Apache Flink

Stream Processing Framework

Introduction

Apache Flink

Apache Flink is an open source stream processing framework

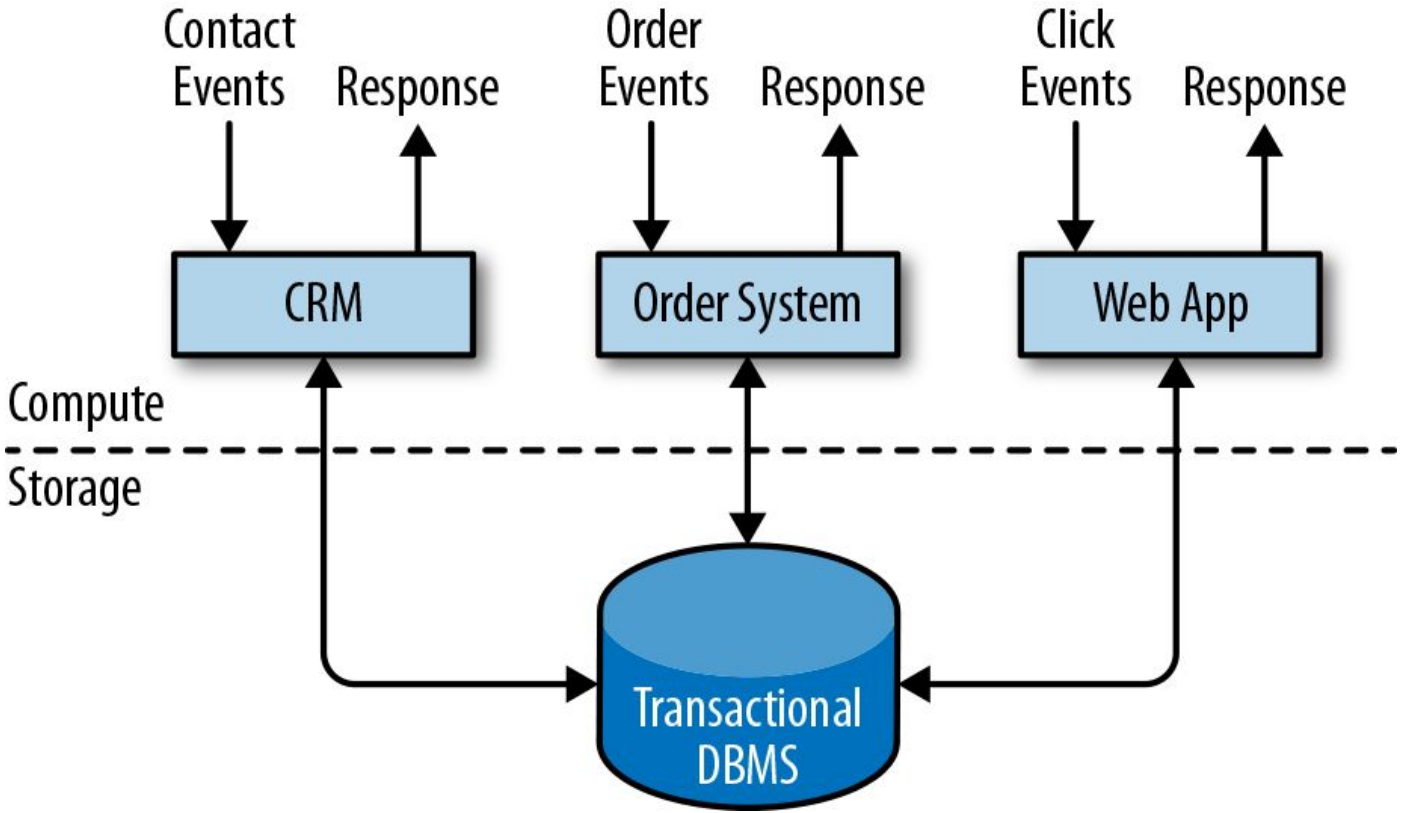
- Low latency
- High throughput
- Stateful
- Distributed

Developed at the Apache Software Foundation, 1.13.0 latest release

Streaming is the **biggest change** in data infrastructure since Hadoop

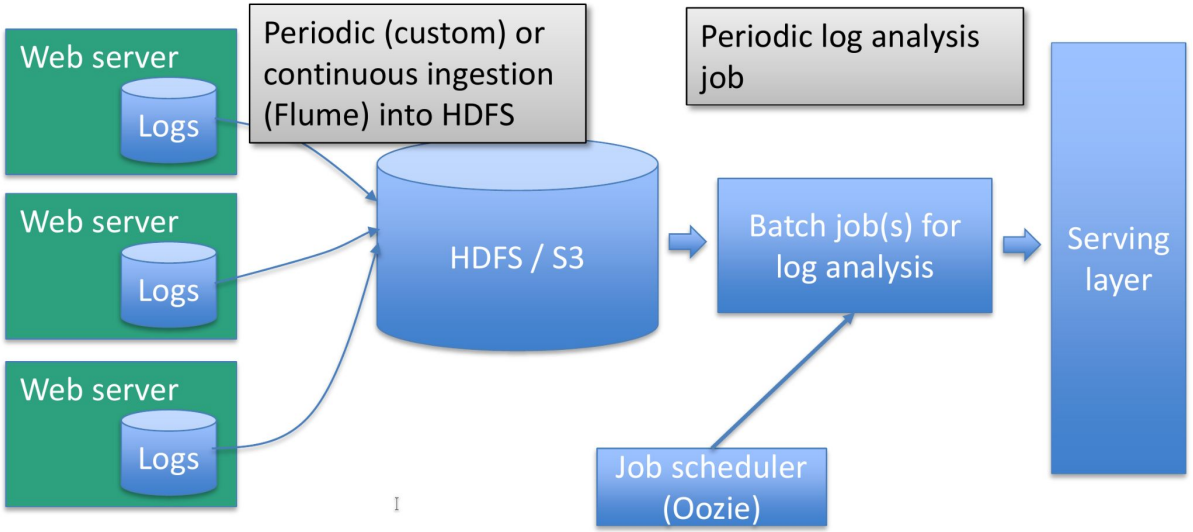
1. Radically simplified infrastructure
2. Do more with your data, faster
3. Can completely subsume batch

Traditional Data Infrastructures - Transactional



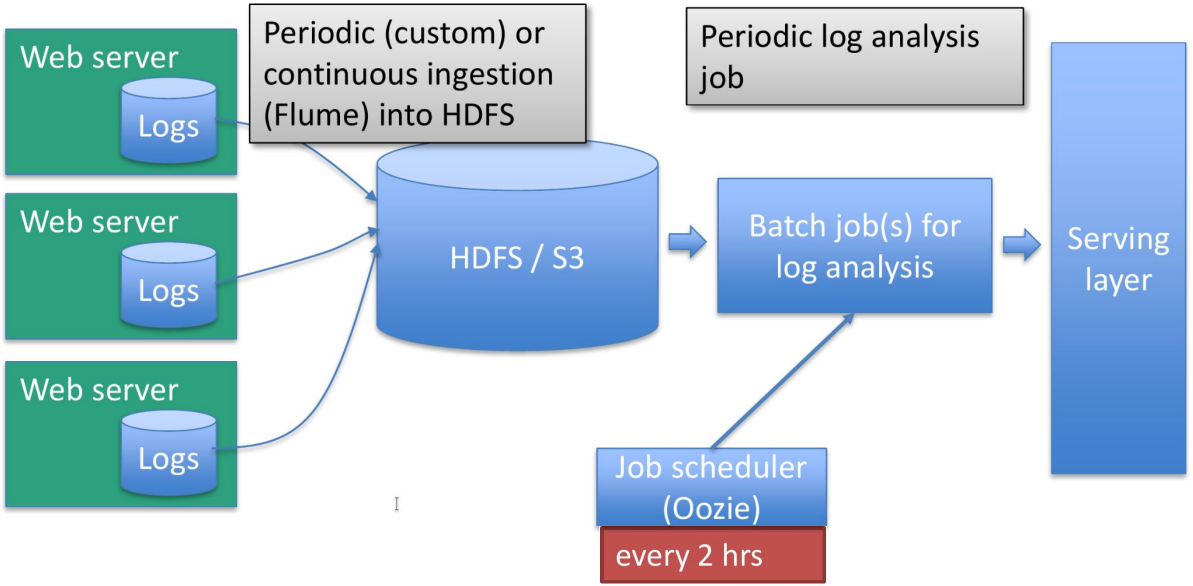
Traditional data processing

Log analysis example using a batch processor



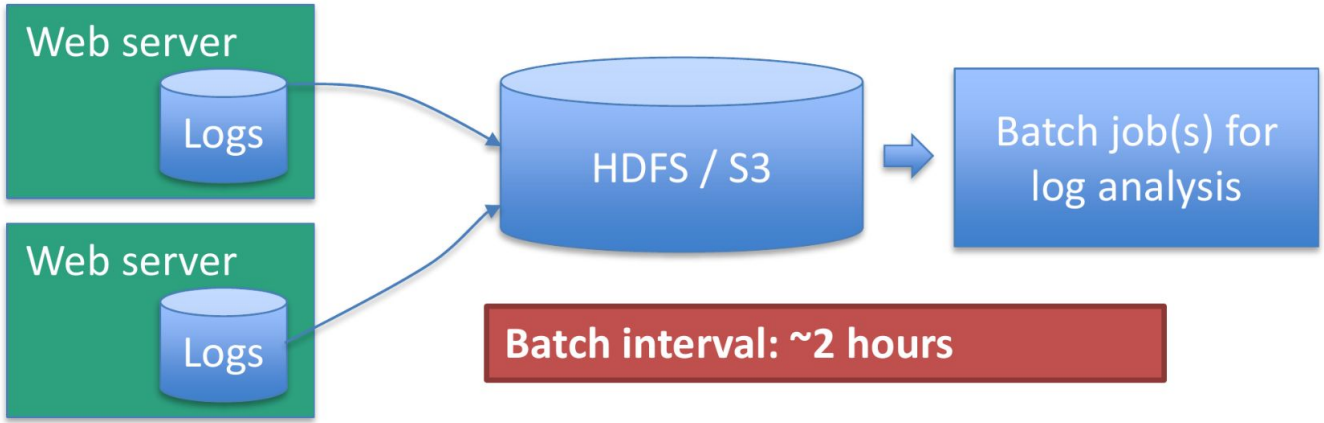
Traditional data processing

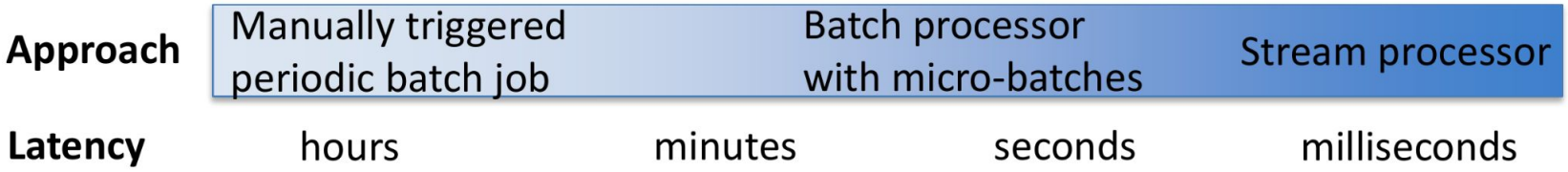
Log analysis example using a batch processor



Data processing without stream processor

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

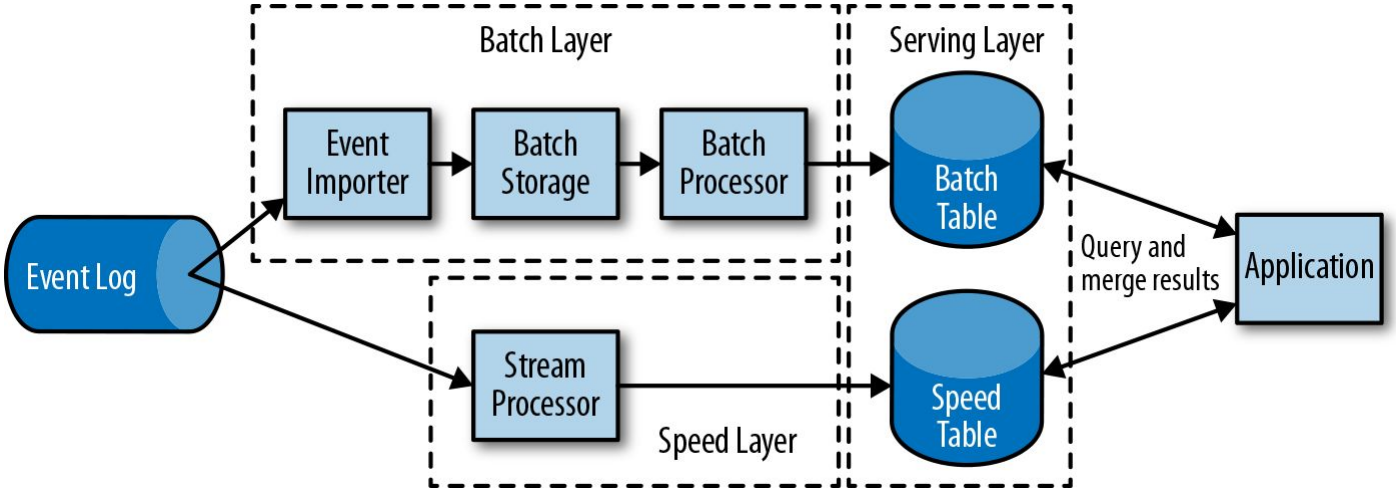




Downsides of stream processing with a batch engine

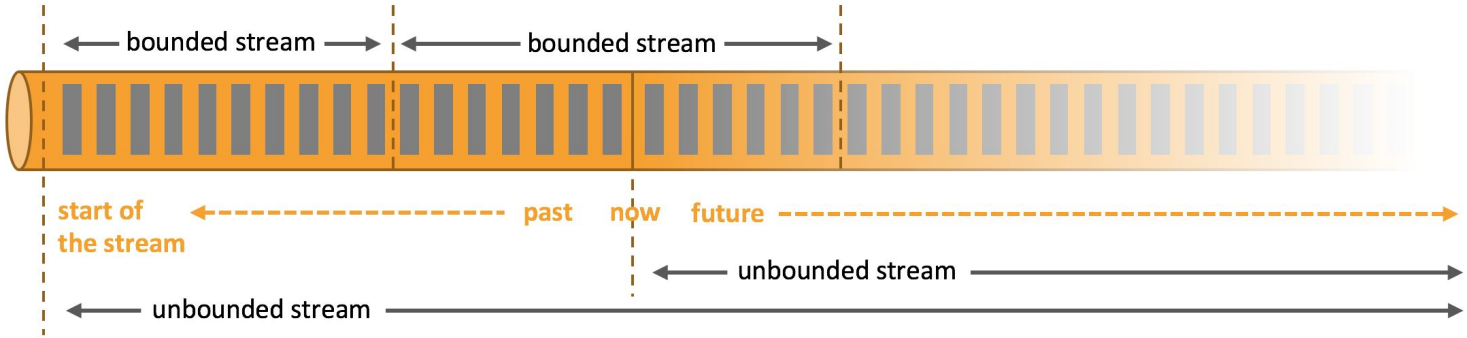
- Very high latency (hours)
- Complex architecture required:
  - Periodic job scheduler (e.g. Oozie, Airflow)
  - Data loading into HDFS (e.g. Flume)
  - Batch processor
  - (When using the “lambda architecture”: a stream processor)
- All these components need to be implemented and maintained
- Backpressure: How does the pipeline handle load spikes?

A bit of history



The lambda architecture augments the traditional periodic batch processing architecture with a speed layer that is powered by a low-latency stream processor. Data arriving at the lambda architecture is ingested by the stream processor and also written to batch storage.

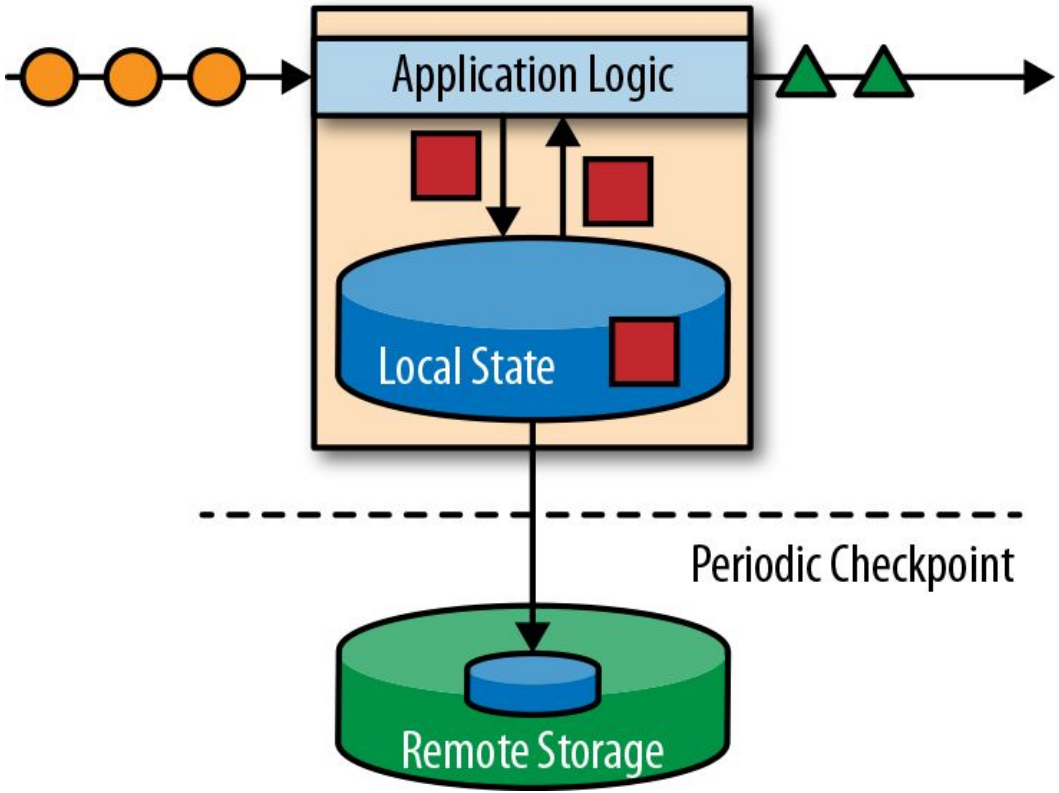
Stream Processing



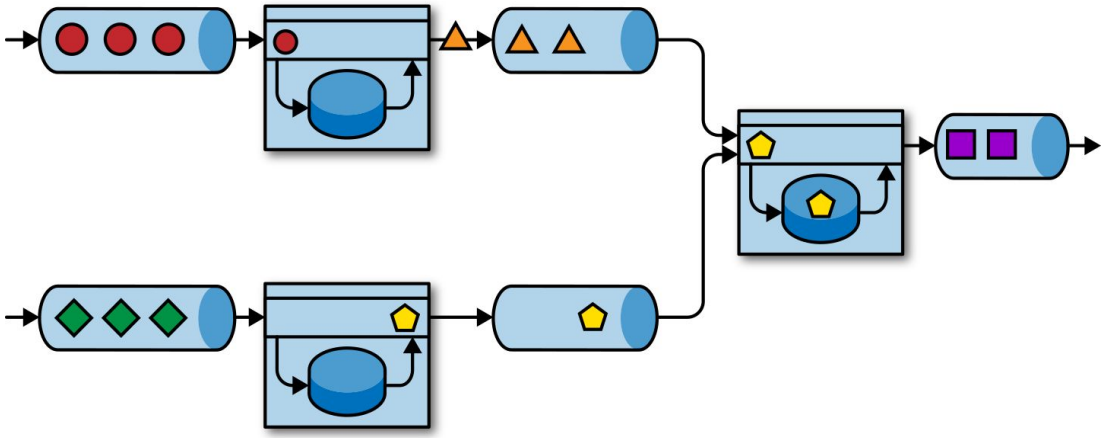
Batch processing is the paradigm at work when you process a bounded data stream. In this mode of operation you can choose to ingest the entire dataset before producing any results, which means that it is possible, for example, to sort the data, compute global statistics, or produce a final report that summarizes all of the input.

Stream processing, on the other hand, involves unbounded data streams. Conceptually, at least, the input may never end, and so you are forced to continuously process the data as it arrives.

Stateful Stream Processing

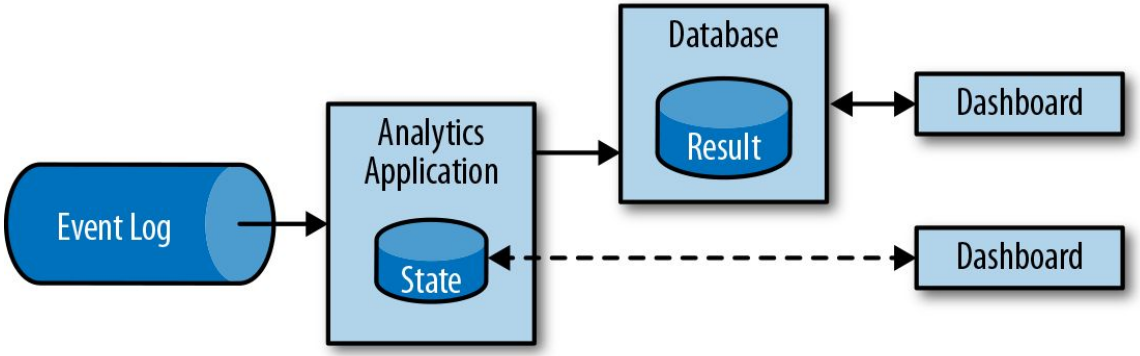


Event-Driven Applications



- Real-time recommendations (e.g., for recommending products while customers browse a retailer's website)
- Pattern detection or complex event processing (e.g., for fraud detection in credit card transactions)
- Anomaly detection (e.g., to detect attempts to intrude a computer network)

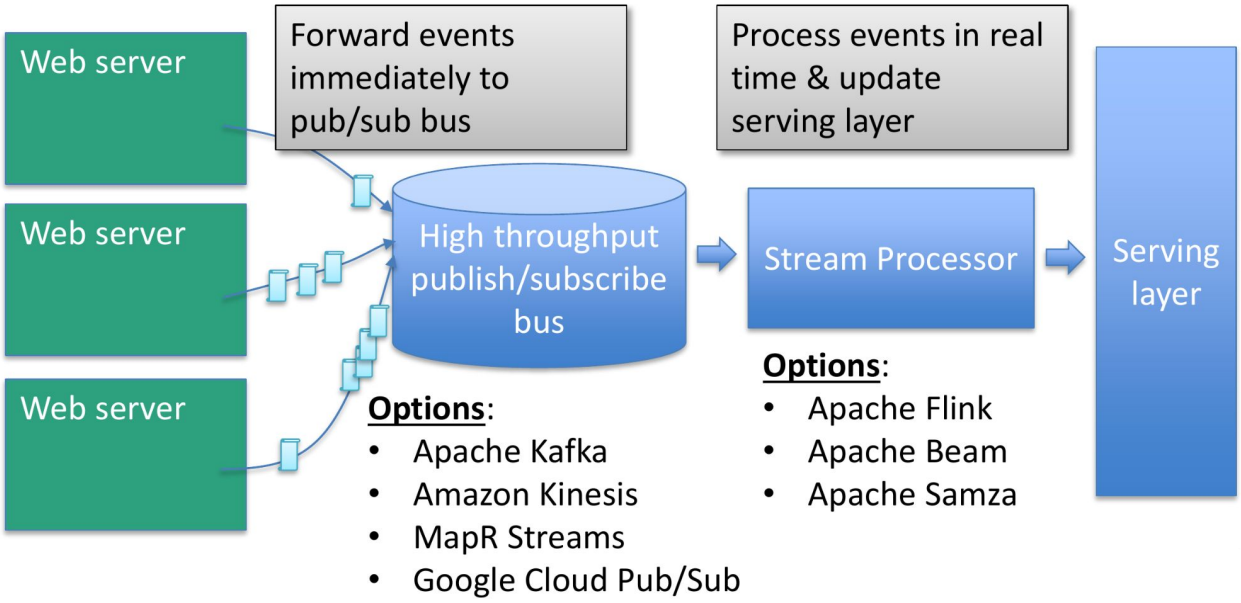
Streaming Analytics



- Monitoring the quality of cellphone networks
- Analyzing user behavior in mobile applications
- Ad-hoc analysis of live data in consumer technology

Log event analysis using a stream processor

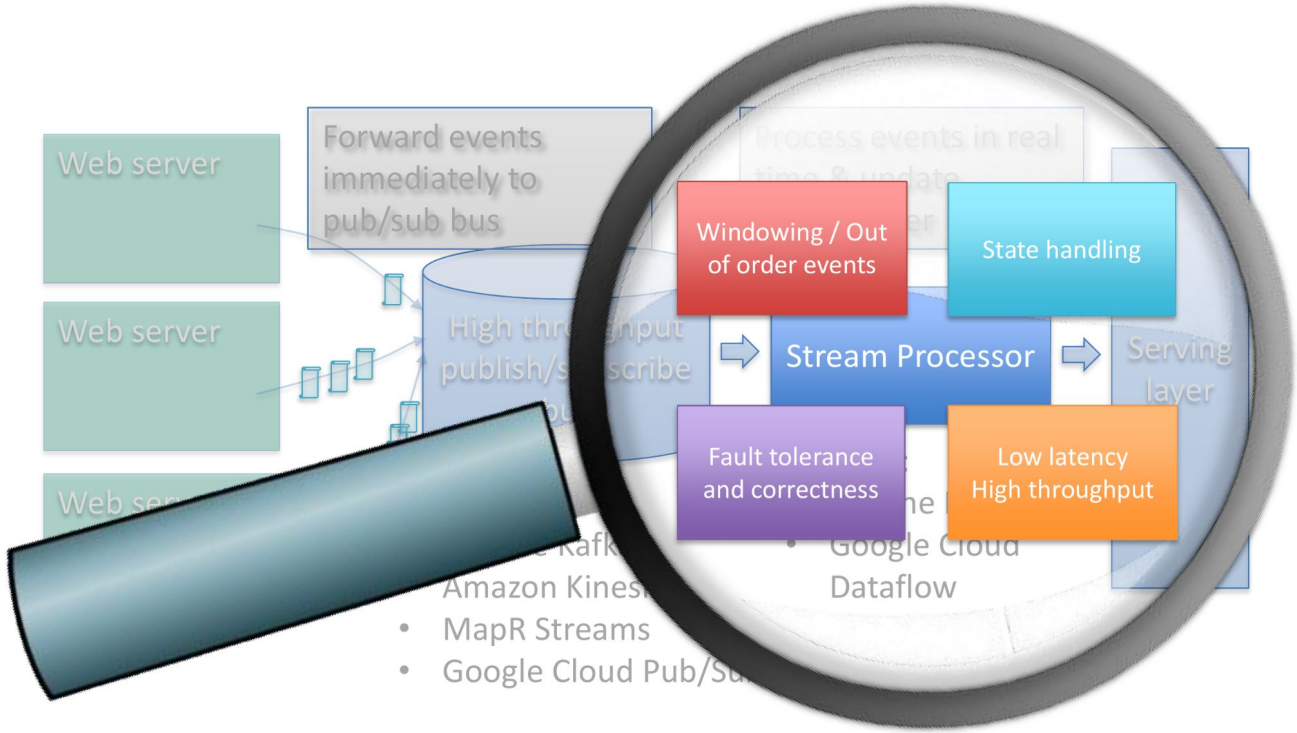
Stream processors allow to analyze events with sub-second latency.



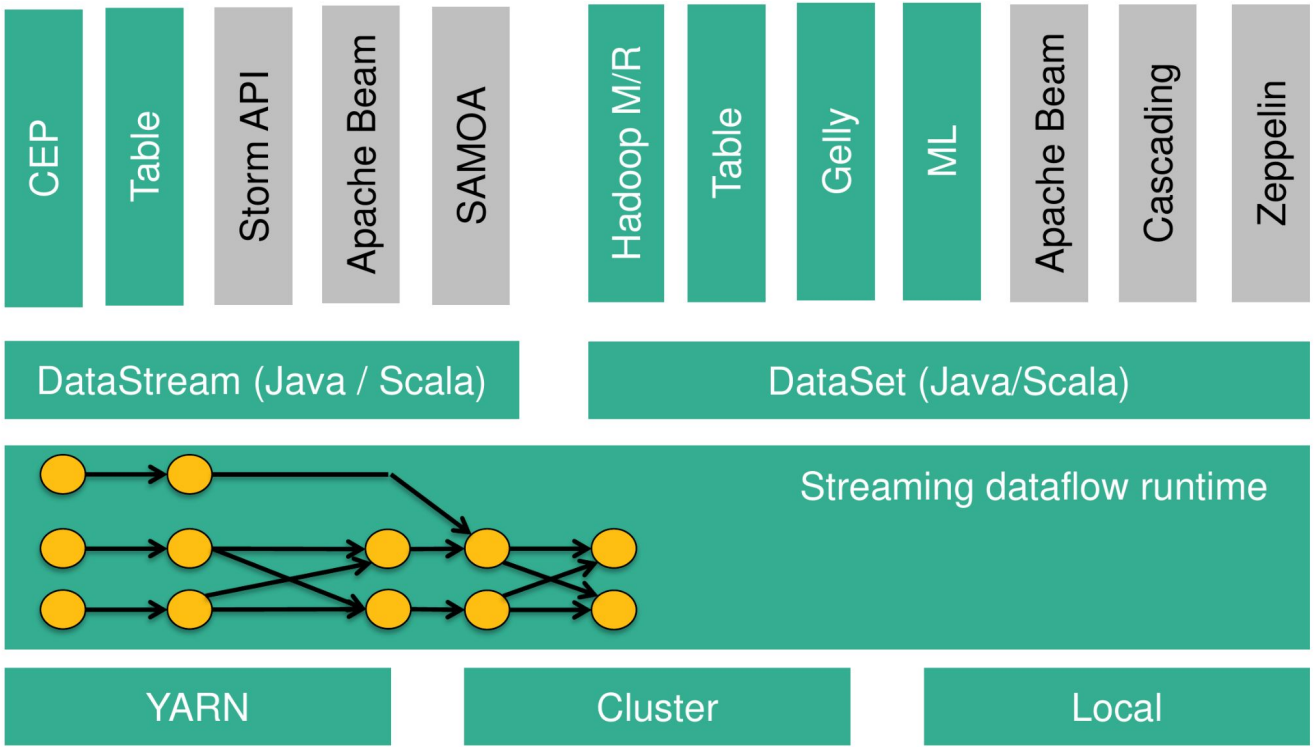
Real-world data is produced in a continuous fashion.

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

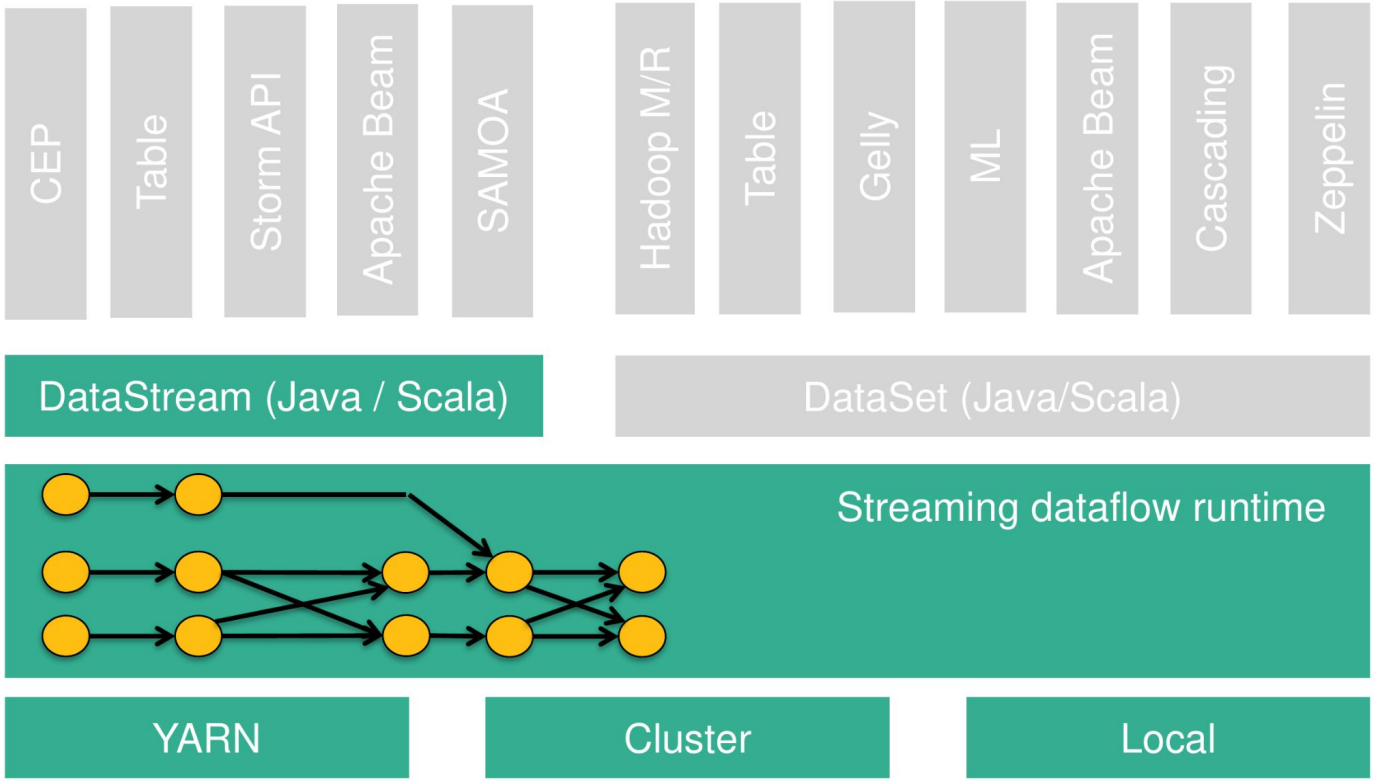
What do we need for replacing the “batch stack”?



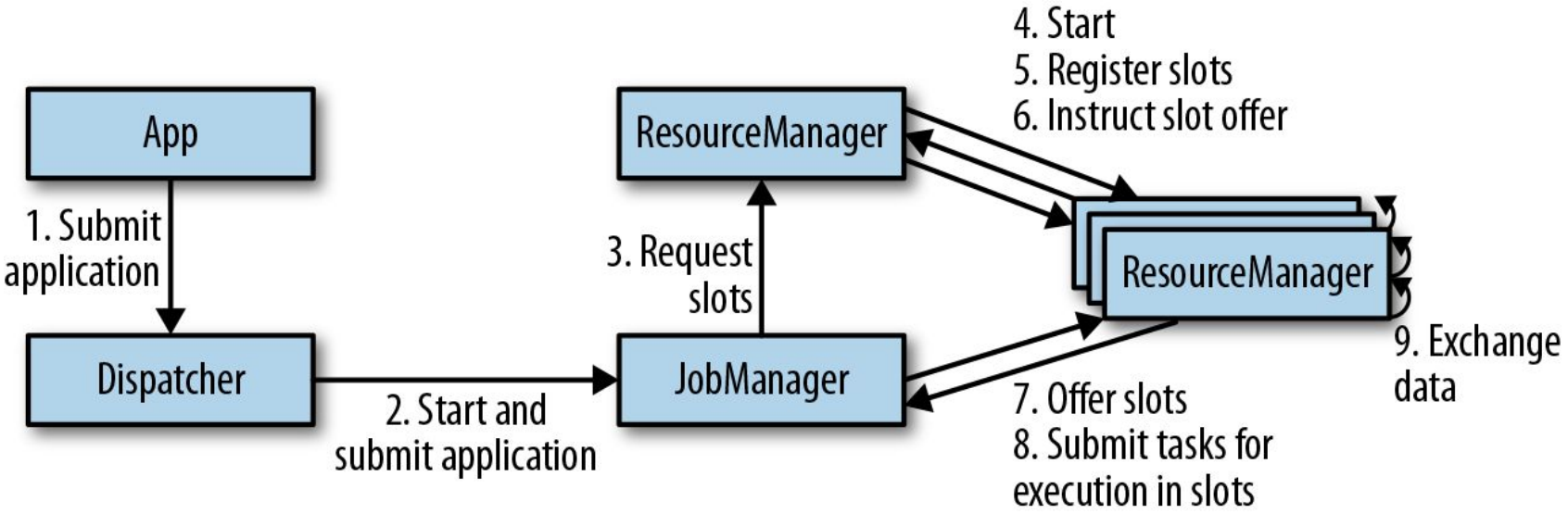
Apache Flink Stack



Use Case


CEP
Table
Storm API
Apache Beam
SAMOA
Hadoop M/R
Table
Gelly
ML
Apache Beam
Cascading
Zeppelin
DataStream (Java / Scala)
DataSet (Java/Scala)
Streaming dataflow runtime
YARN
Cluster
Local

Flink Component Interaction



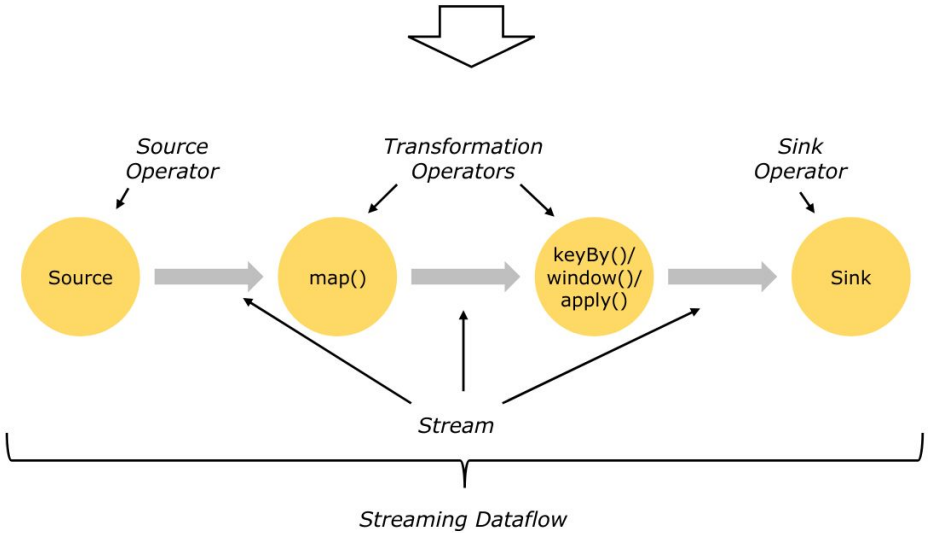
```
DataStream<String> lines = env.addSource(
                                new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy(event -> event.id)
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new MySink(...));
```

Source

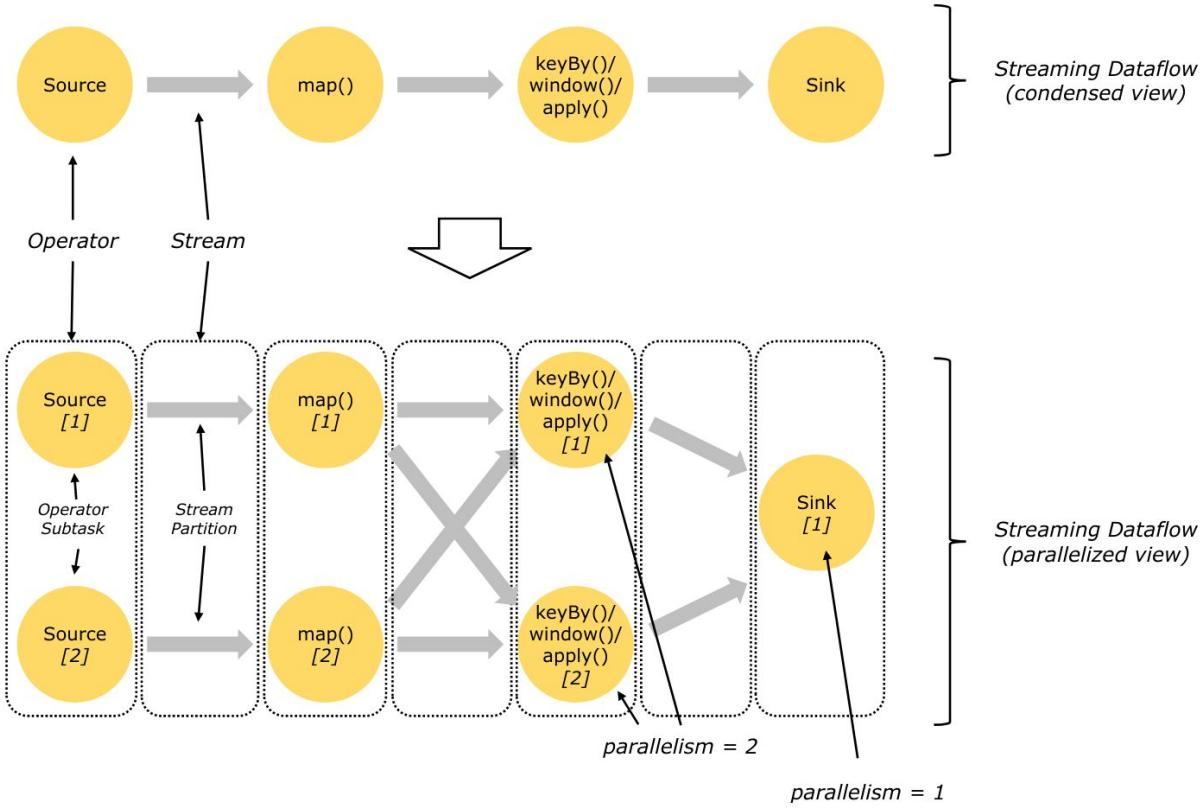
Transformation

Transformation

Sink



Parallel Dataflow



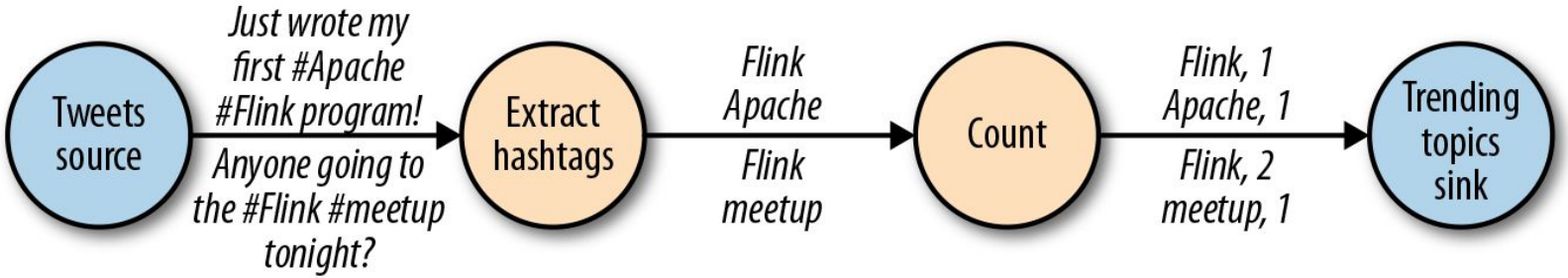
Start your first flink job

```
io.github.streamingwithflink.chapter1.AverageSensorReading
```

```
tail ./log/flink-nurcahyopujo-taskexecutor-0-flink-training.out
```

Stream Processing Fundamental

Dataflow Graphs



Dataflow program describes how data flows between operations.

Dataflow programs are commonly represented as directed graphs

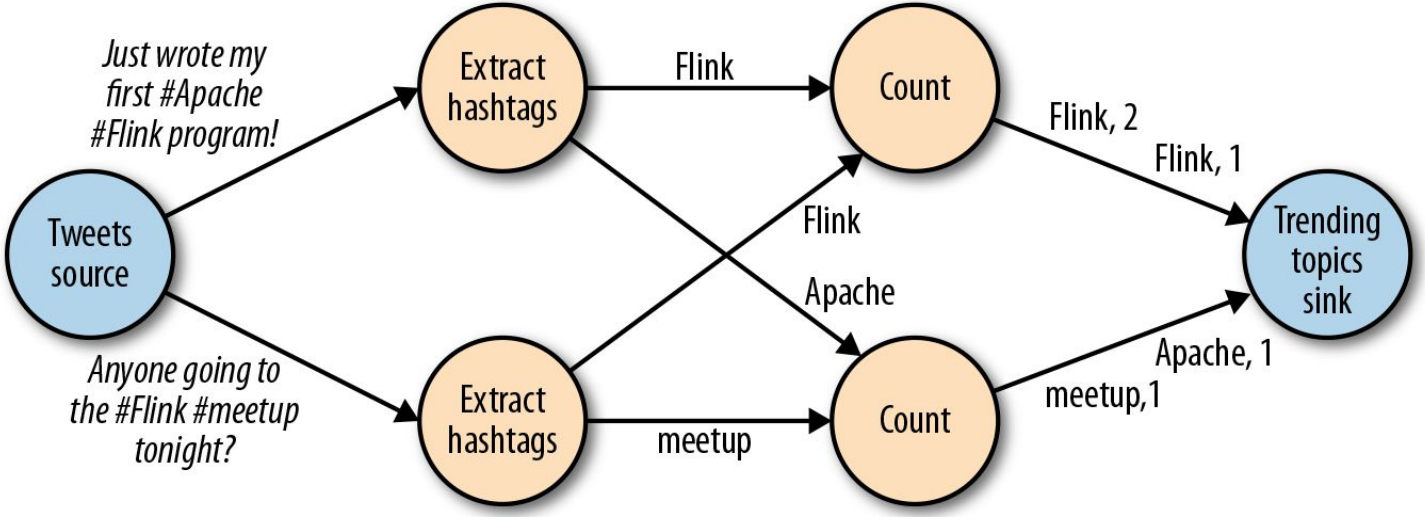
nodes are called operators and represent computations and edges represent data dependencies.

Operators are the basic functional units of a dataflow application.

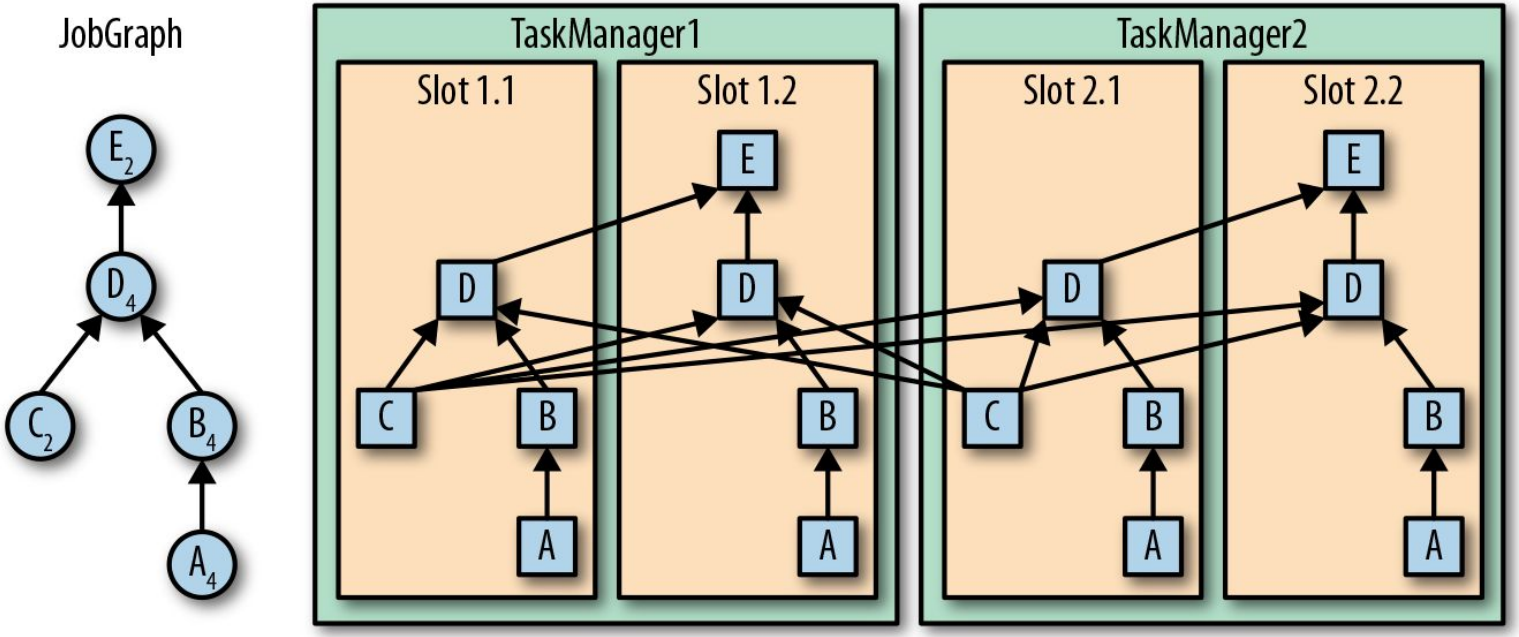
Operators without input ports are called data sources and operators without output ports are called data sinks.

A dataflow graph must have at least one data source and one data sink.

Dataflow Graphs (Parallelism)



Task Execution



Data Parallelism and Task Parallelism

Data parallelism

- you can partition your input data and have tasks of the same operation execute on the data subsets in parallel
- useful because it allows for processing large volumes of data and spreading the computation load across several computing nodes

Task Parallelism

- you can have tasks from different operators performing computations on the same or different data in parallel
- Using task parallelism, you can better utilize the computing resources of a cluster

Latency and Throughput

Latency indicates how long it takes for an event to be processed

Throughput is a measure of the system's processing capacity—its rate of processing. That is, throughput tells us how many events the system can process per time unit

Latency and throughput are not independent metrics. If events take a long time to travel in the data processing pipeline, we cannot easily ensure high throughput. Similarly, if a system's capacity is small, events will be buffered and have to wait before they get processed.

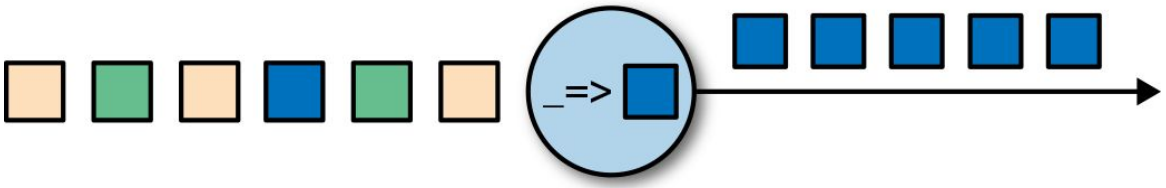
Operations on Data Streams

Data Ingestion and Egress

Data ingestion and data egress operations allow the stream processor to communicate with external systems.

Transformation Operations

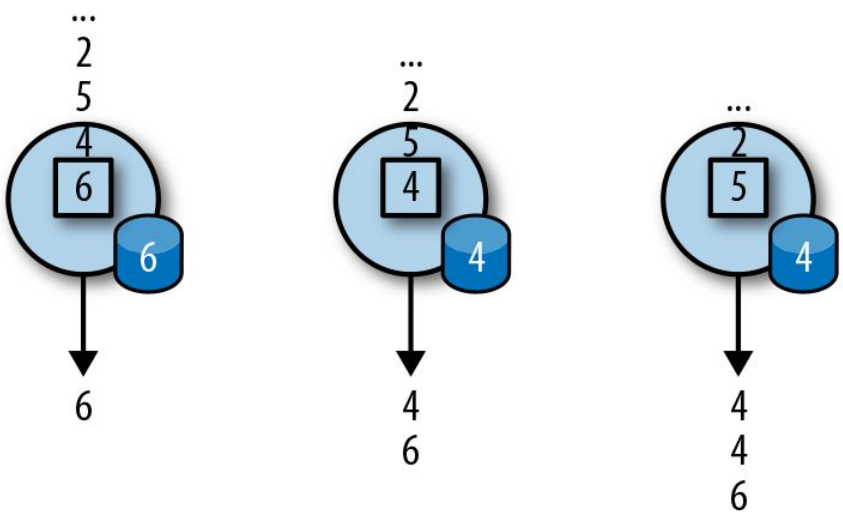
These operations consume one event after the other and apply some transformation to the event data, producing a new output stream.



Operations on Data Streams

Rolling Aggregations

A rolling aggregation is an aggregation, such as sum, minimum, and maximum, that is continuously updated for each input event.



Windowing Operator

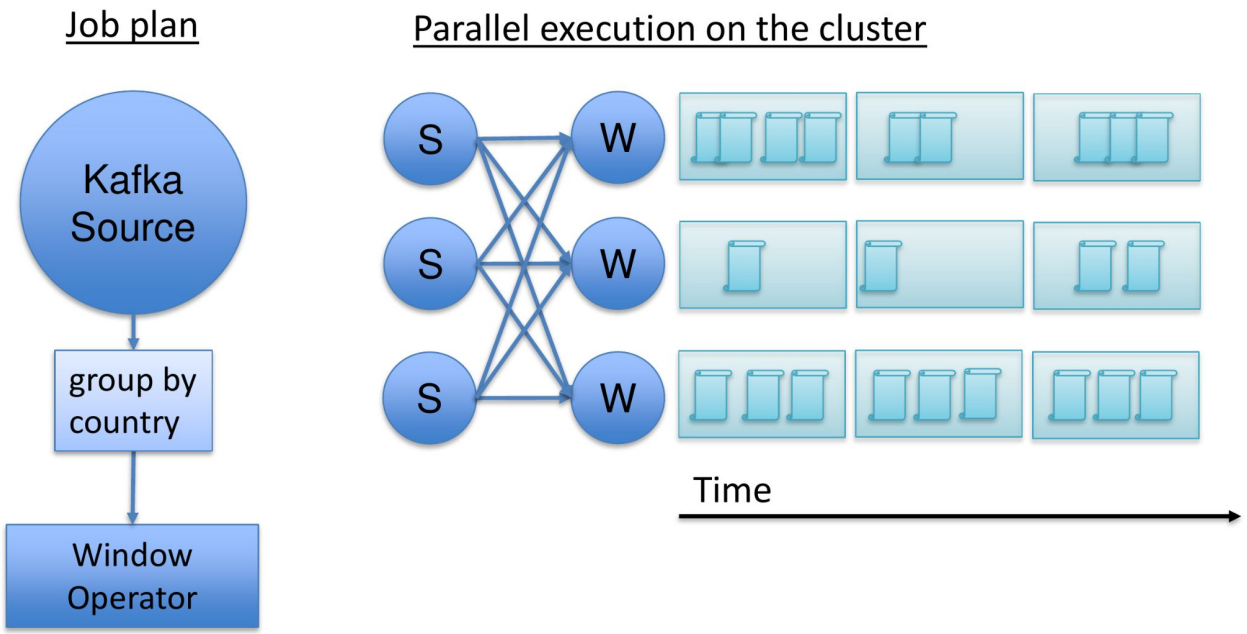
Building windows from a stream

“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());
```

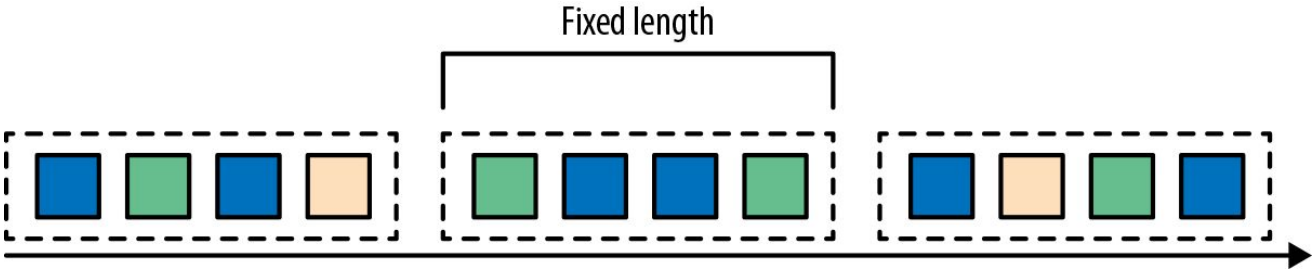
Building windows: Execution

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

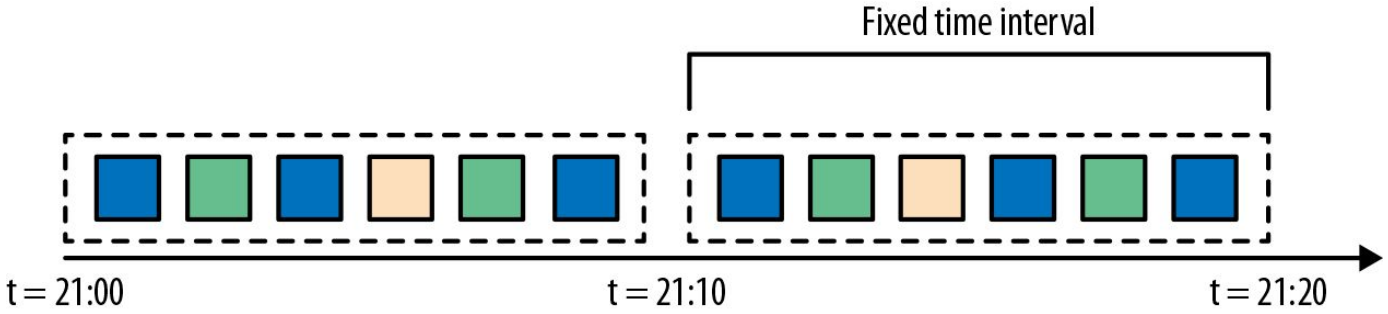


Window types in Flink (Tumbling)

- Count Based

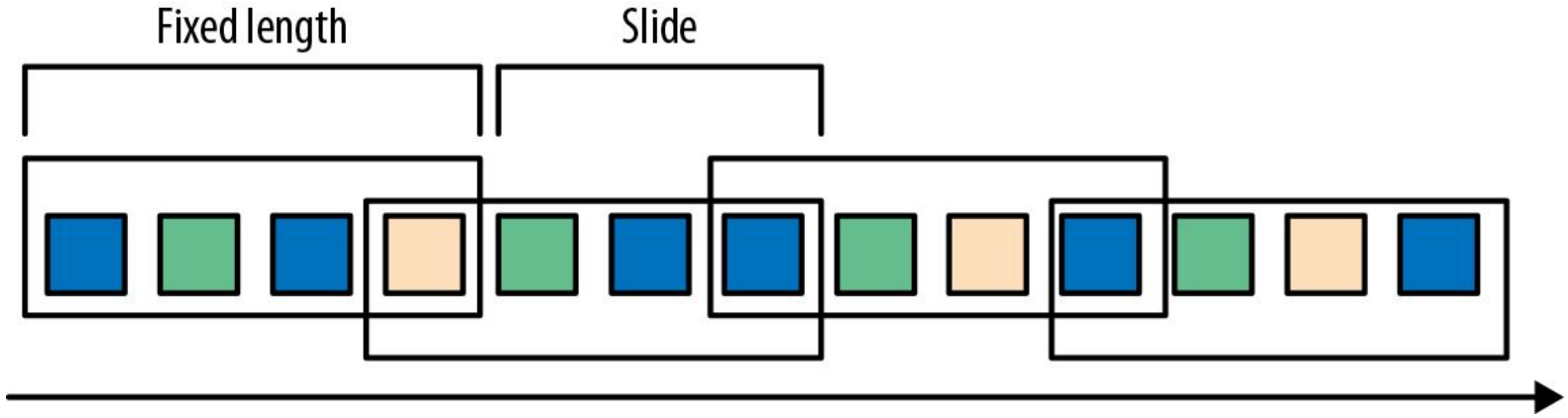


- Time Based



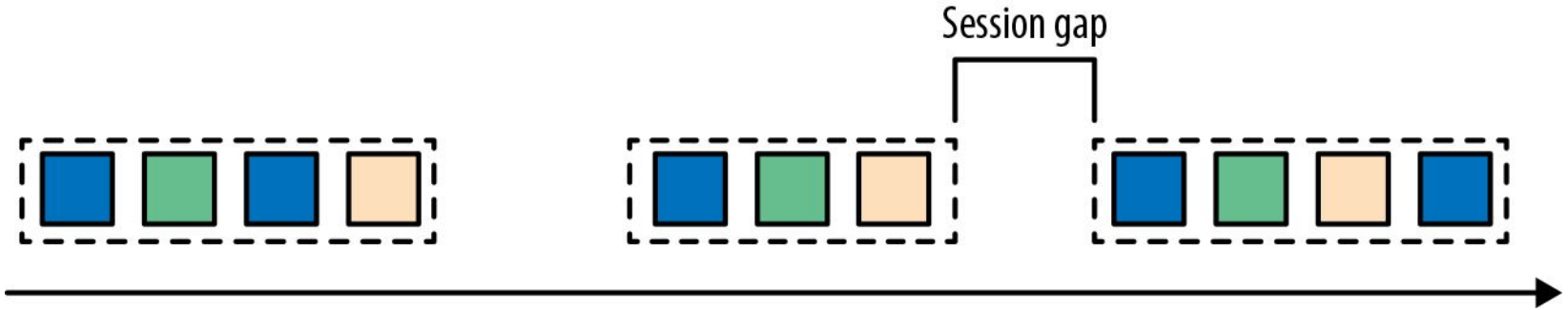
Sliding Windows

- **Sliding windows** assign events into overlapping buckets of fixed size. Thus, an event might belong to multiple buckets. We define sliding windows by providing their length and their slide.



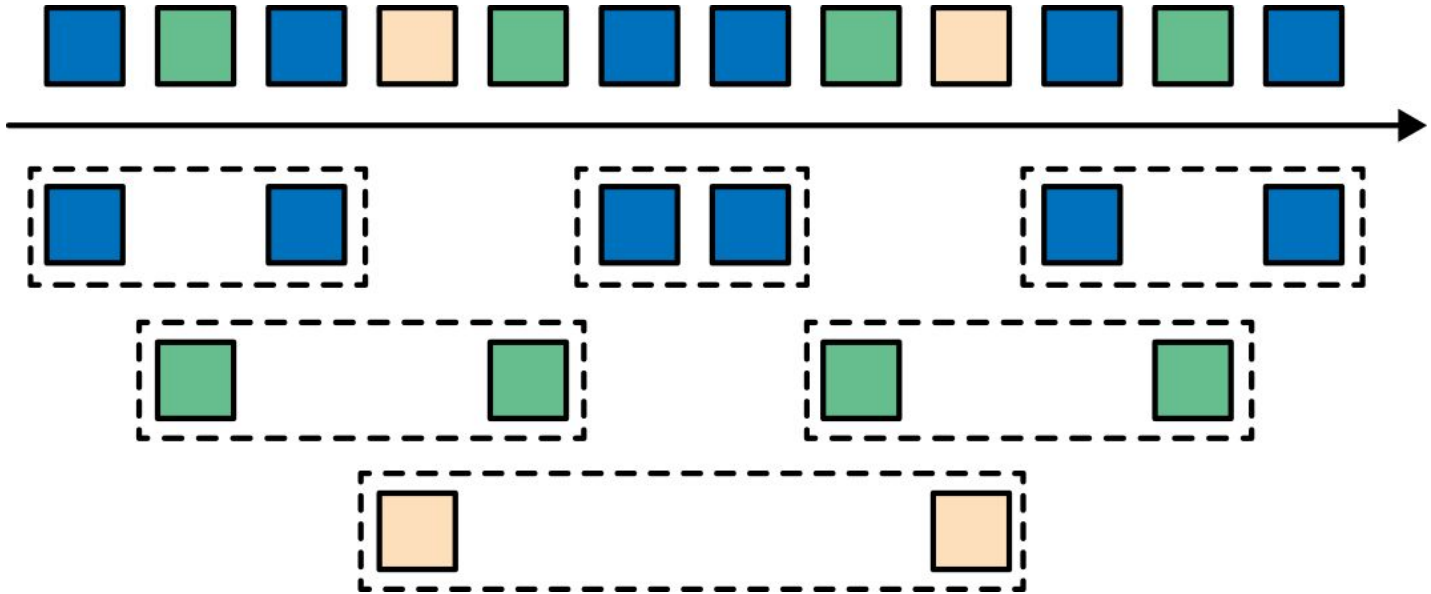
Session Windows

- **Session window** grouping a series of events happening in adjacent times followed by a period of inactivity



parallel windows

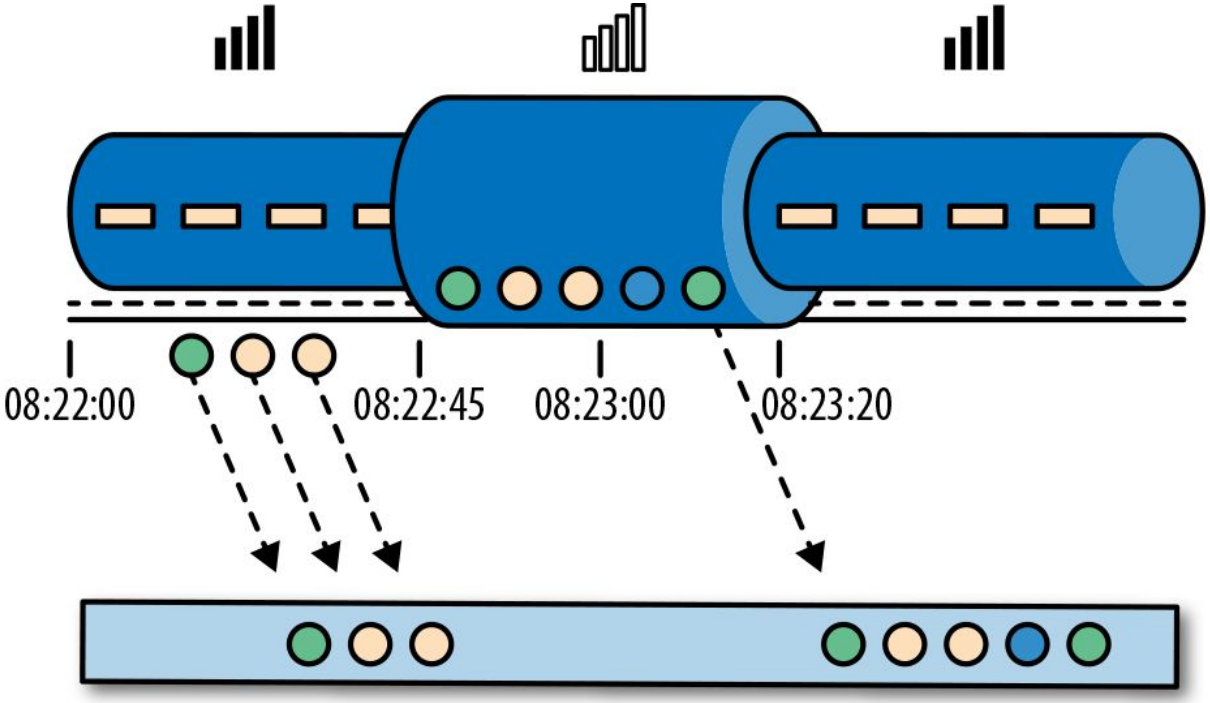
In parallel windows, each partition applies the window policies independently of other partitions.



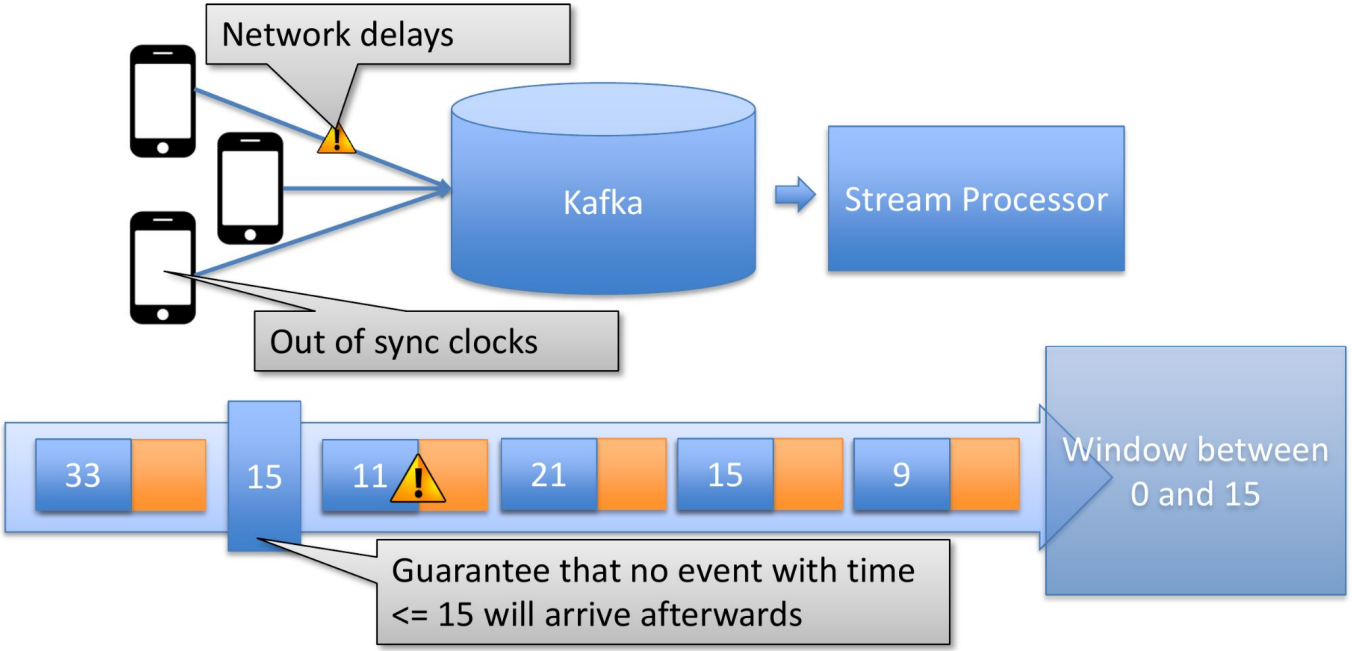
Time characteristics in Apache Flink

- **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**

Scenario: Application lost connections



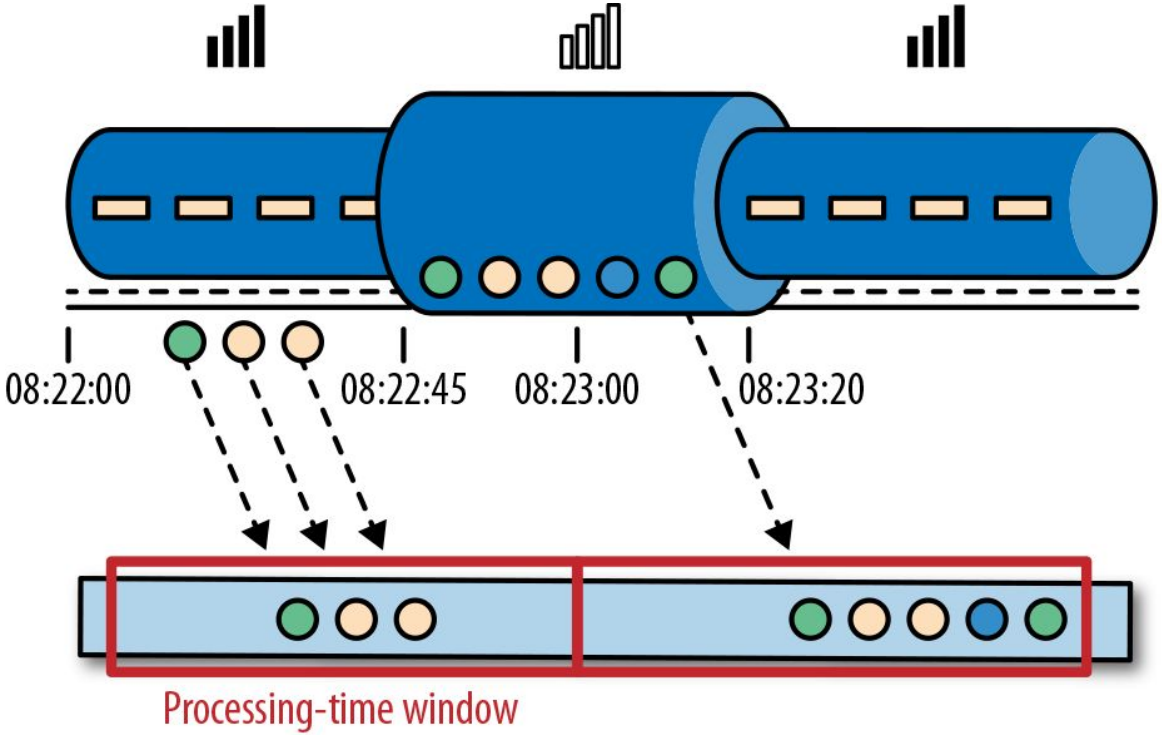
A look at the reality of time



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

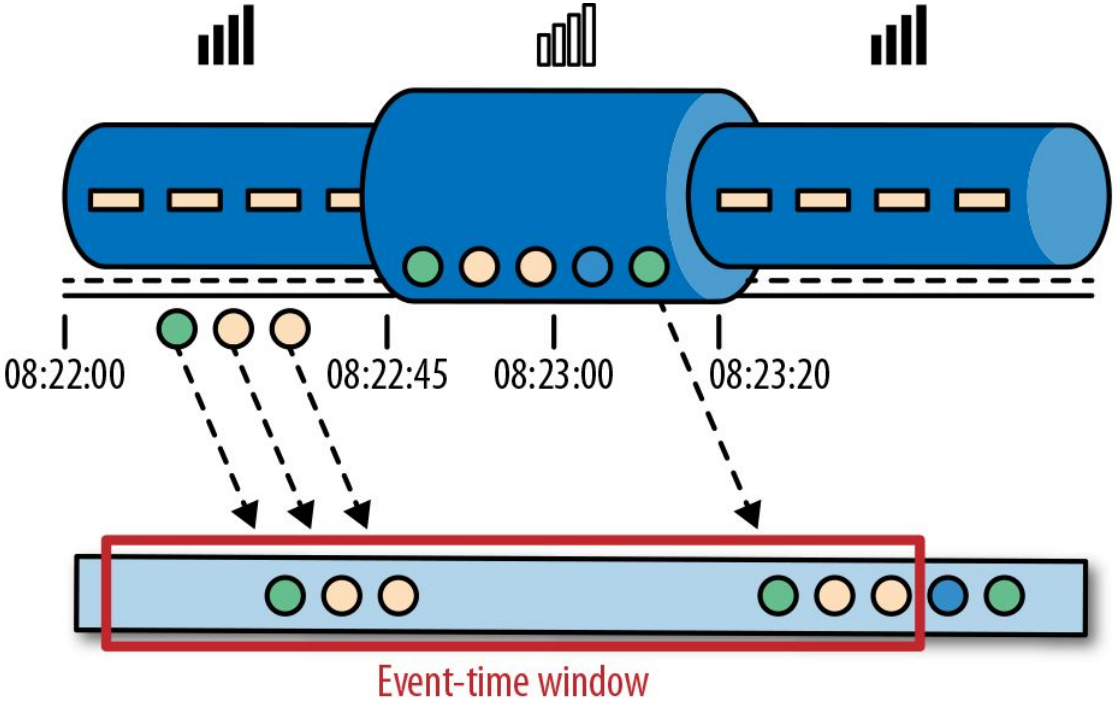
Procesing Time

Processing time is the time of the local clock on the machine where the operator processing the stream is being executed.



Event Time

Event time completely decouples the processing speed from the results. Operations based on event time are predictable and their results are deterministic.

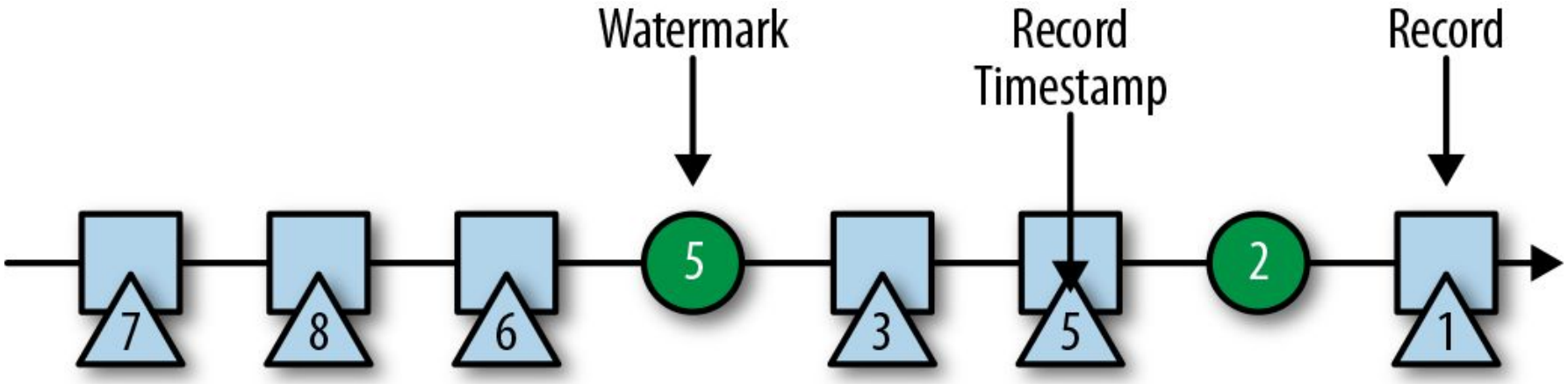


Watermarks

- A watermark is a global progress metric that indicates the point in time when we are confident that no more delayed events will arrive.
- When an operator receives a watermark with time T, it can assume that no further events with timestamp less than T will be received.
- Watermarks provide a configurable tradeoff between results confidence and latency.

```
DataStream<Event> stream = ...

WatermarkStrategy<Event> strategy = WatermarkStrategy
        .<Event>forBoundedOutOfOrderness(Duration.ofSeconds(20))
        .withTimestampAssigner((event, timestamp) -> event.timestamp);

DataStream<Event> withTimestampsAndWatermarks =
    stream.assignTimestampsAndWatermarks(strategy);
```

Watermark



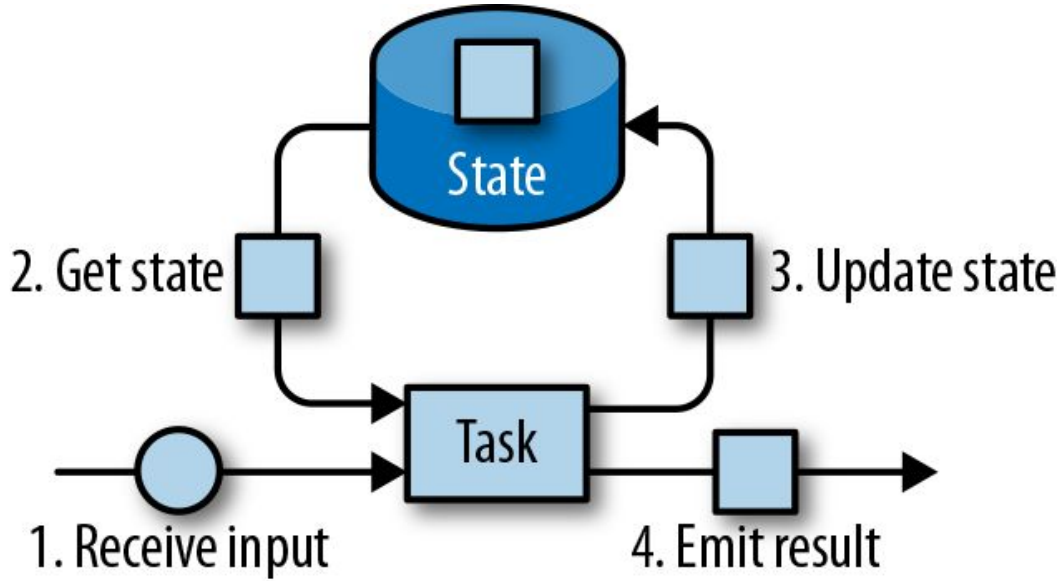
Processing Time vs Event Time

- Processing-time windows introduce the lowest latency possible
- processing time comes in handy. Another case is when you need to periodically report results in real time, independently of their accuracy
- An example application would be a real-time monitoring dashboard that displays event aggregates as they are received
- On the other hand, event time guarantees deterministic results and allows you to deal with events that are late or even out of order.

State handling

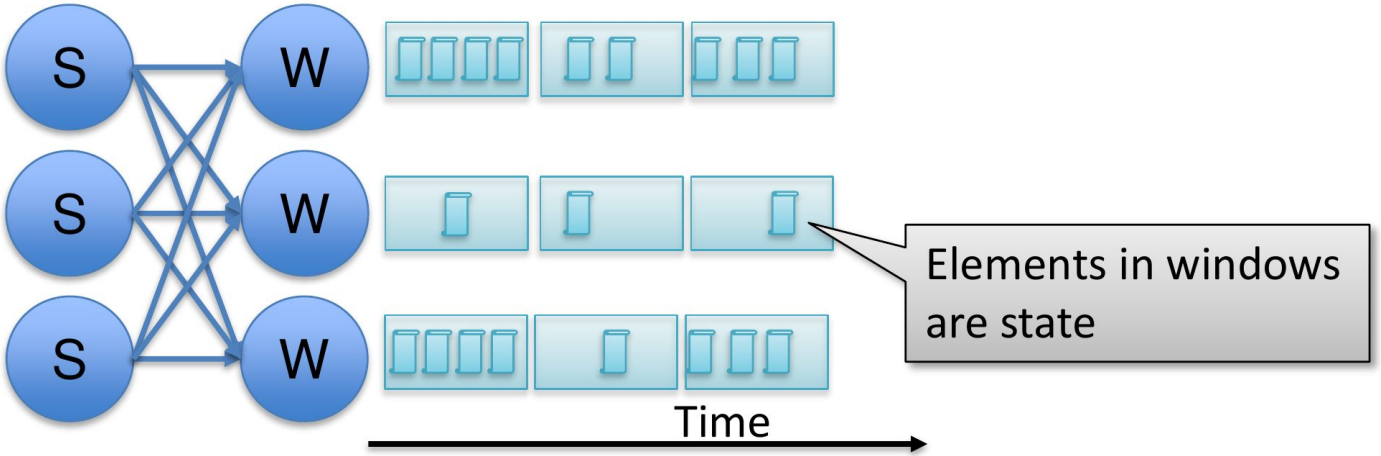
State Management

In general, all data maintained by a task and used to compute the results of a function belong to the state of the task. You can think of state as a local or instance variable that is accessed by a task's business logic.



State in streaming

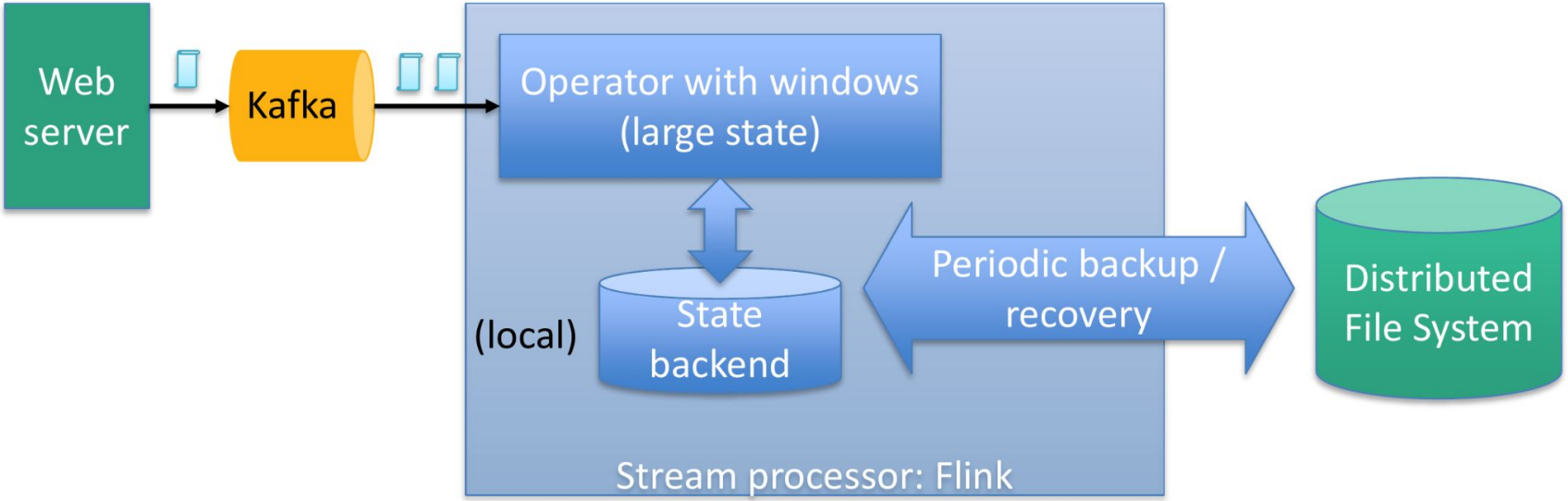
- Where do we store the elements from our windows?



- 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



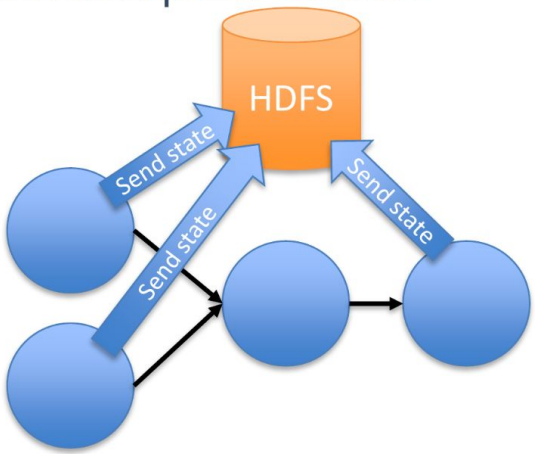
Managing the state

- 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

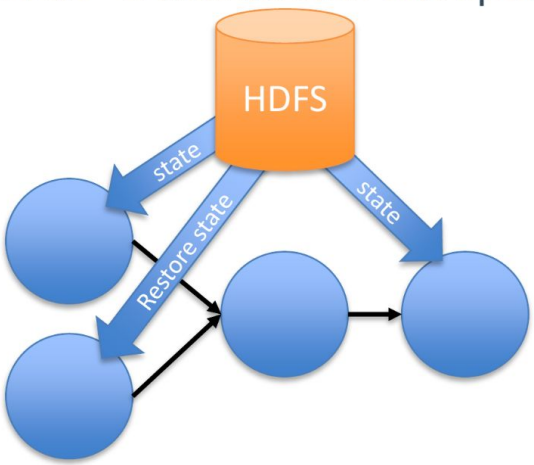
- Savepoint: Create an addressable copy of a job's current state.
- Restart a job from any savepoint.

> 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

Fault tolerance in streaming

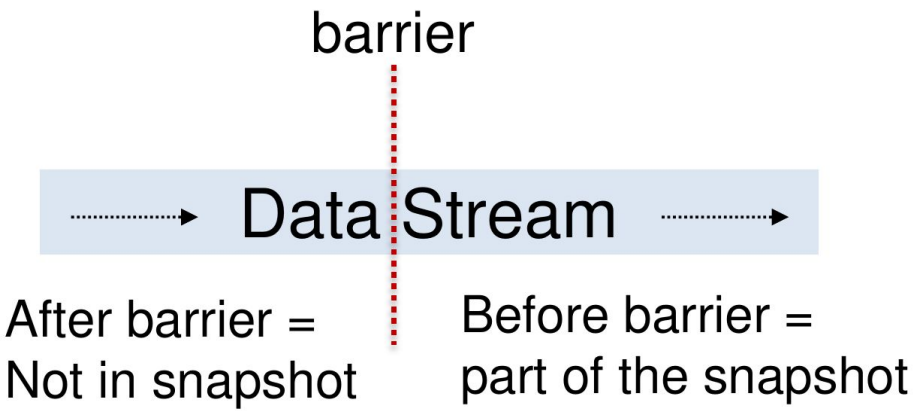
- How do we ensure the results (number of visitors) are always correct?
- Failures should not lead to data loss or incorrect results

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

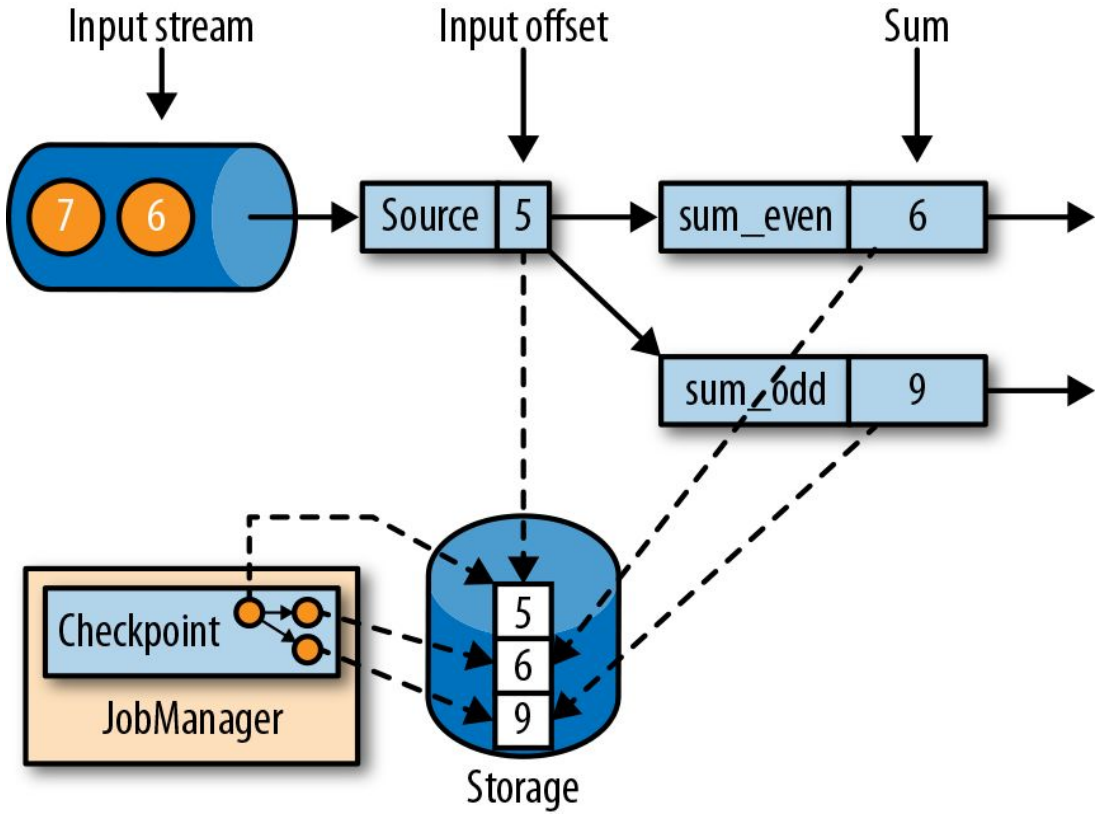
Flink’s Distributed Snapshots

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

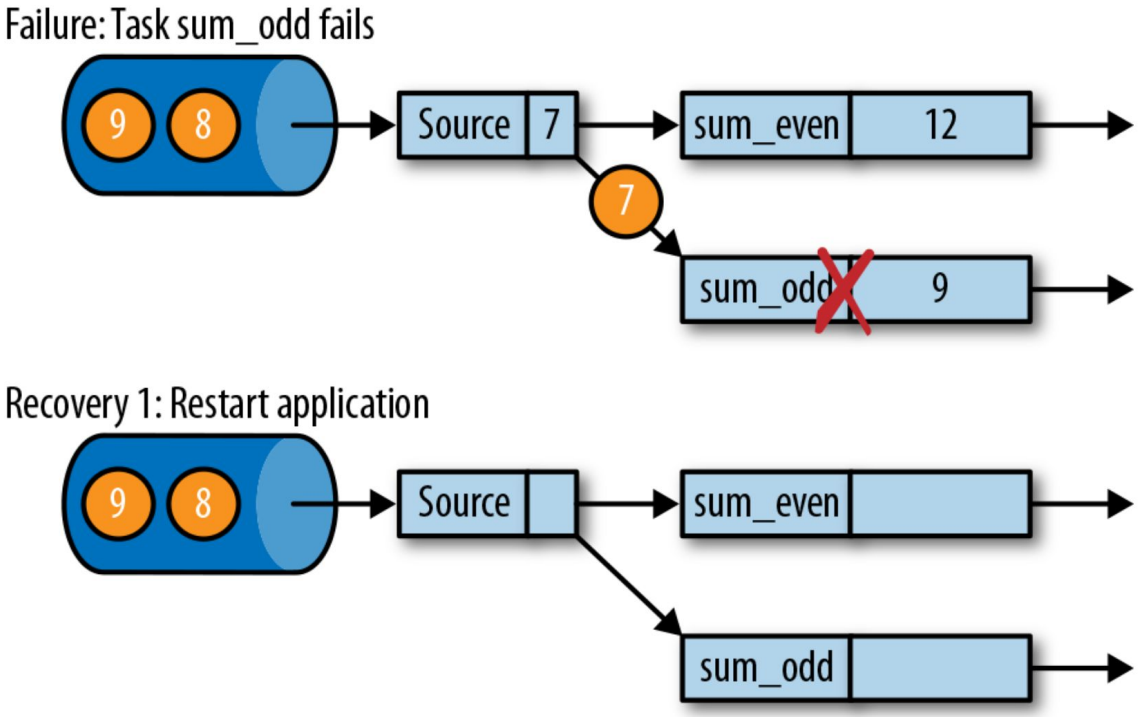


Further reading: http://blog.acolyer.org/2015/08/19/asynchronous-distributed-snapshots-for-distributed-dataflows/

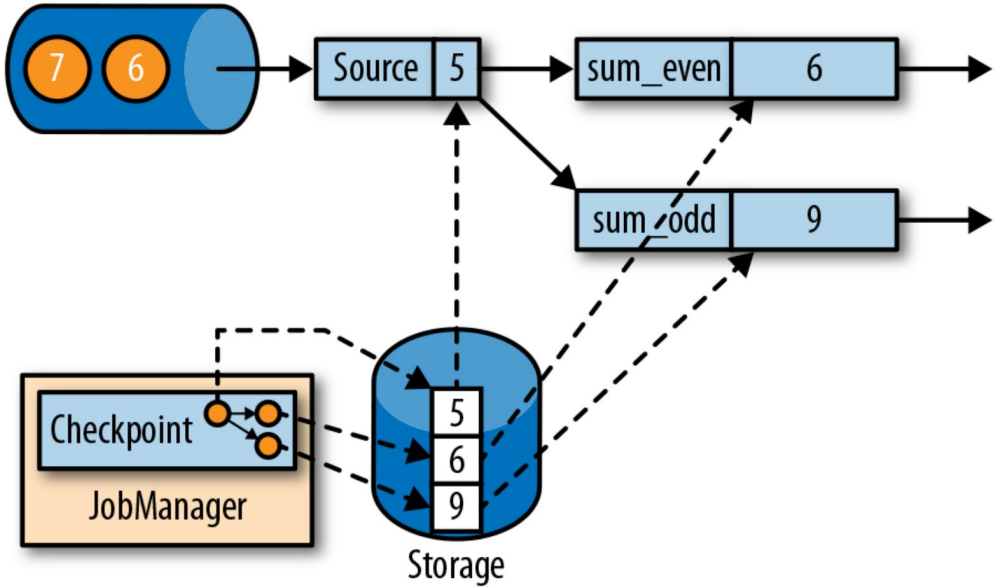
Consistent Checkpoints



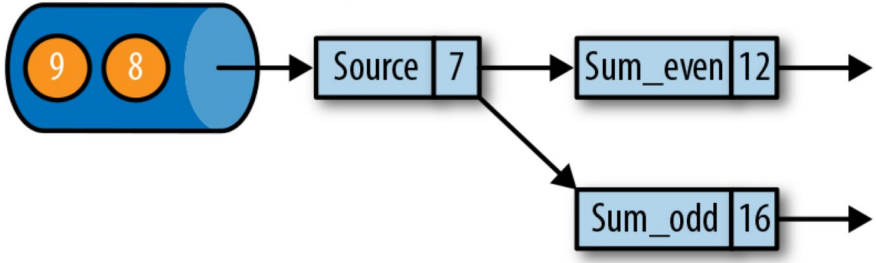
Recovery from a Consistent Checkpoint



Recovery 2: Reset application state from Checkpoint



Recovery 3: Continue processing



Savepoints

- Checkpoints are periodically taken and automatically discarded according to a configurable policy
- In principle, savepoints are created using the same algorithm as checkpoints and hence are basically checkpoints with some additional metadata.
- Flink does not automatically take a savepoint
- You can start a different but compatible application from a savepoint.
- You can start the same application with a different parallelism and scale the application out or in.

Savepoints

- You can start the same application on a different cluster. This allows you to migrate an application to a newer Flink version or to a different cluster or data-center.
- You can use a savepoint to pause an application and resume it later. This gives the possibility to release cluster resources for higher-priority applications or when input data is not continuously produced.
- You can also just take a savepoint to version and archive the state of an application.

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

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! Benchmark

“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.”

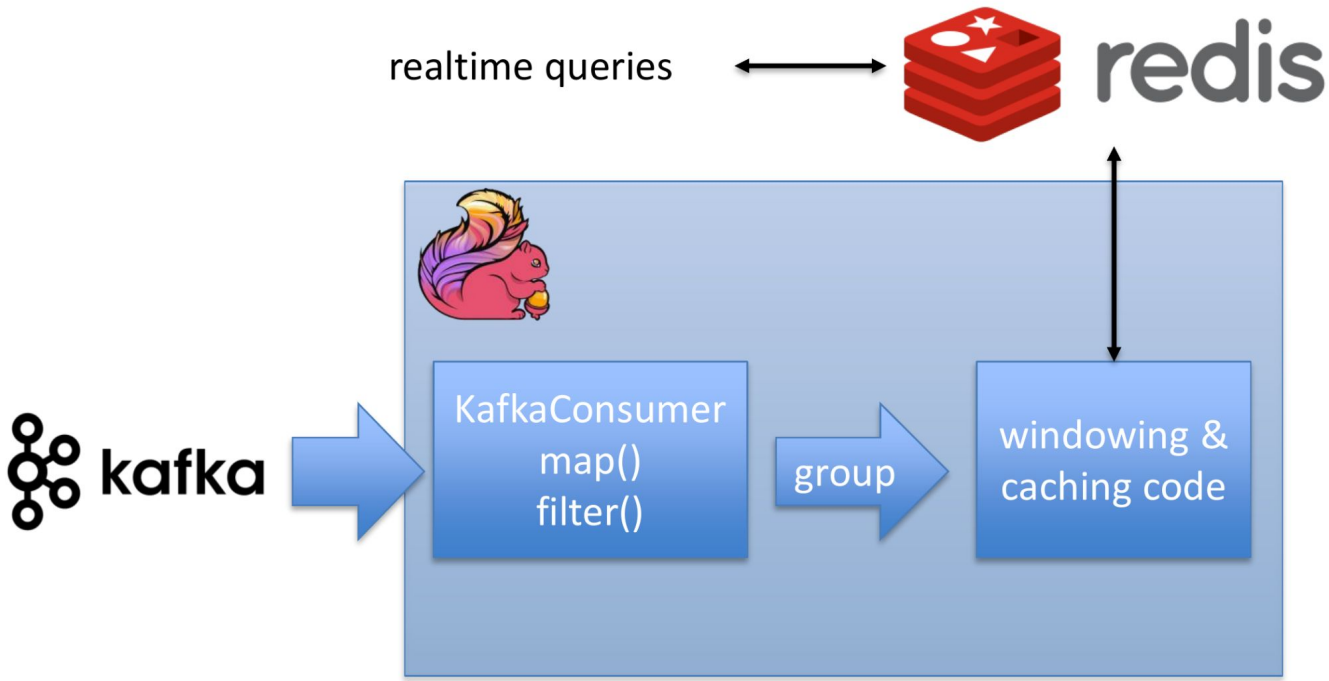
Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

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?

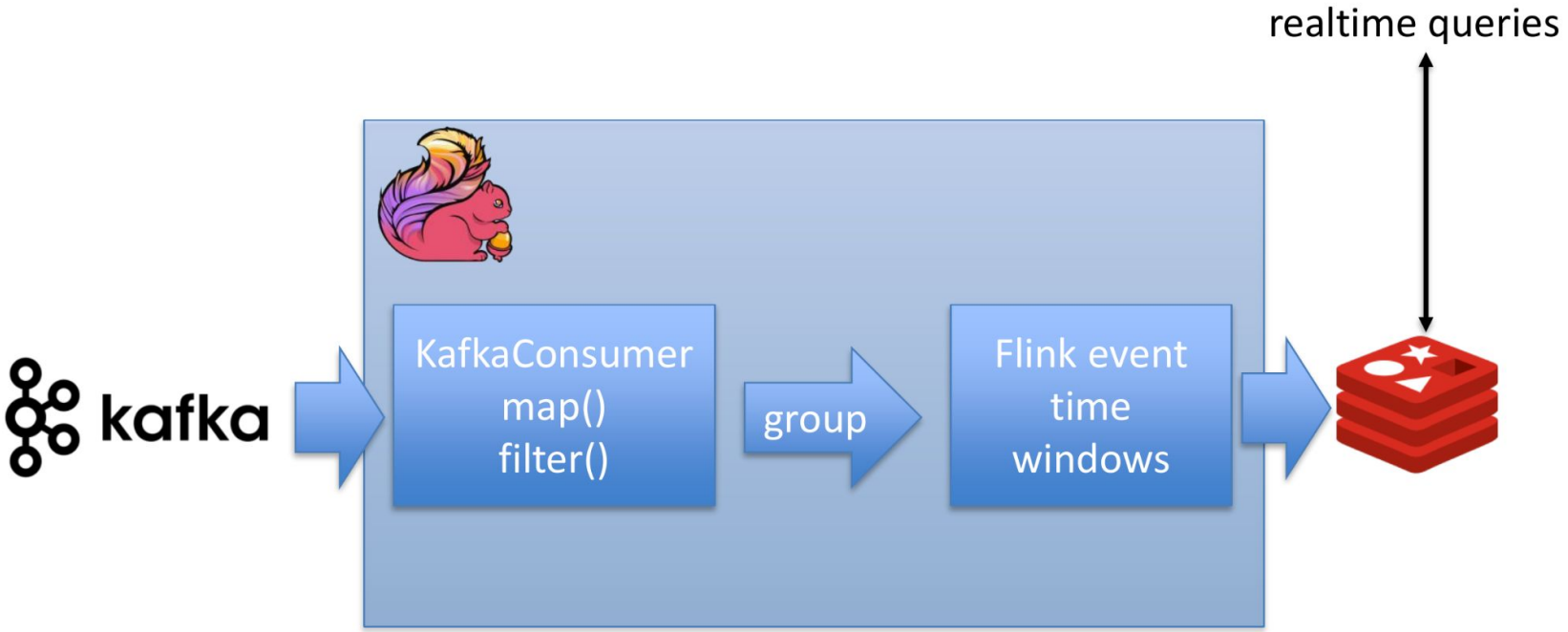
Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Windowing with state in Redis



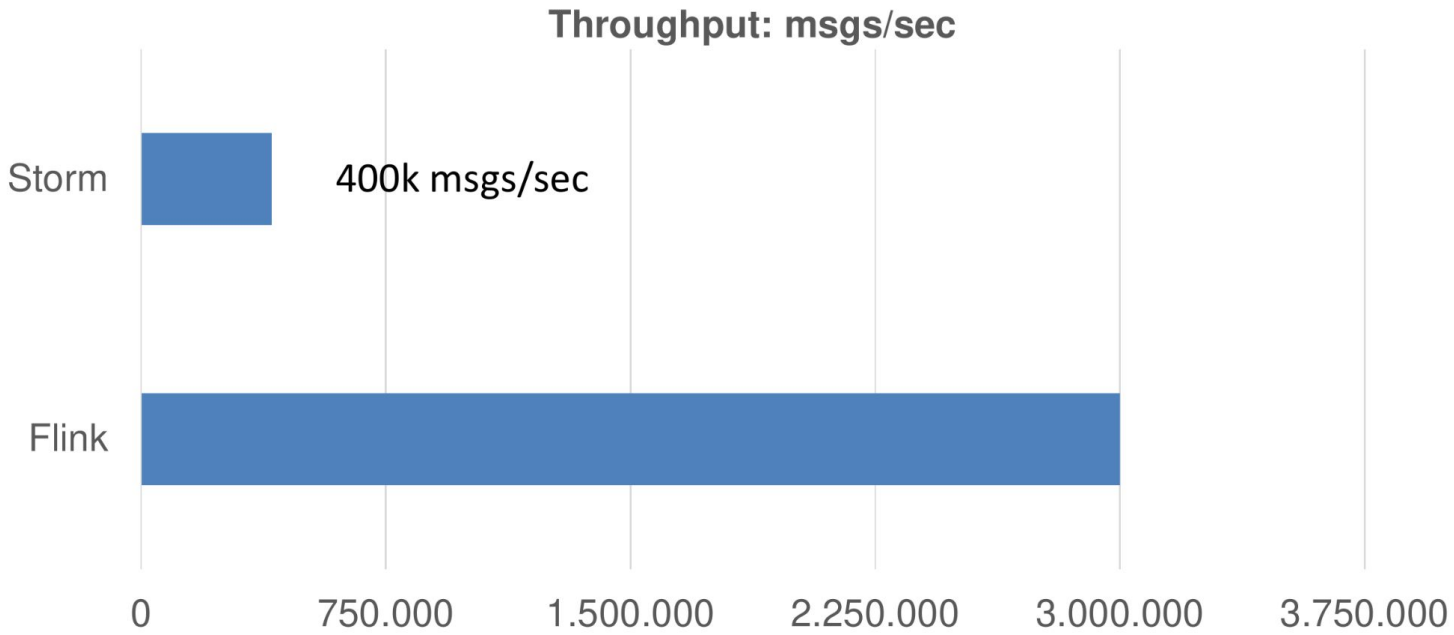
Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Rewrite to use Flink's own window



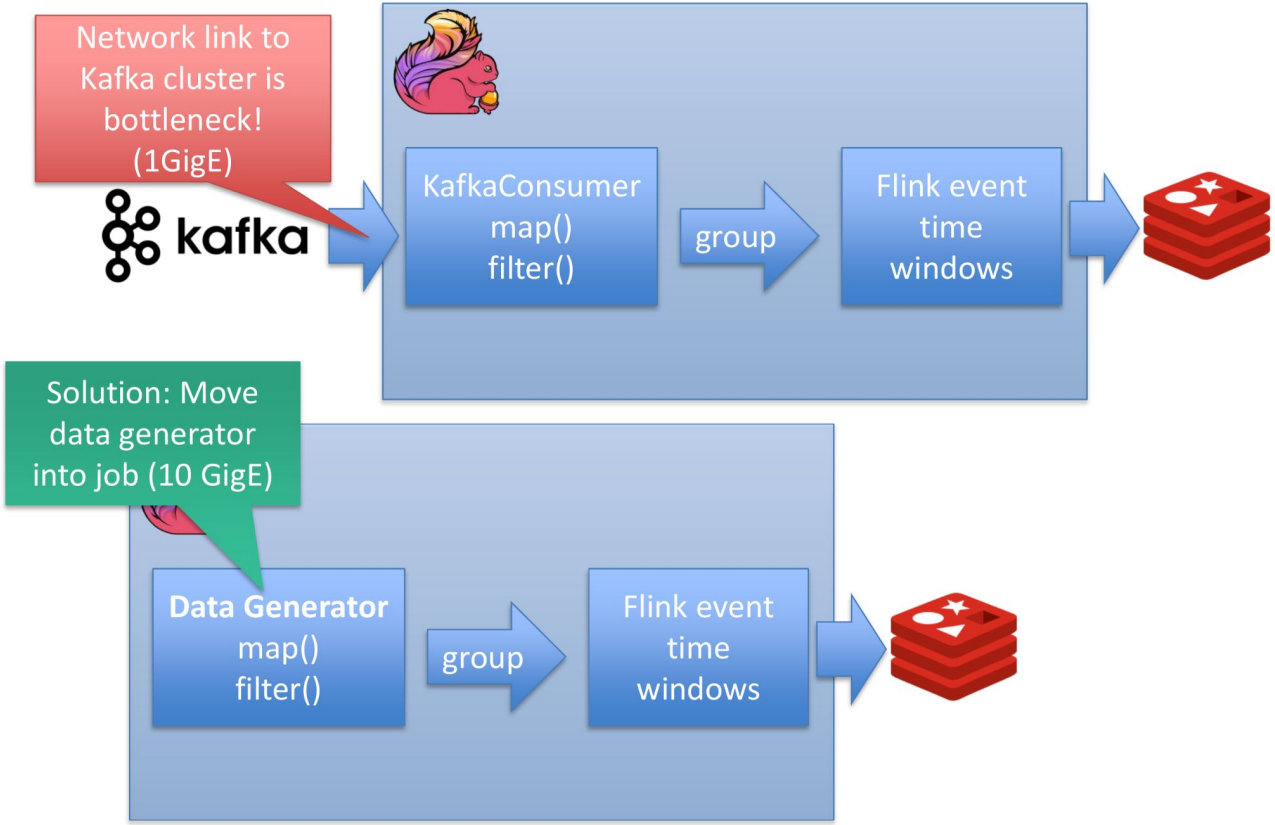
Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Results after rewrite

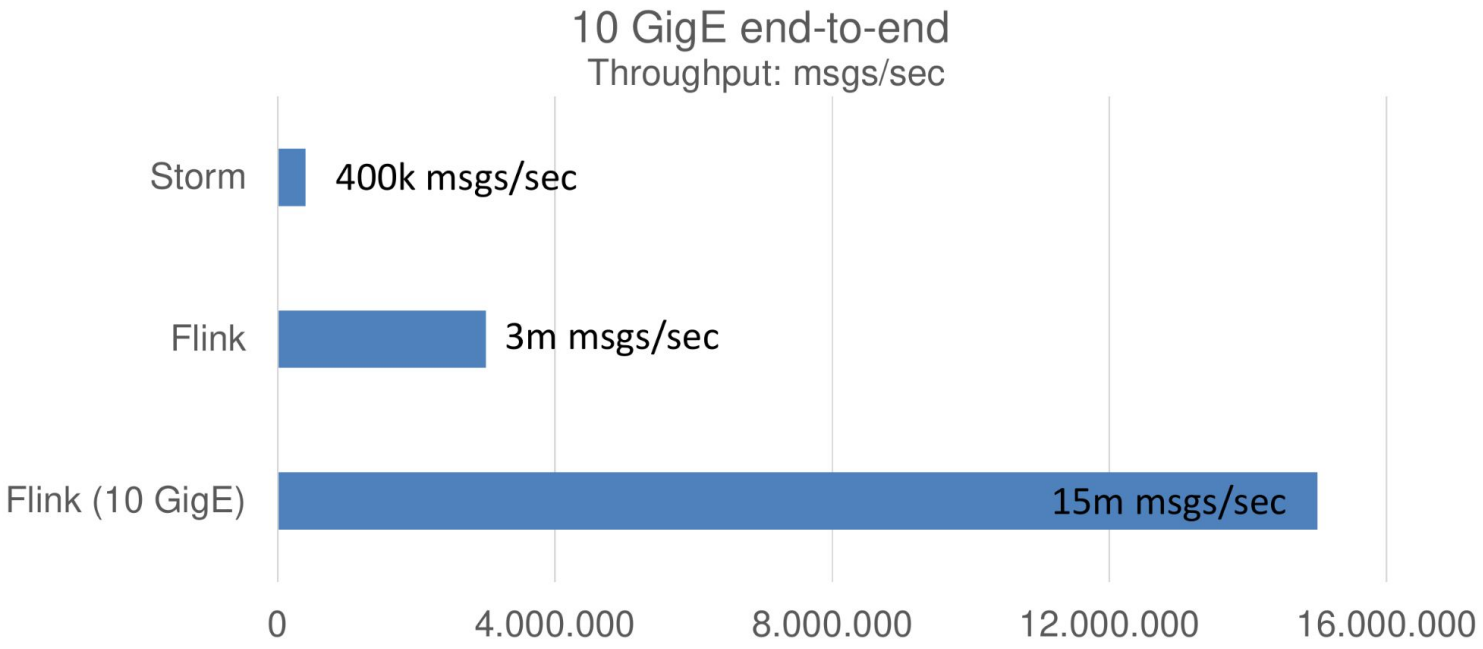


Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Can we even go further?



Results without network bottleneck



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

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/