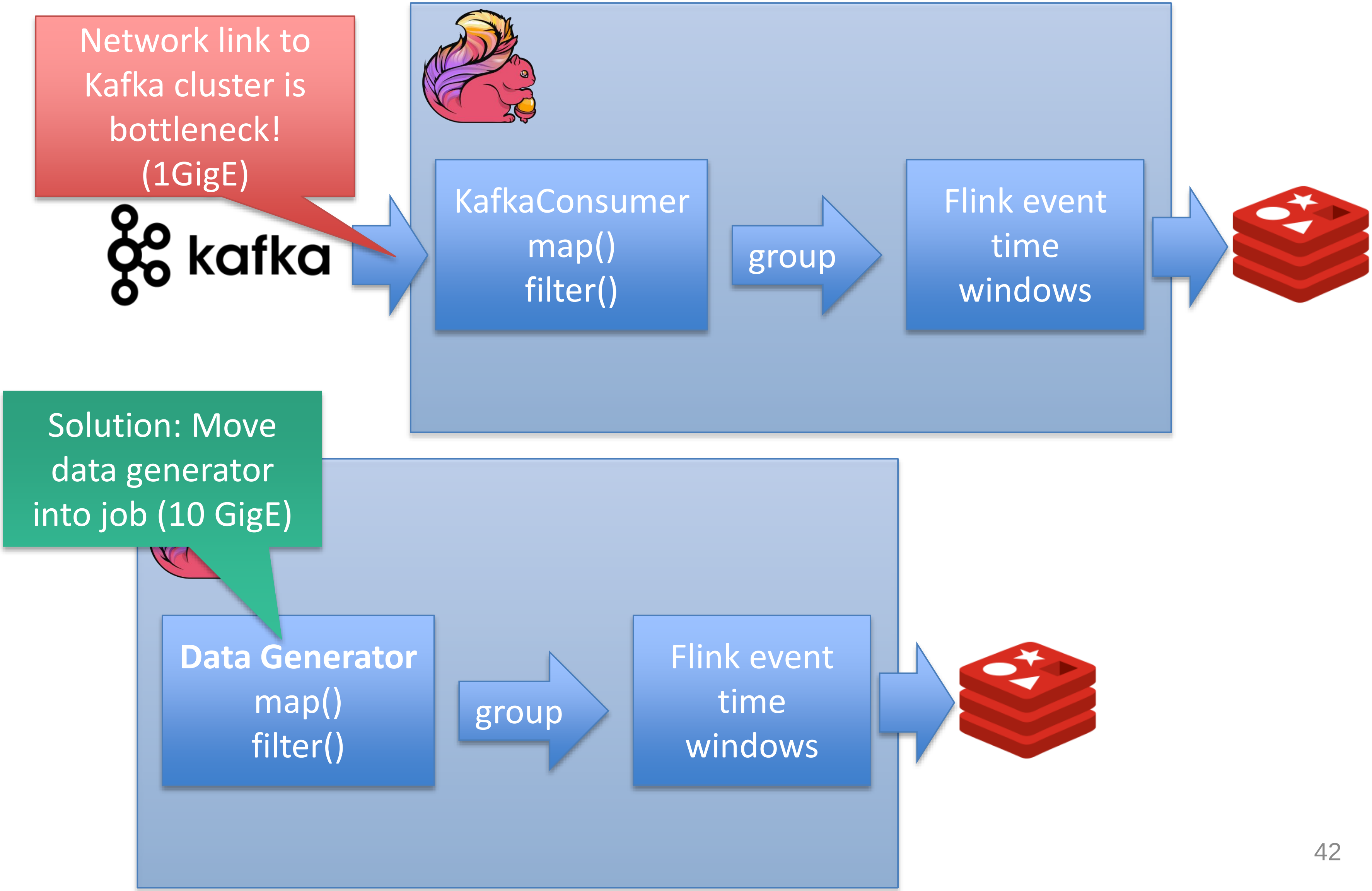
Rewrite to use Flink's own window



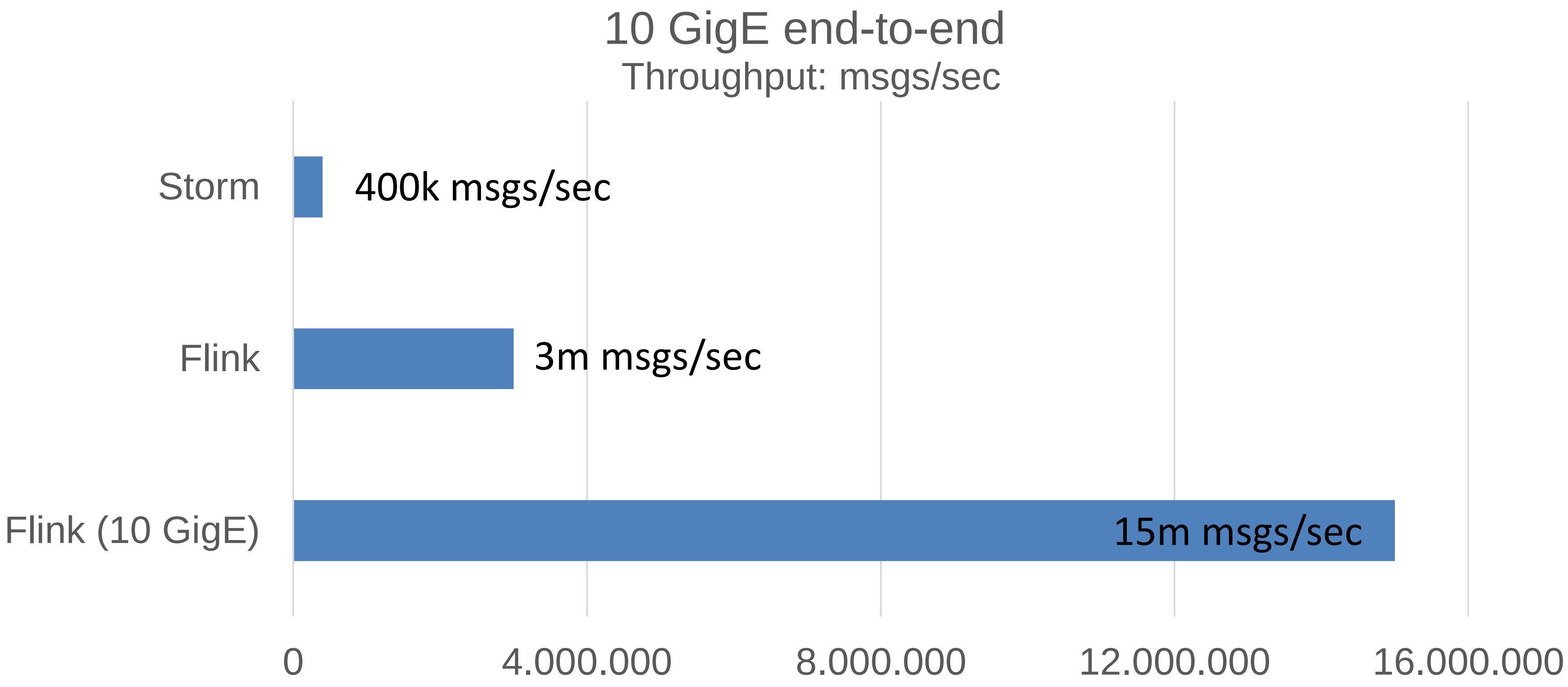
Results after rewrite



Can we even go further?



Results without network bottleneck



Benchmark summary

- Flink achieves **throughput of 15 million messages/second** on 10 machines
- **35x higher throughput** compared to Storm (80x compared to Yahoo's runs)
- Flink ran with **exactly once** guarantees, Storm with **at least once**.
- Read the full report: http://data-artisans.com/extending-the-yahoo-streaming-benchmark/

Closing

Other notable features

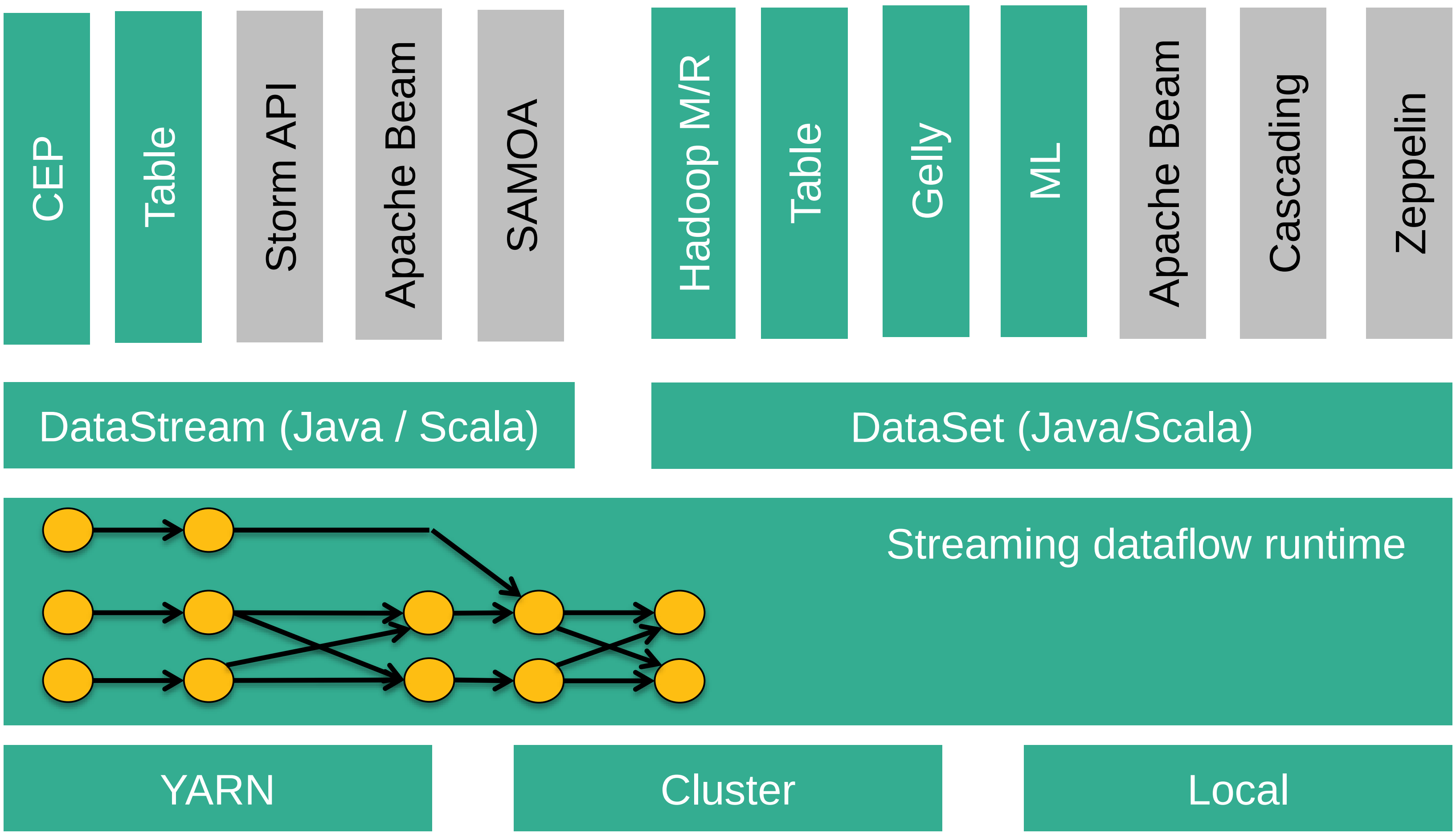
- Expressive **DataStream API** (similar to high level APIs known from the batch world)
- Flink is a full-fledged **batch processor** with an optimizer, managed memory, memory-aware algorithms, build-in iterations
- Many **libraries**: Complex Event Processing (CEP), Graph Processing, Machine Learning
- **Integration** with YARN, HBase, ElasticSearch, Kafka, MapReduce, …

Questions?

- Ask now!
- eMail: rmetzger@apache.org
- Twitter: @rmetzger_

- Follow: @ApacheFlink
- Read: flink.apache.org/blog, data-artisans.com/blog/
- Mailinglists: (news | user | dev)@flink.apache.org

Apache Flink stack



Appendix

Roadmap 2016

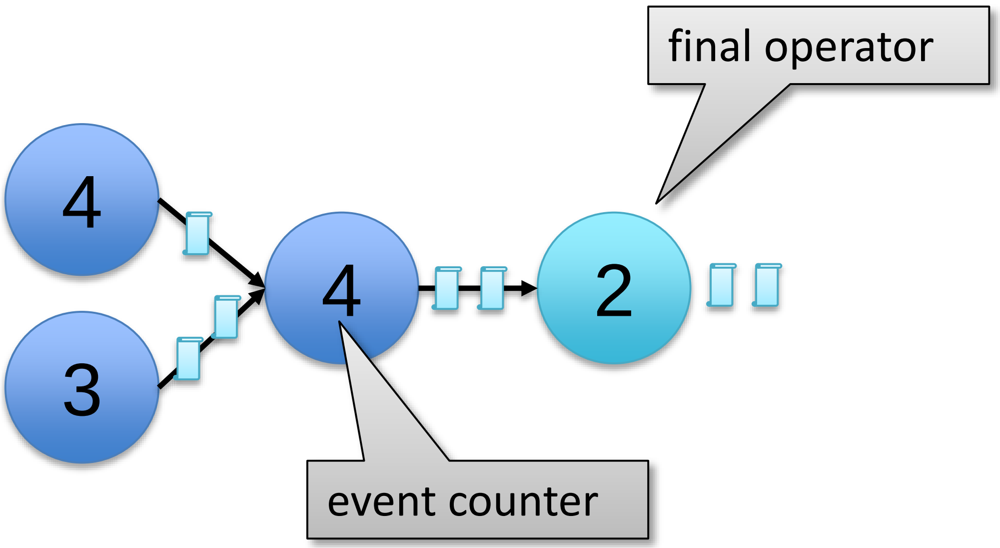
- SQL / StreamSQL
- CEP Library
- Managed Operator State
- Dynamic Scaling
- Miscellaneous

Miscellaneous

- Support for Apache Mesos
- Security
  - Over-the-wire encryption of RPC (akka) and data transfers (netty)
- More connectors
  - Apache Cassandra
  - Amazon Kinesis
- Enhance metrics
  - Throughput / Latencies
  - Backpressure monitoring
  - Spilling / Out of Core

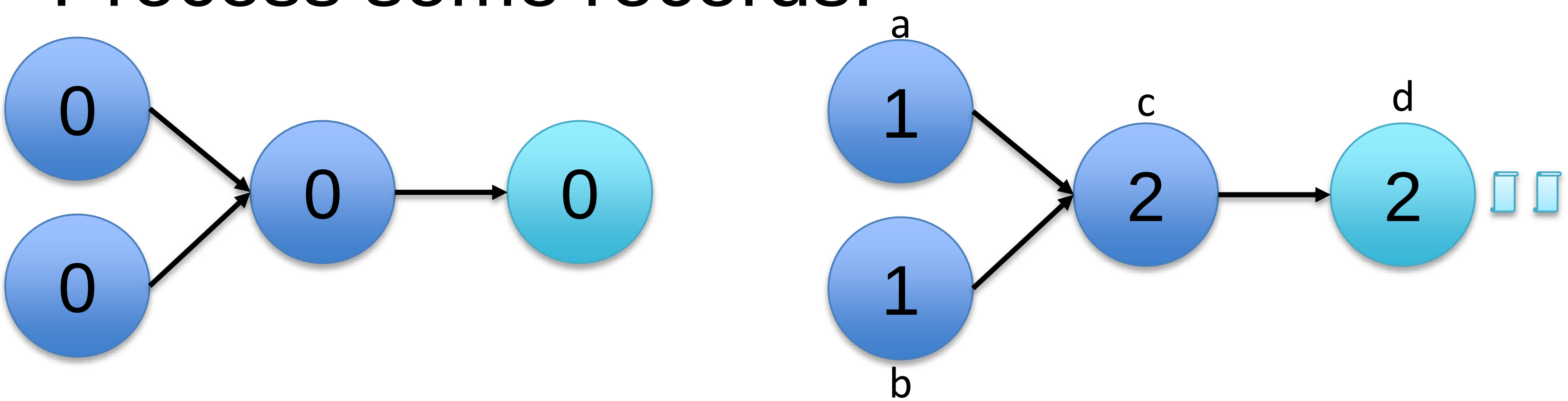
Fault Tolerance and correctness

- How can we ensure the state is always in sync with the events?



Naïve state checkpointing approach

- Process some records:

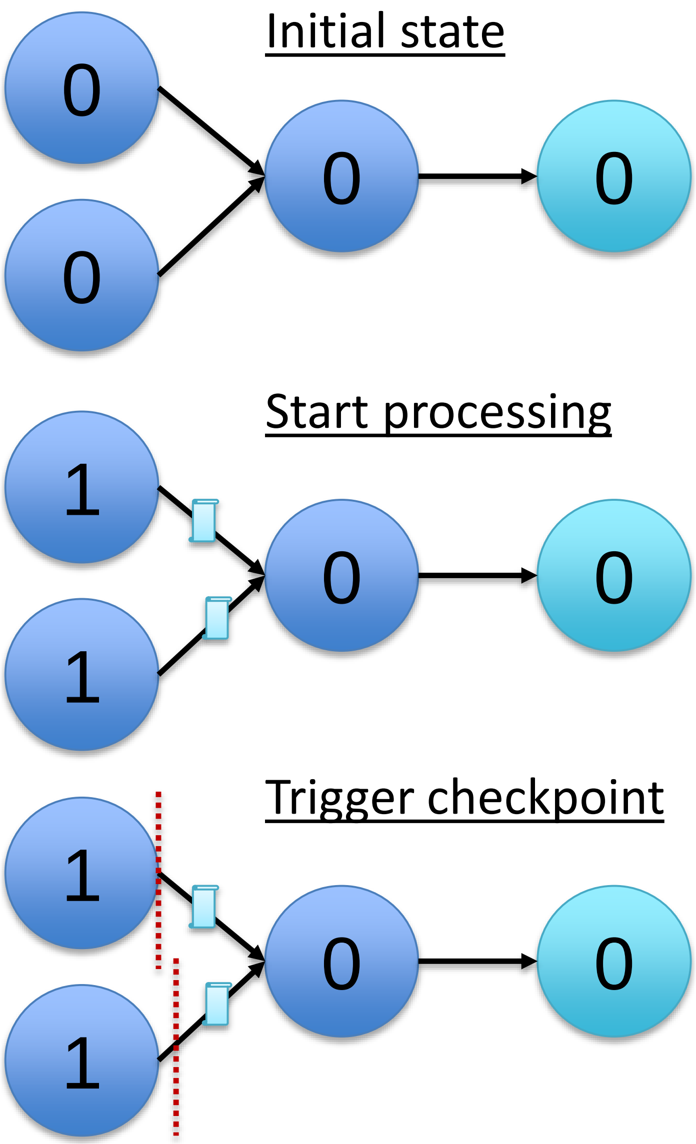


- Stop everything, store state:

Operator	State
a	1
b	1
c	2
d	2

- Continue processing …

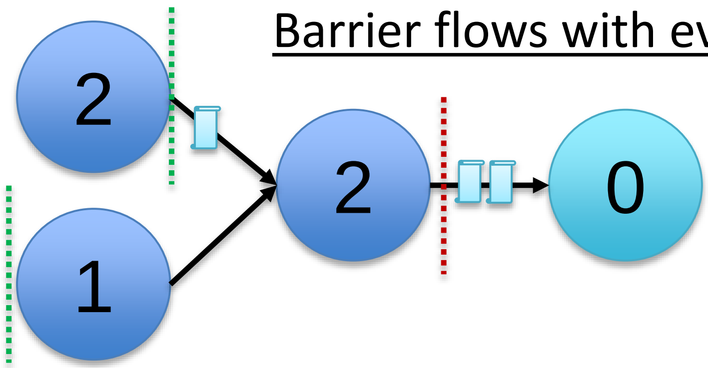
Distributed Snapshots



Operator	State
a	1
b	1

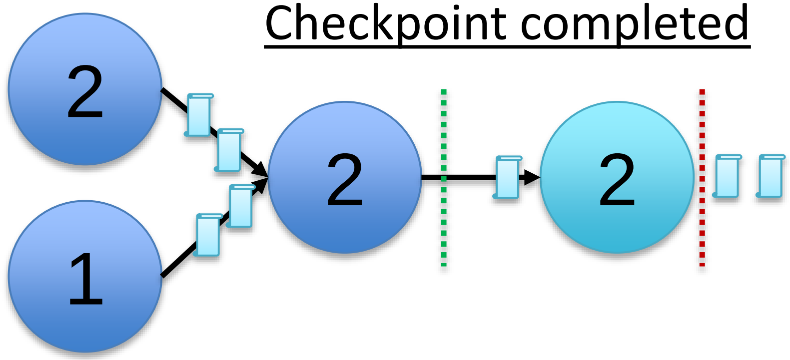
Distributed Snapshots

Barrier flows with events



Operator	State
a	1
b	1
c	2

Checkpoint completed



Operator	State
a	1
b	1
c	2
d	2

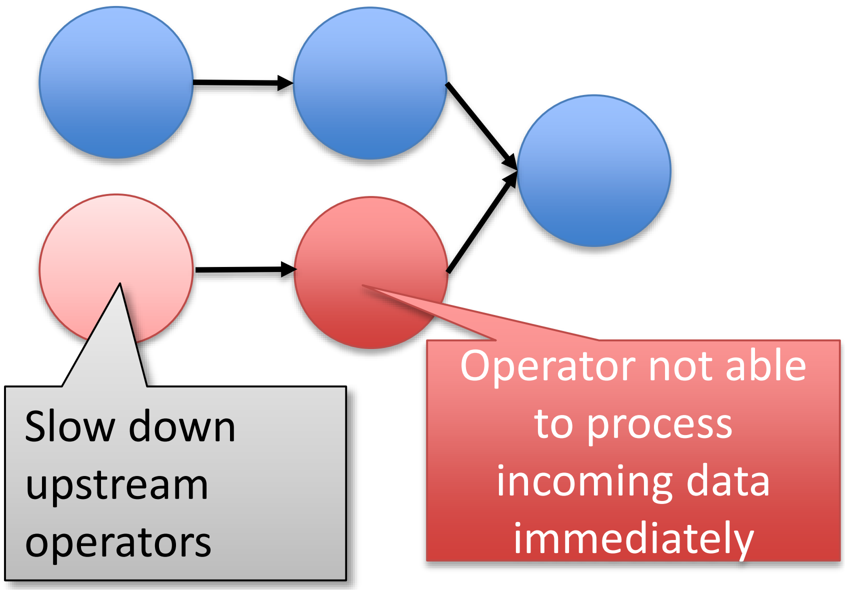
Complete, consistent state snapshot

- Valid snapshot without stopping the topology
- Multiple checkpoints can be in-flight

Analysis of naïve approach

- – Introduces latency
- – Reduces throughput
- Can we create a **correct snapshot while keeping the job running**?
- Yes! By creating a distributed snapshot

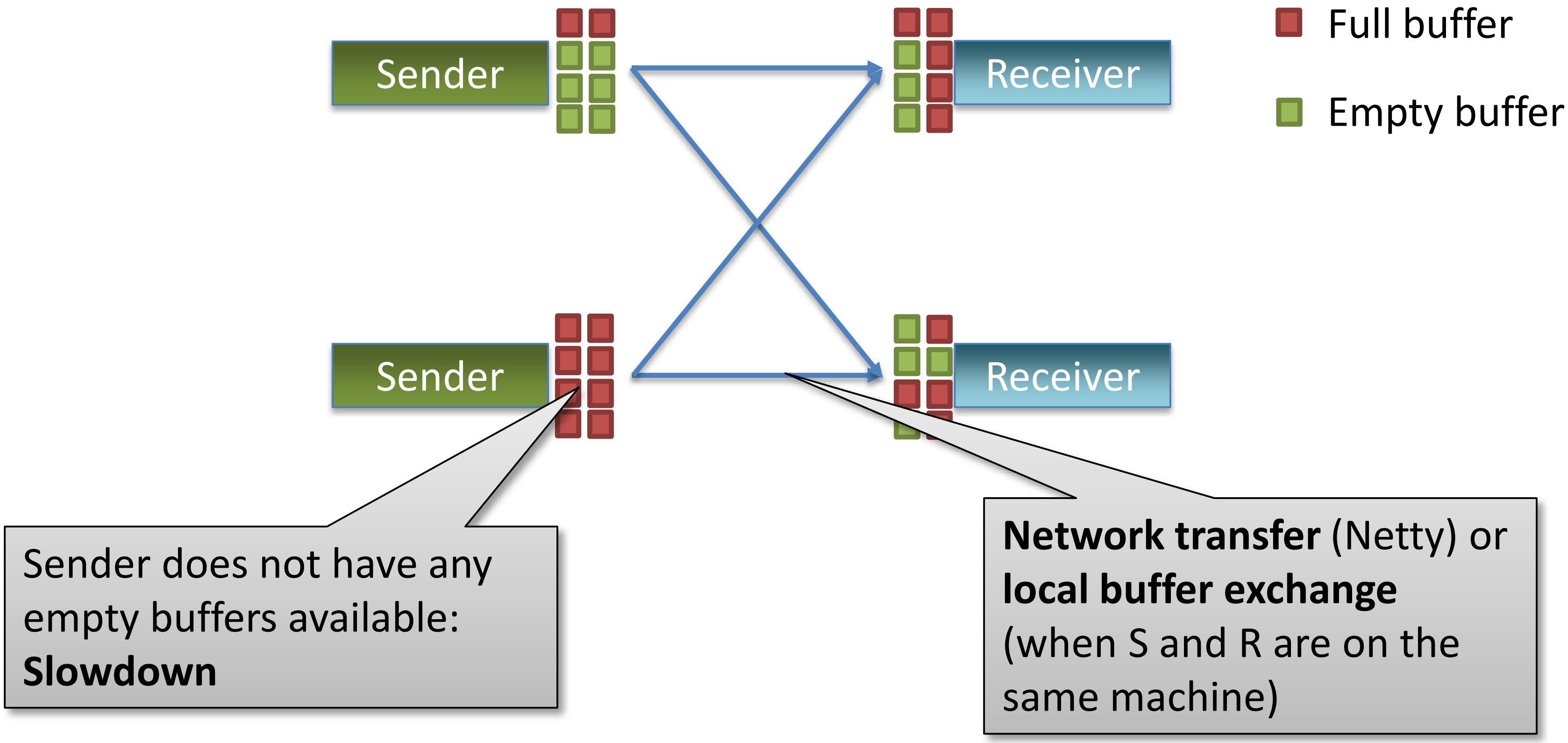
Handling Backpressure



Backpressure might occur when:

- Operators create checkpoints
- Windows are evaluated
- Operators depend on external resources
- JVMs do Garbage Collection

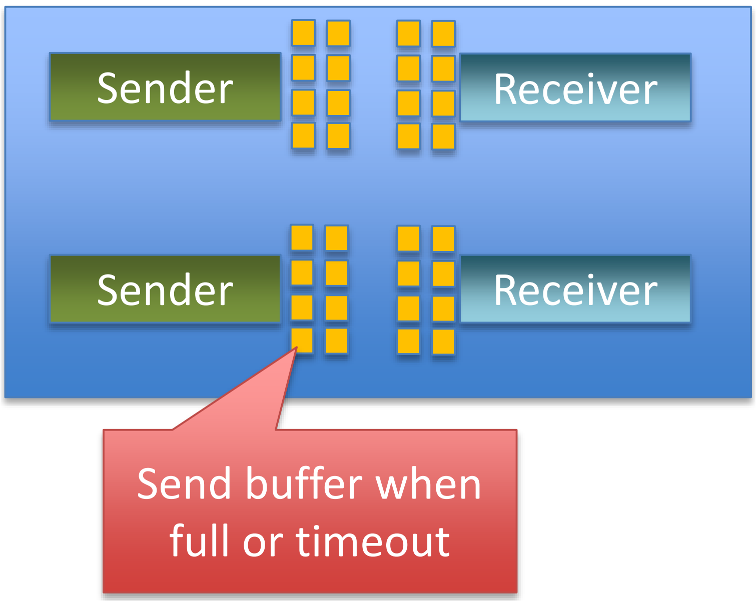
Handling Backpressure

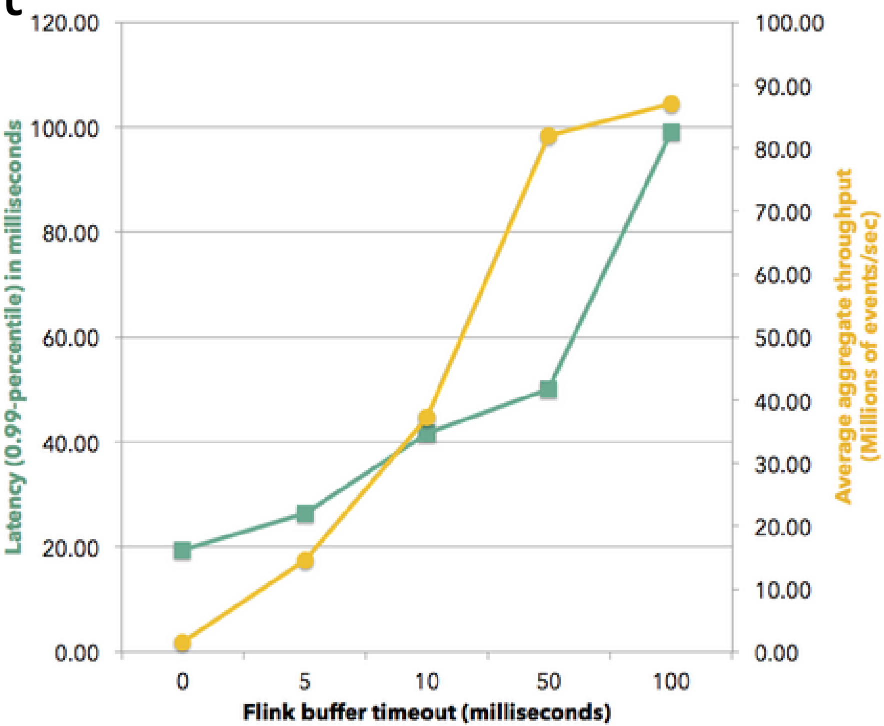


- Data sources slow down pulling data from their underlying system (Kafka or similar queues)

How do latency and throughput affect each other?

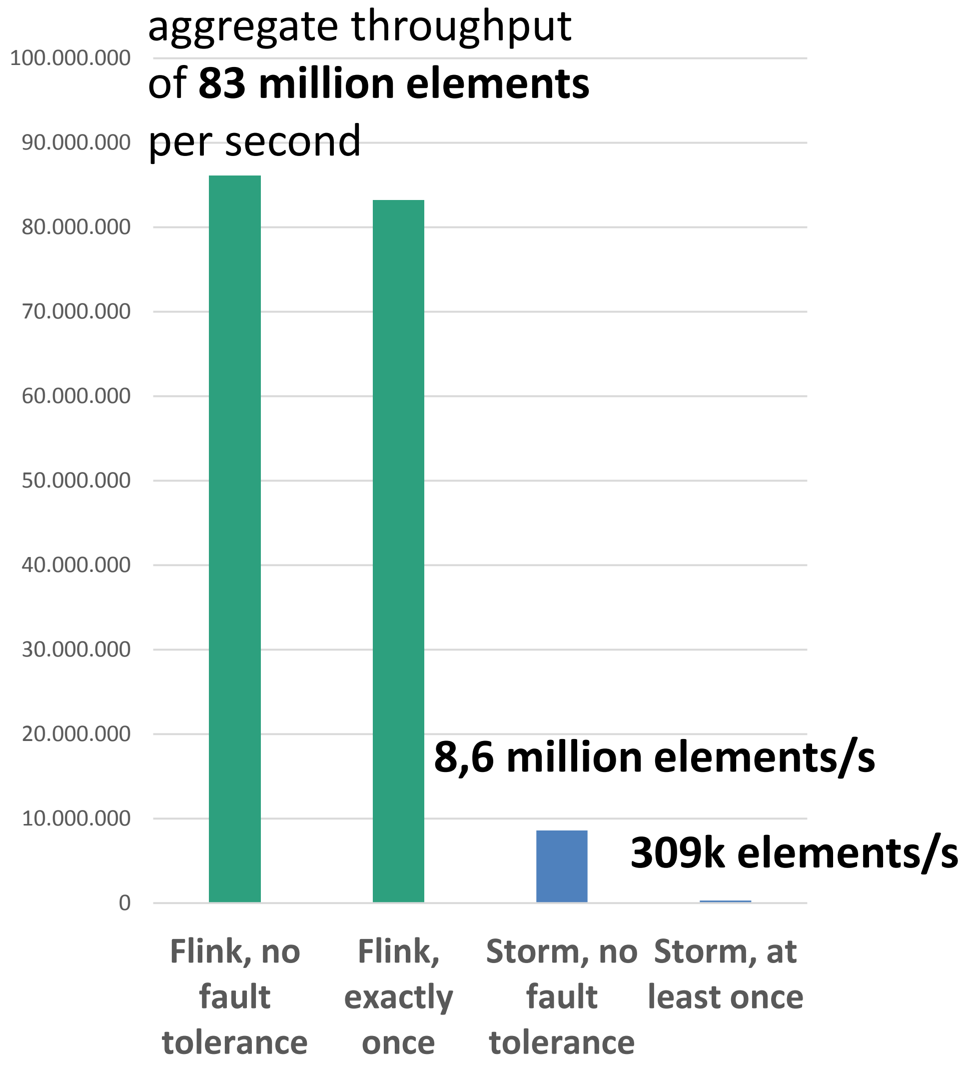
- High throughput by batching events in network buffers
- Filling the buffers introduces latency
- Configurable buffer timeout

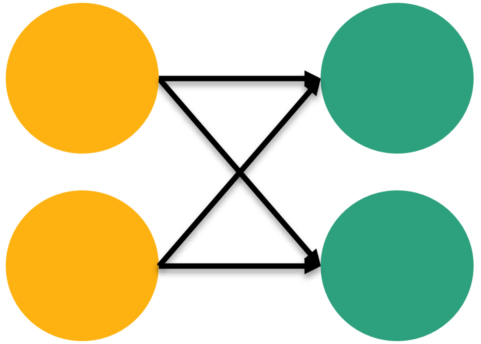




30 Machines, one repartition step

Aggregate throughput for stream record grouping





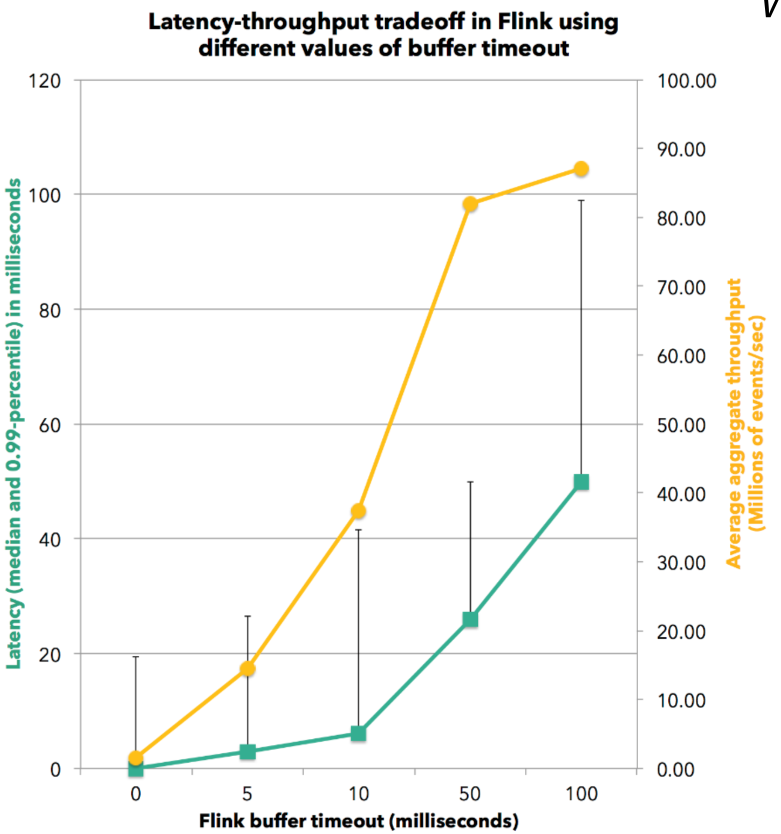
30 machines,
120 cores,
Google Compute

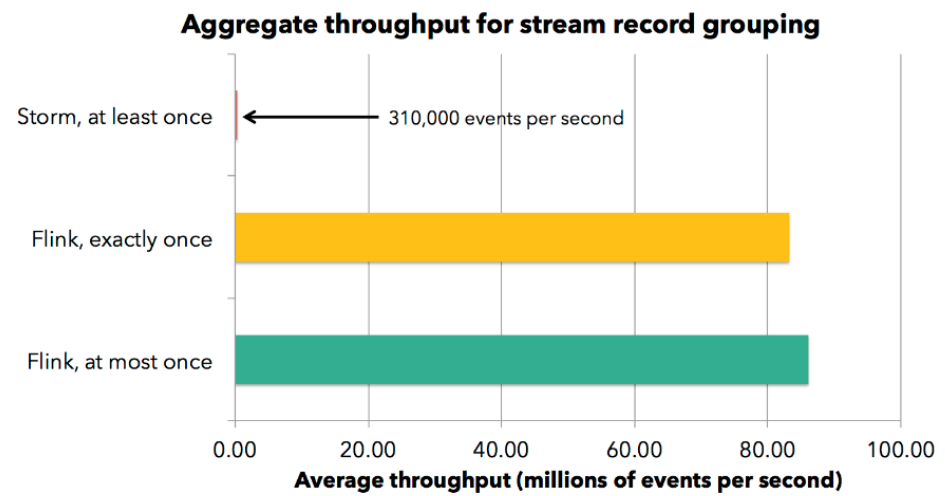
→ Flink achieves 260x higher throughput with fault tolerance

Performance: Summary

Continuous streaming + Latency-bound buffering + Distributed Snapshots = High Throughput & Low Latency

With configurable throughput/latency tradeoff





The building blocks: Summary

Windowing / Out of order events

- Tumbling / sliding windows
- Event time / processing time
- Low watermarks for out of order events

State handling

- Managed operator state for backup/recovery
- Large state with RocksDB
- Savepoints for operations

Fault tolerance and correctness

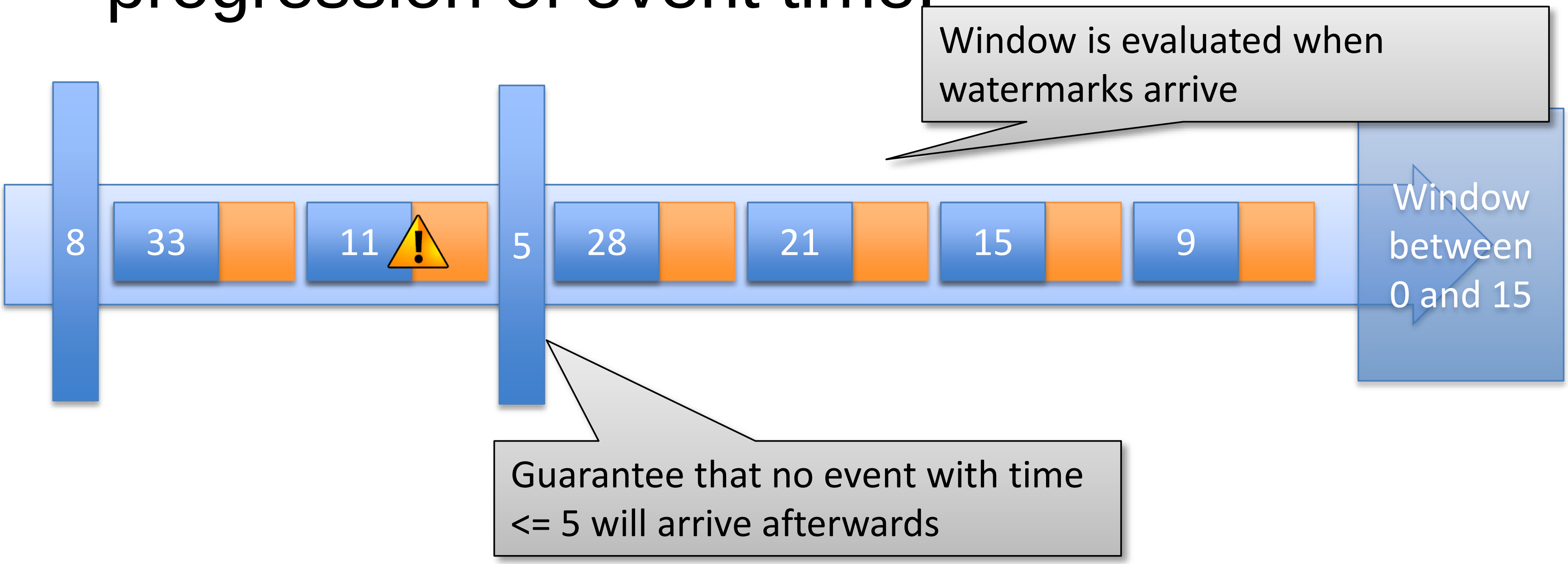
- Exactly-once semantics for managed operator state
- Lightweight, asynchronous distributed snapshotting algorithm

Low latency
High throughput

- Efficient, pipelined runtime
- no per-record operations
- tunable latency / throughput tradeoff
- Async checkpoints

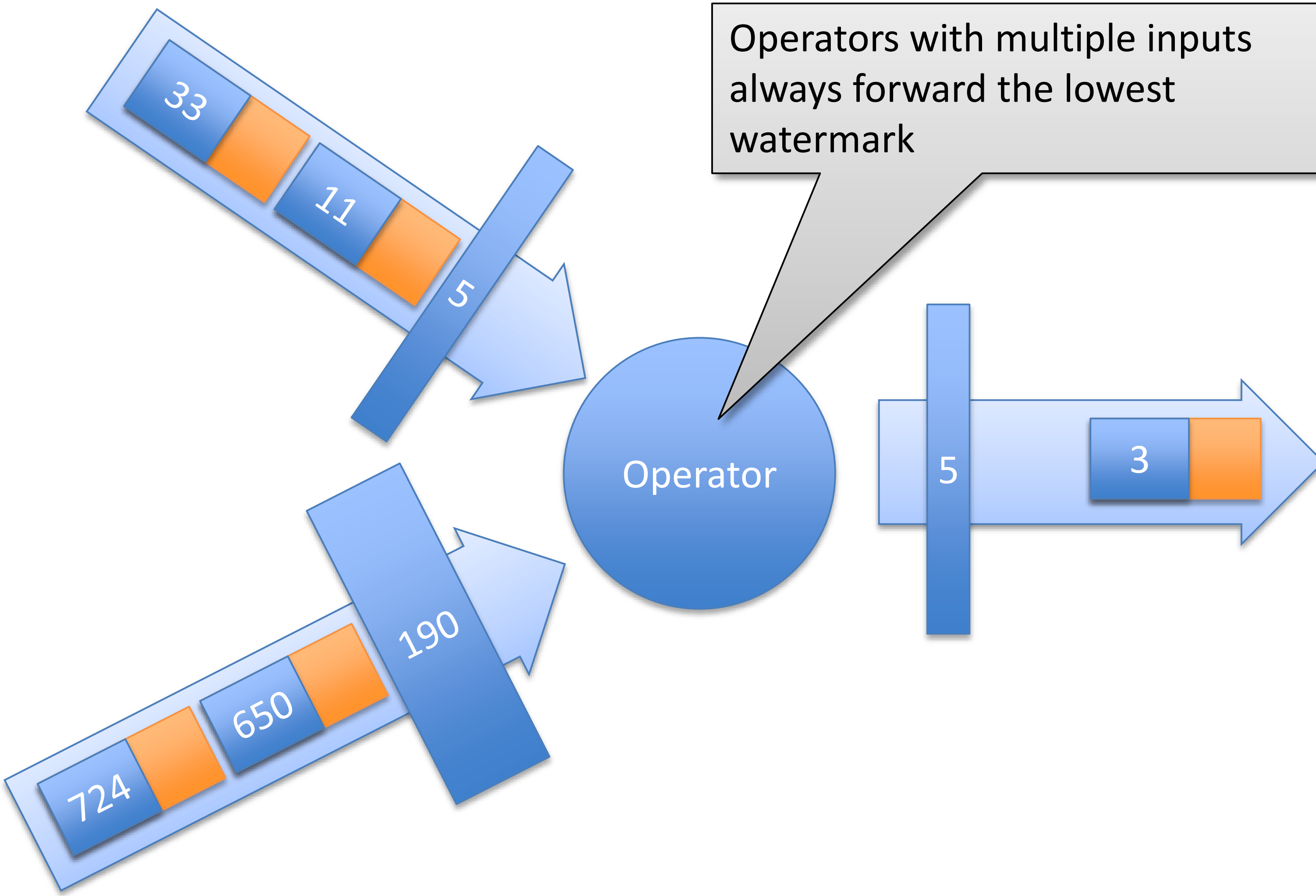
Low Watermarks

- We periodically send low-watermarks through the system to indicate the progression of event time.



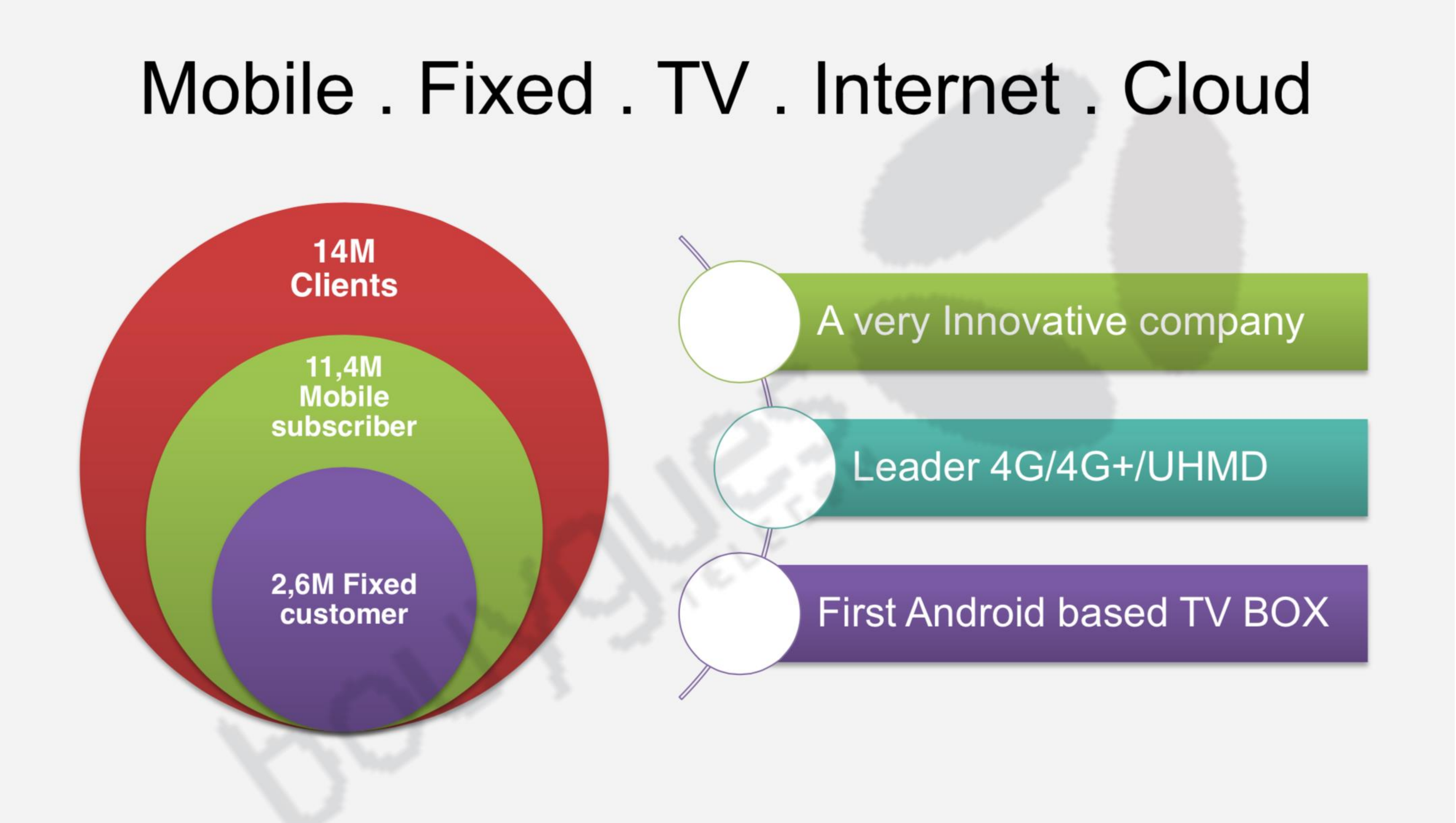
For more details: "MillWheel: Fault-Tolerant Stream Processing at Internet Scale" by T. Akidau et. al.

Low Watermarks



For more details: “MillWheel: Fault-Tolerant Stream Processing at Internet Scale” by T. Akidau et. al.

Bouygues Telecom



Bouygues Telecom

- Produce Mobile **QoE** indicators from massive network equipment's event logs (**4 Billions/day**).

- Goals:
  - QoE (User) instead of QoS (Machine).
  - Real-time Diagnostic (**<60sec.** end-to-end latency)
  - Business Intelligence
  - Real-time alarming
  - Reporting

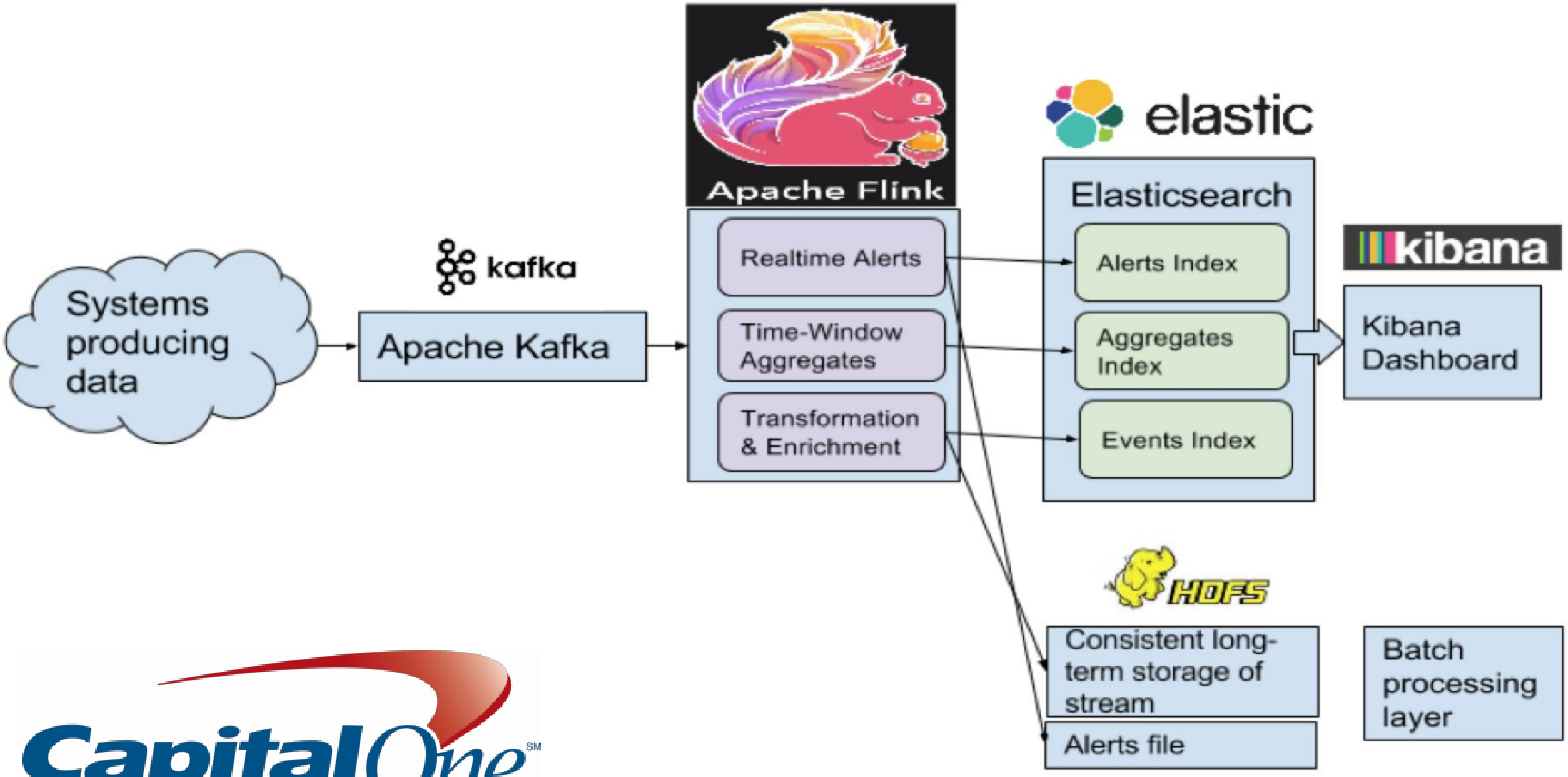
Bouygues Telecom

- We ran stress tests on our biggest raw Kafka topic:
  - A day of Data.
  - **2 Billions** events (480Gib compressed).
  - **10** Kafka partitions
  - **10** Flink TaskManagers (Only 1GB Memory each)

Enrich Rate (in Tickets/second)

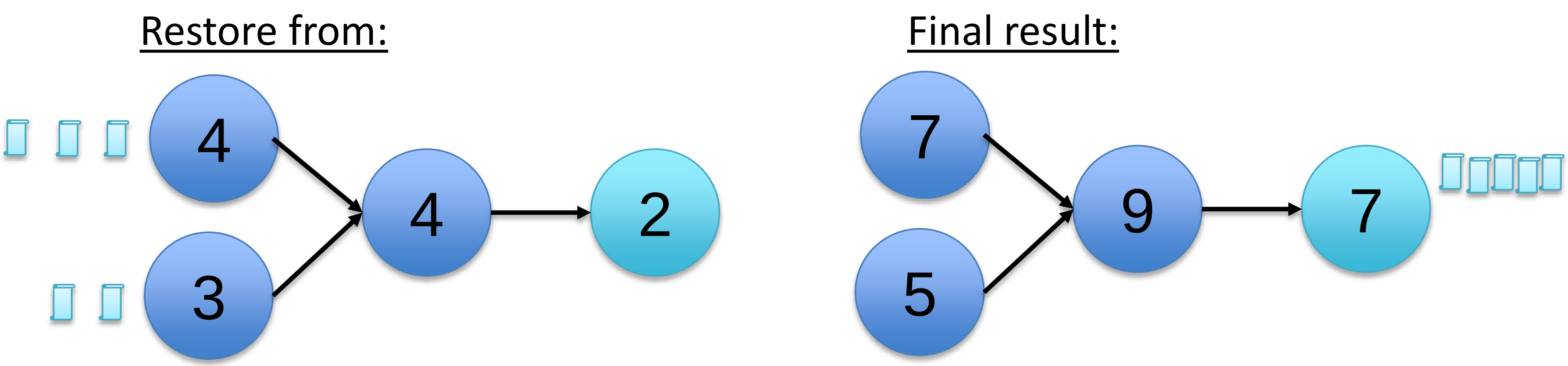
Total Processing time (ms)
Kafka I/O Duration (ms)

Capital One


Systems producing data
kafka
Apache Kafka
Apache Flink
Realtime Alerts
Time-Window Aggregates
Transformation & Enrichment
elastic
Elasticsearch
Alerts Index
Aggregates Index
Events Index
kibana
Kibana Dashboard
HDFS
Consistent long-term storage of stream
Alerts file
Batch processing layer

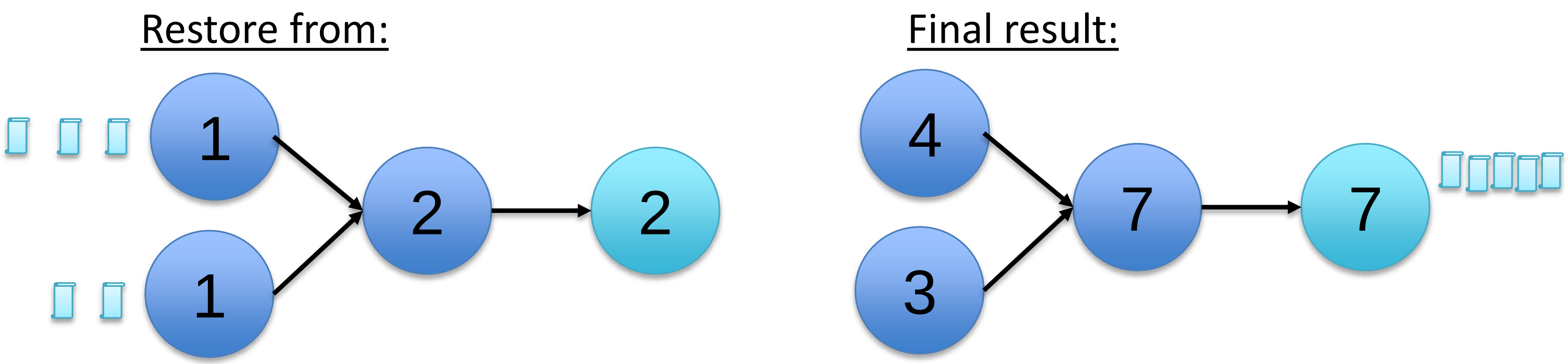
Fault Tolerance in streaming

- **Failure with “at least once”:** replay



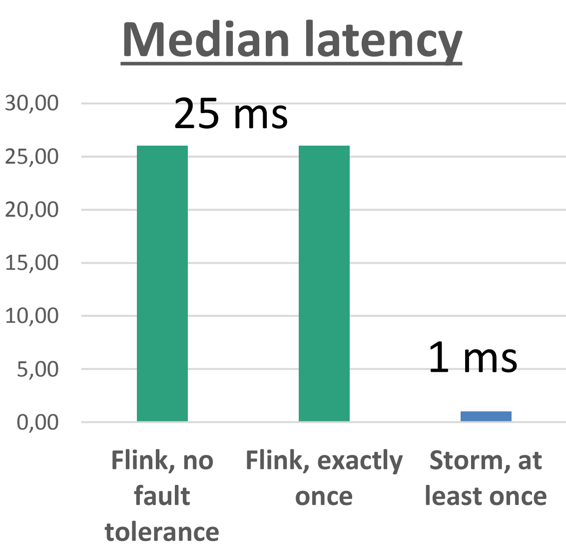
Fault Tolerance in streaming

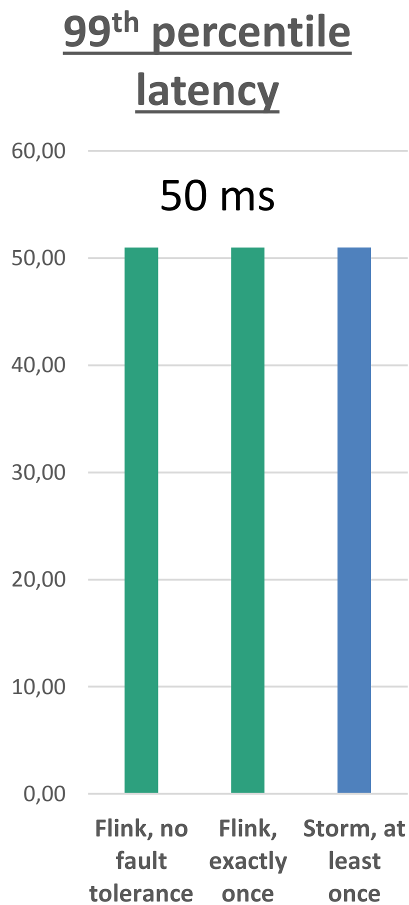
- **Failure with "exactly once":** state restore

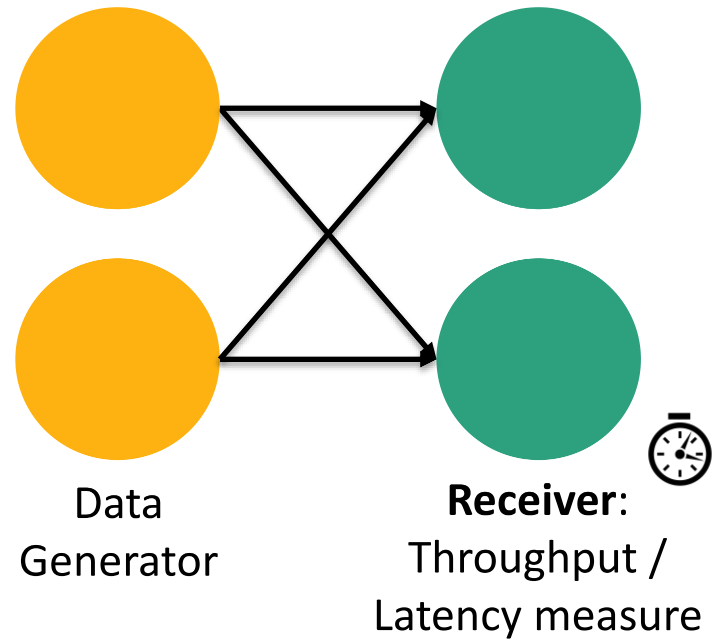


Latency in stream record grouping

- Measure time for a record to travel from source to sink







Savepoints: Simplifying Operations

- Streaming jobs usually run 24x7 (unlike batch).
- **Application bug fixes**: Replay your job from a certain point in time (savepoint)
- **Flink bug fixes**
- **Maintenance** and system migration
- **What-If simulations**: Run different implementations of your code against a savepoint

Pipelining

Basic building block to "keep the data moving"

- Low latency
- Operators push data forward
- Data shipping as buffers, not tuple-wise
- Natural handling of back-pressure