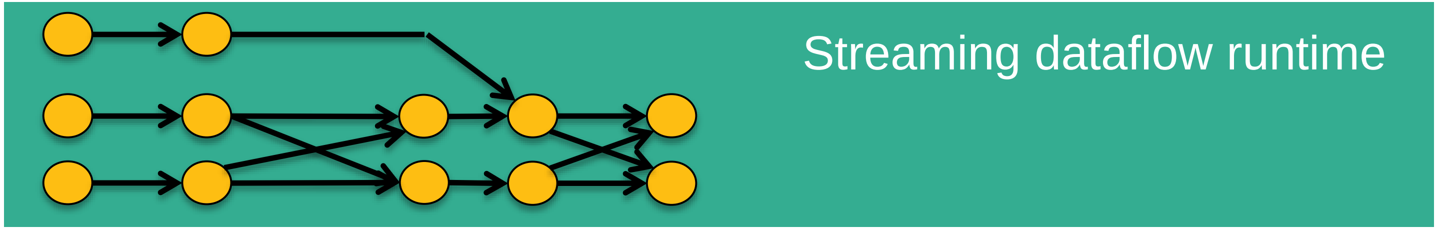
Apache Flink™: Stream and Batch processing at Scale

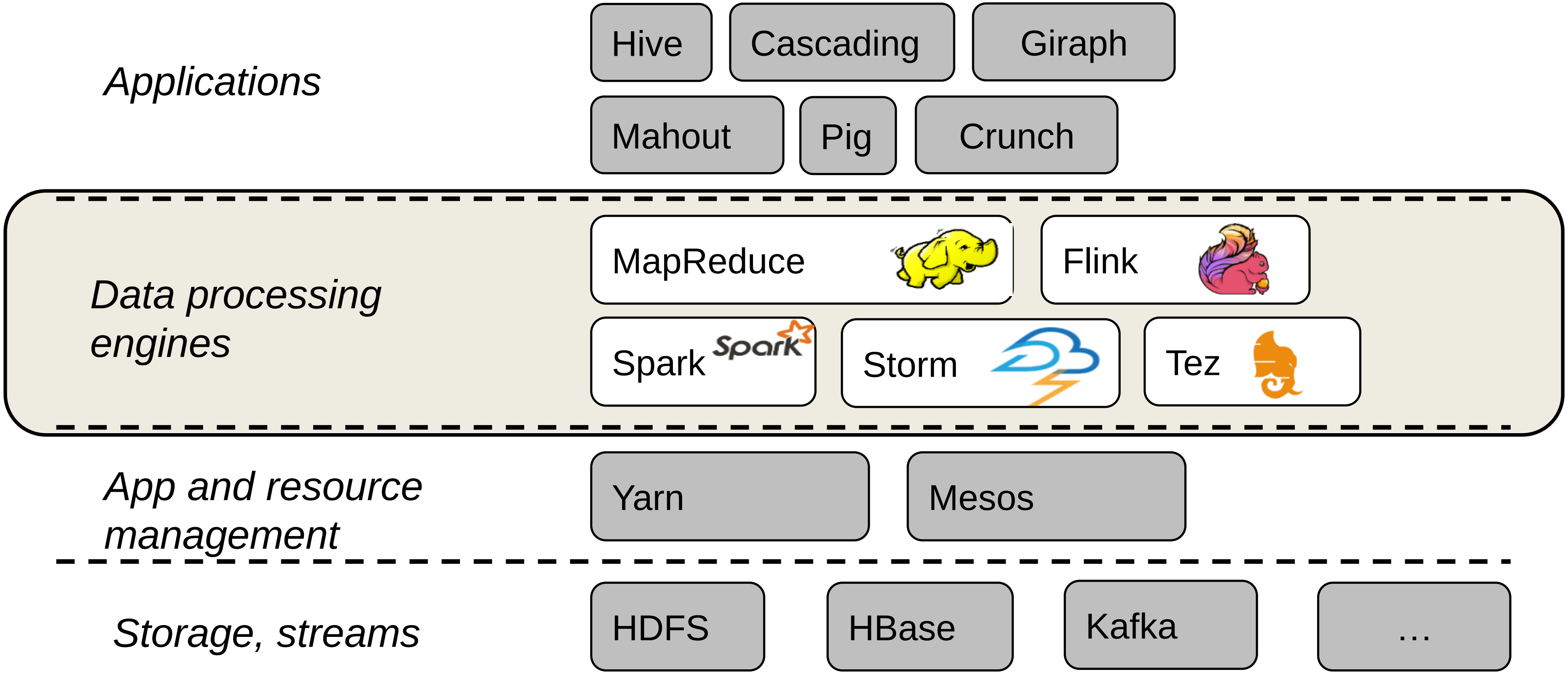
Marton Balassi (ELTE/SZTAKI, Hungary)
Paris Carbone (KTH, Stockholm, Sweden)
Gyula Fora (SICS, Stockholm, Sweden)
Vasia Kalavri (KTH, Stockholm, Sweden)
Asterios Katsifodimos (TU Berlin, Germany)

What is Flink?

A platform for distributed
batch and streaming analytics

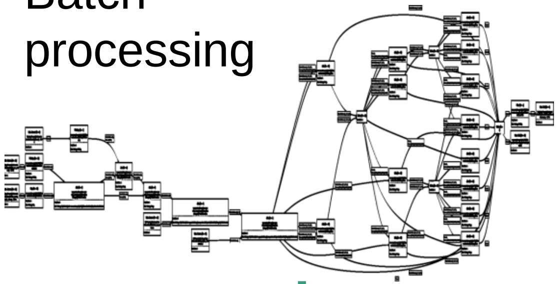


Flink in the Analytics Ecosystem

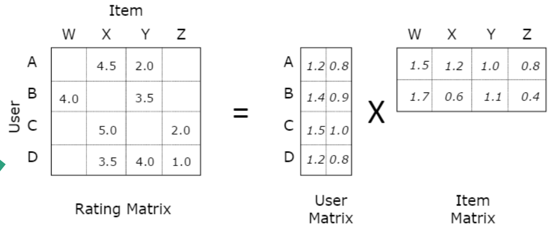


What can I do with it?

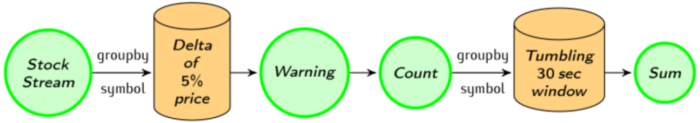
Batch processing



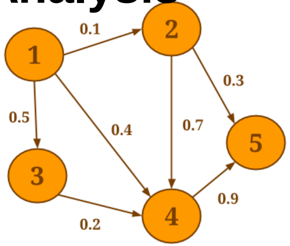
Machine Learning at scale



Stream processing



Graph Analysis



Flink

An engine that can **natively** support all these workloads.

Datasets and Streams

```
case class Word (word: String, frequency: Int)
```

DataSet API (batch):

```
val lines: DataSet[String] = env.readTextFile(...)
lines.flatMap {line => line.split(" ")
                                 .map(word => Word(word,1))}
      .groupBy("word").sum("frequency")
      .print()
```

DataStream API (streaming):

```
val lines: DataStream[String] = env.fromSocketStream(...)
lines.flatMap {line => line.split(" ")
                                 .map(word => Word(word,1))}
       .keyBy("word")
       .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS))
       .sum("frequency”)
       .print()
```

Tables and Graphs

Table API (SQL):

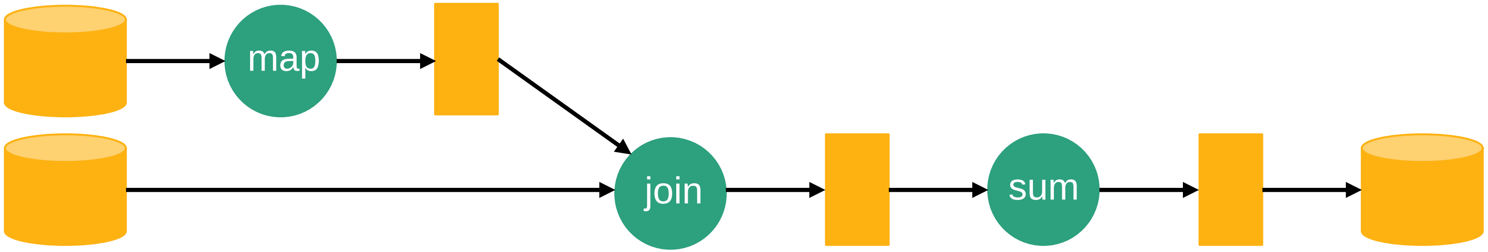
```
case class Result(a: String, d: Int)
val input1 = env.fromElements(...).toTable('a, 'b)
val input2 = env.fromElements(...).toTable('c, 'd)
val joined = input1.join(input2).where("b = a && d > 42").select("a,d").toDataSet[Result]
```

Gelly API (Graphs):

```
Graph<Long, Long, NullValue> graph = ...

DataSet<Vertex<Long, Long>> verticesWithCommunity = graph.run(
                      new LabelPropagation<Long>(30)).getVertices();

verticesWithCommunity.print();
```

Execution Model

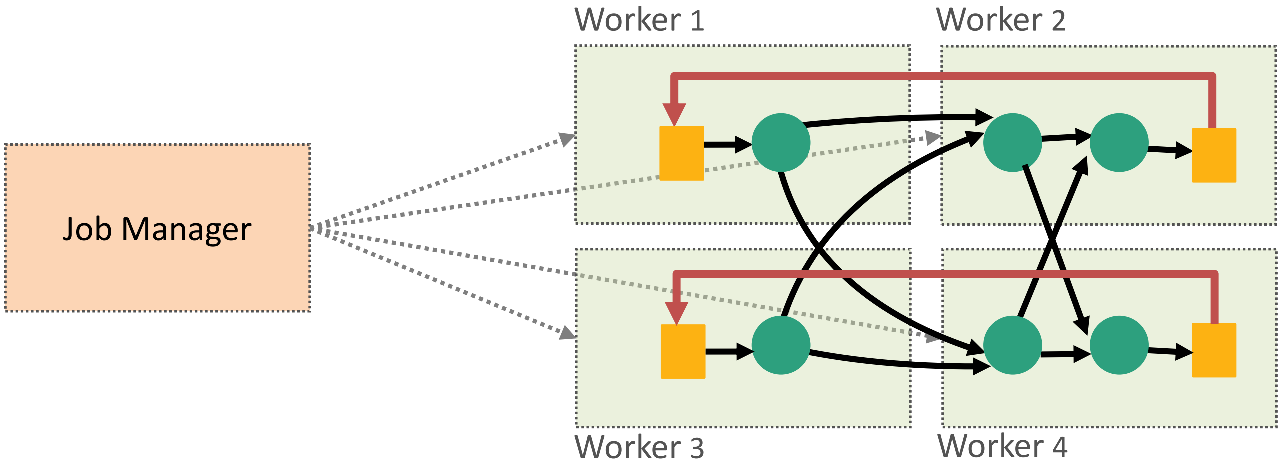
- Flink program = DAG* of operators and intermediate results
- Operator = computation + state
- Intermediate result = logical stream of records



Architecture

- Hybrid MapReduce and MPP database runtime

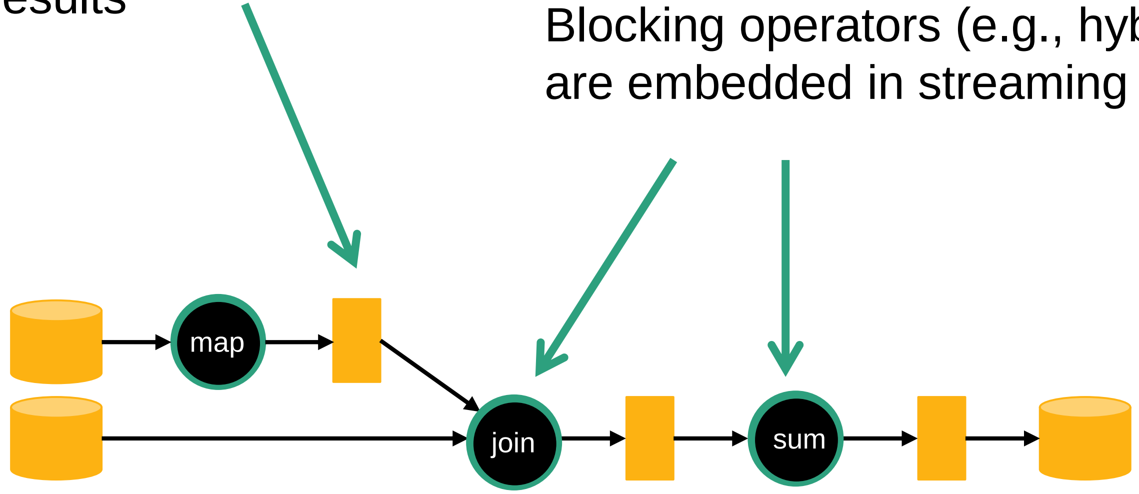
- Pipelined/Streaming engine
  - Complete DAG deployed



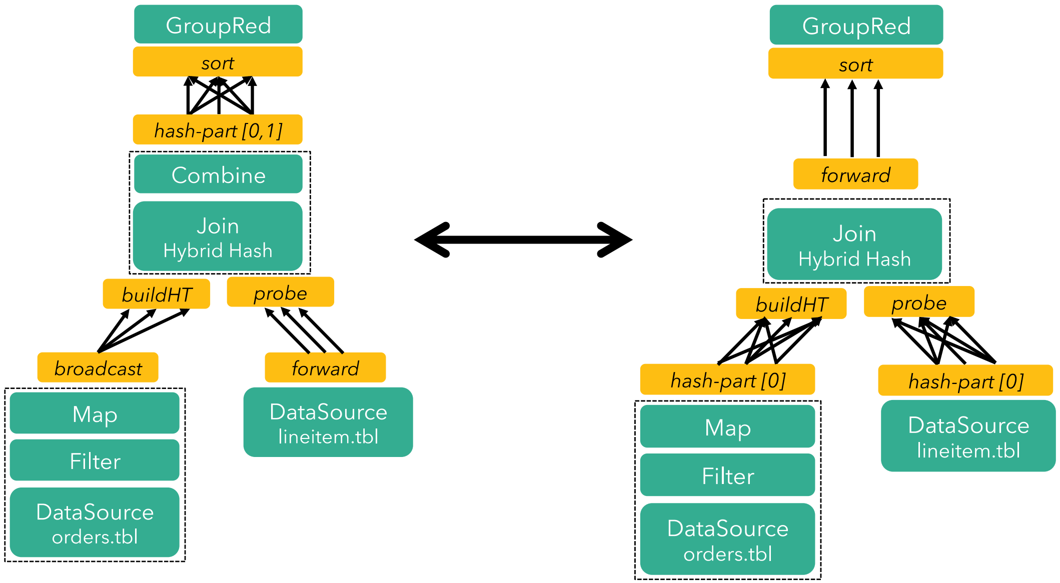
Batch is a Special Case of Streaming

Lower-overhead fault-tolerance via replaying intermediate results

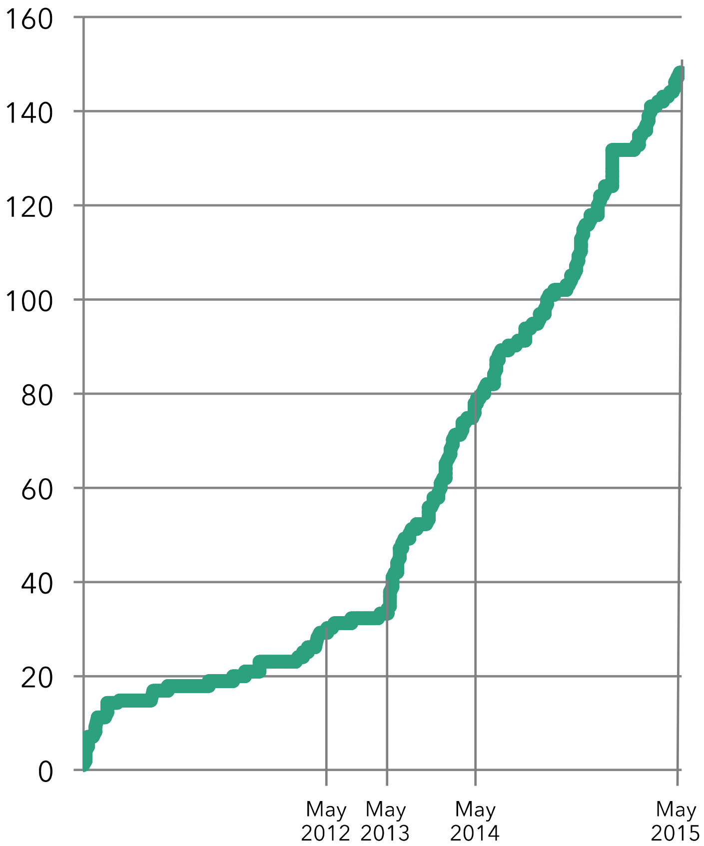
Blocking operators (e.g., hybrid hash join) are embedded in streaming topology



Cost-based optimizer



Community



Flink started as the Stratosphere project in in 2009, led by TU Berlin. KTH, ELTE/SZTAKI, and companies are contributing to the code.

Entered incubation April 2014 graduated on December 2014.

Now one of the most active big data projects after over a year in the Apache Software Foundation.

Flink Forward

BERLIN 12/13 OCT 2015

http://flink-forward.org

Today

- **Introduction**
  - 15’ Overview
  - 15' Gelly (Graph) API
- **30' Break**
- **Graph Processing**
  - 20' DataSet/Gelly Hands-on
- **Stream processing with Flink**
  - 10’ DataStream API
  - 15’ Fault Tolerance Demo
  - 45' Streaming Hands-on

Appendix

FlinkML

- API for ML pipelines inspired by scikit-learn
- Collection of packaged algorithms
  - SVM, Multiple Linear Regression, Optimization, ALS, ...

```
val trainingData: DataSet[LabeledVector] = ...
val testingData: DataSet[Vector] = ...

val scaler = StandardScaler()
val polyFeatures = PolynomialFeatures().setDegree(3)
val mlr = MultipleLinearRegression()

val pipeline = scaler.chainTransformer(polyFeatures).chainPredictor(mlr)

pipeline.fit(trainingData)

val predictions: DataSet[LabeledVector] = pipeline.predict(testingData)
```

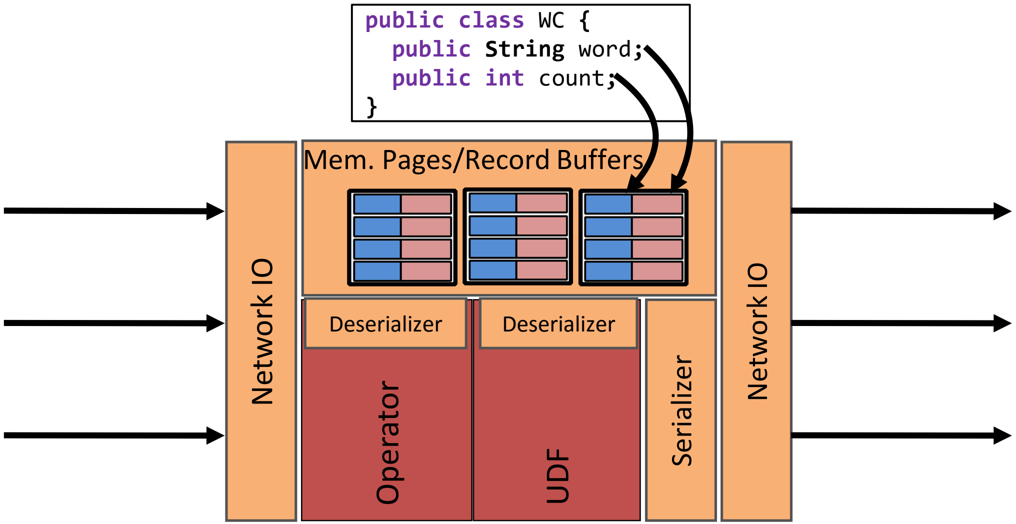
Gelly

- Graph API and library
- Packaged algorithms
  - PageRank, SSSP, Label Propagation, Community Detection, Connected Components

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

Graph<Long, Long, NullValue> graph = ...

DataSet<Vertex<Long, Long>> verticesWithCommunity = graph.run(
                             new LabelPropagation<Long>(30)).getVertices();

verticesWithCommunity.print();

env.execute();
```

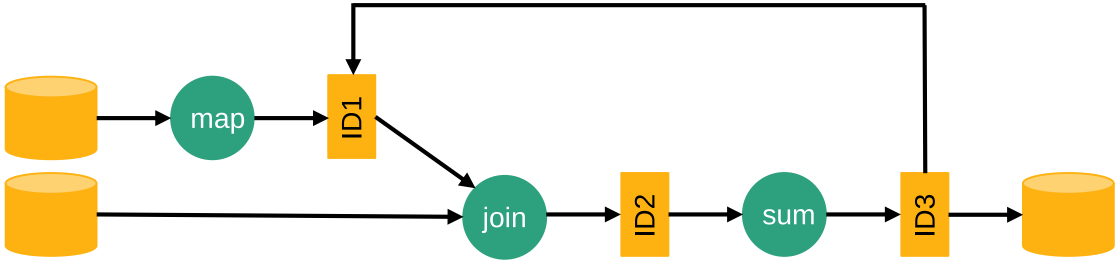
Managed Memory

- Language APIs automatically converts objects to tuples
  - Tuples mapped to pages/buffers of bytes
  - Operators can work on pages/buffers
- Full control over memory, out-of-core enabled
- Operators (e.g., Hybrid Hash Join) address individual fields (not deserialize object): robust



Iterative processing in Flink

Flink offers built-in iterations and delta iterations to execute ML and graph algorithms efficiently



Exactly once approaches

- Discretized streams – mini-batching (Spark Streaming)
  - Treat streaming as a series of small atomic computations
  - “Fast track” to fault tolerance, but does not separate business logic from recovery

- MillWheel (Google Cloud Dataflow)
  - State update and derived events committed as atomic transaction to a high-throughput transactional store
  - Needs a very high-throughput transactional store ☺

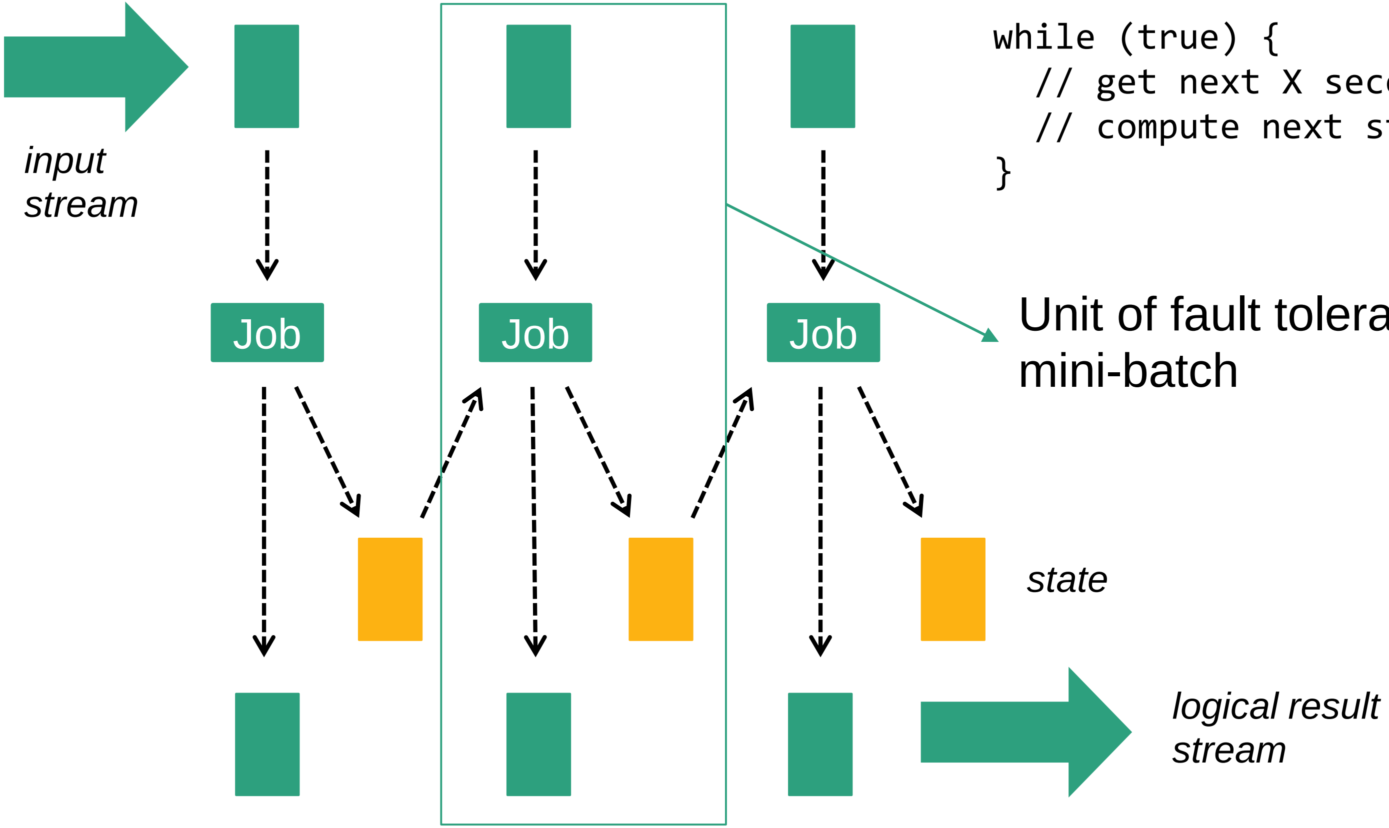
- Chandy-Lamport-inspired distributed snapshots (Flink)*

*[Carbone et. al. 2015] “Lightweight Asynchronous Snapshots for Distributed Dataflows”, Tech. Report. http://arxiv.org/abs/1506.08603

Roadmap

- Short-term (3-6 months)
  - Graduate DataStream API from beta
  - Fully managed window and user-defined state with pluggable backends
  - Table API for streams (towards StreamSQL)
- Long-term (6+ months)
  - Highly available master
  - Dynamic scale in/out
  - FlinkML and Gelly for streams
  - Full batch + stream unification

Discretized streams




```
while (true) {
  // get next X seconds of data
  // compute next stream and state
}
```

Unit of fault tolerance is mini-batch

Problems of mini-batch

- Latency
  - Each mini-batch schedules a new job, loads user libraries, establishes DB connections, etc
- Programming model
  - Does not separate business logic from recovery – changing the mini-batch size changes query results
- Power
  - Keeping and updating state across mini-batches only possible by immutable computations

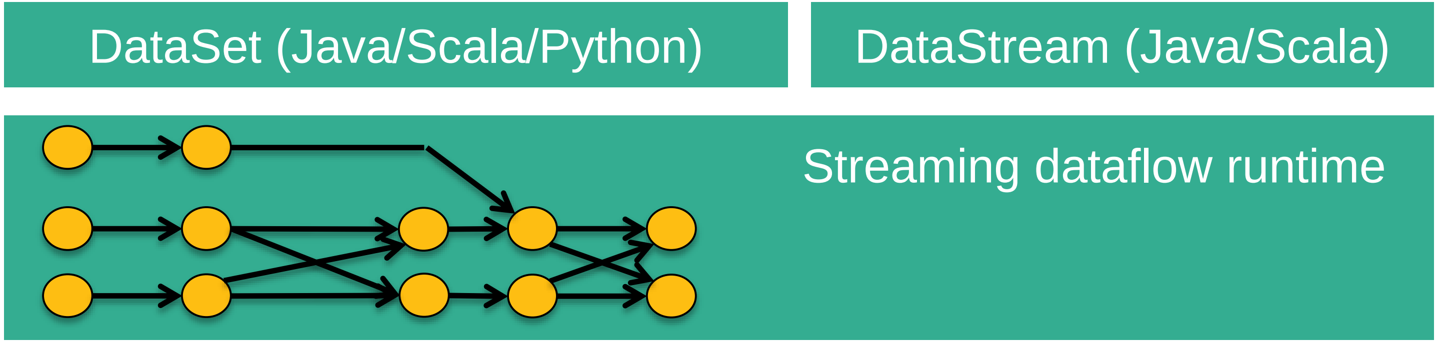
Exactly once approaches

- Discretized streams – mini-batching (Spark Streaming)
  - Treat streaming as a series of small atomic computations
  - “Fast track” to fault tolerance, but does not separate business logic from recovery
- MillWheel (Google Cloud Dataflow)
  - State update and derived events committed as atomic transaction to a high-throughput transactional store
  - Needs a very high-throughput transactional store ☺
- Chandy-Lamport-inspired distributed snapshots (Flink)*

*[Carbone et. al. 2015] “Lightweight Asynchronous Snapshots for Distributed Dataflows”, Tech. Report. http://arxiv.org/abs/1506.08603

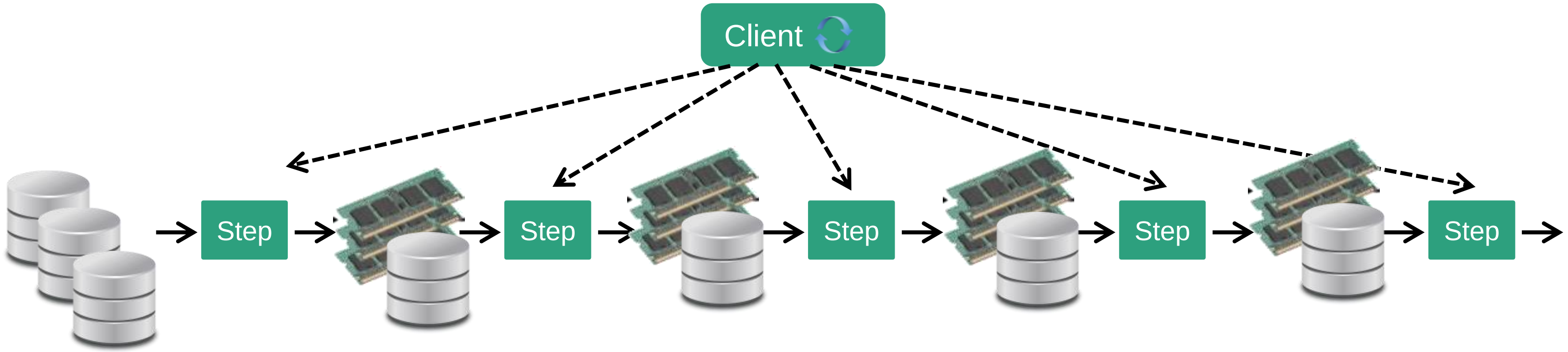
Integration with batch

- Currently cannot mix DataSet & DataStream programs
- However, DataStream programs can read batch sources, they are just finite streams ☺
- Goal is to evolve DataStream to a batch/stream-agnostic API



e.g.: Non-native iterations

```
for (int i = 0; i < maxIterations; i++) {
    // Execute MapReduce job
}
```



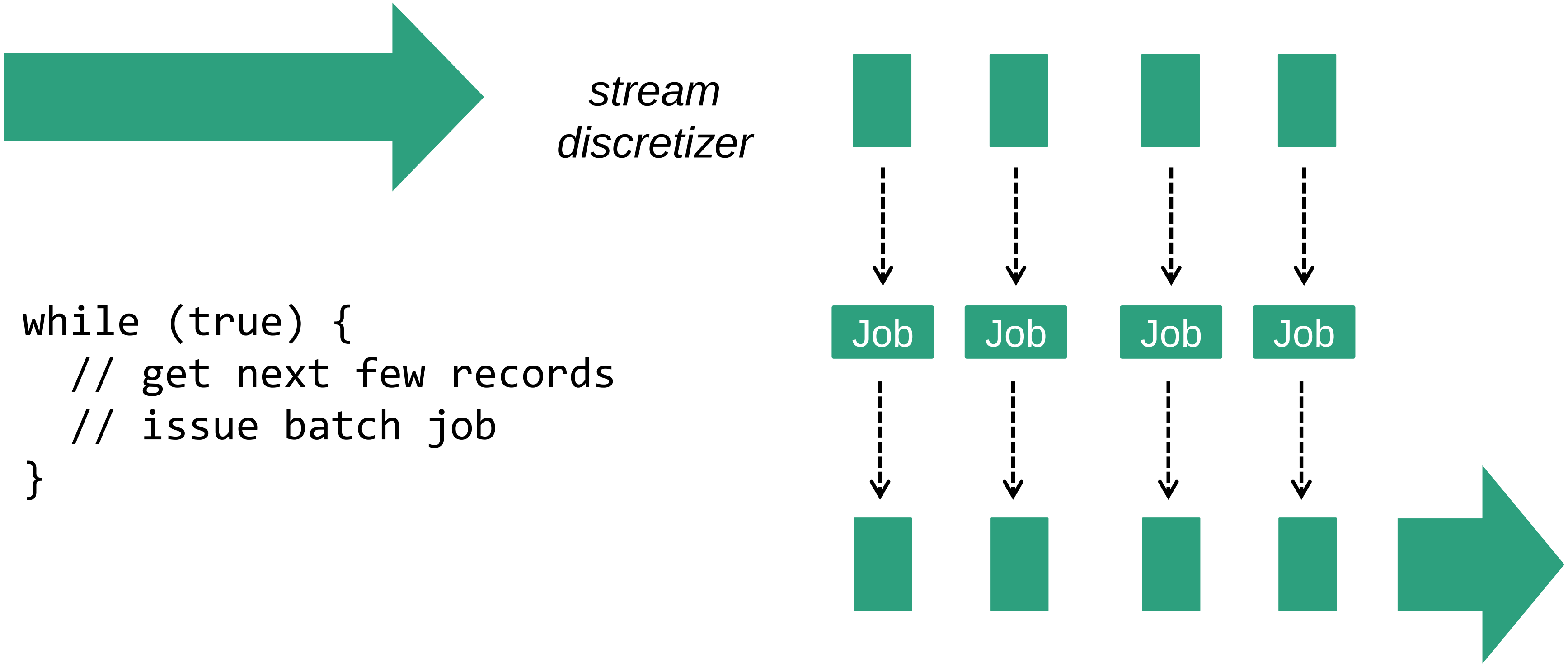
What is Operator State?

- User-defined state
  - Objects in Flink long running operators (map/reduce/etc)

- Windowing operators
  - Time, count, data-driven, etc. window discretizers

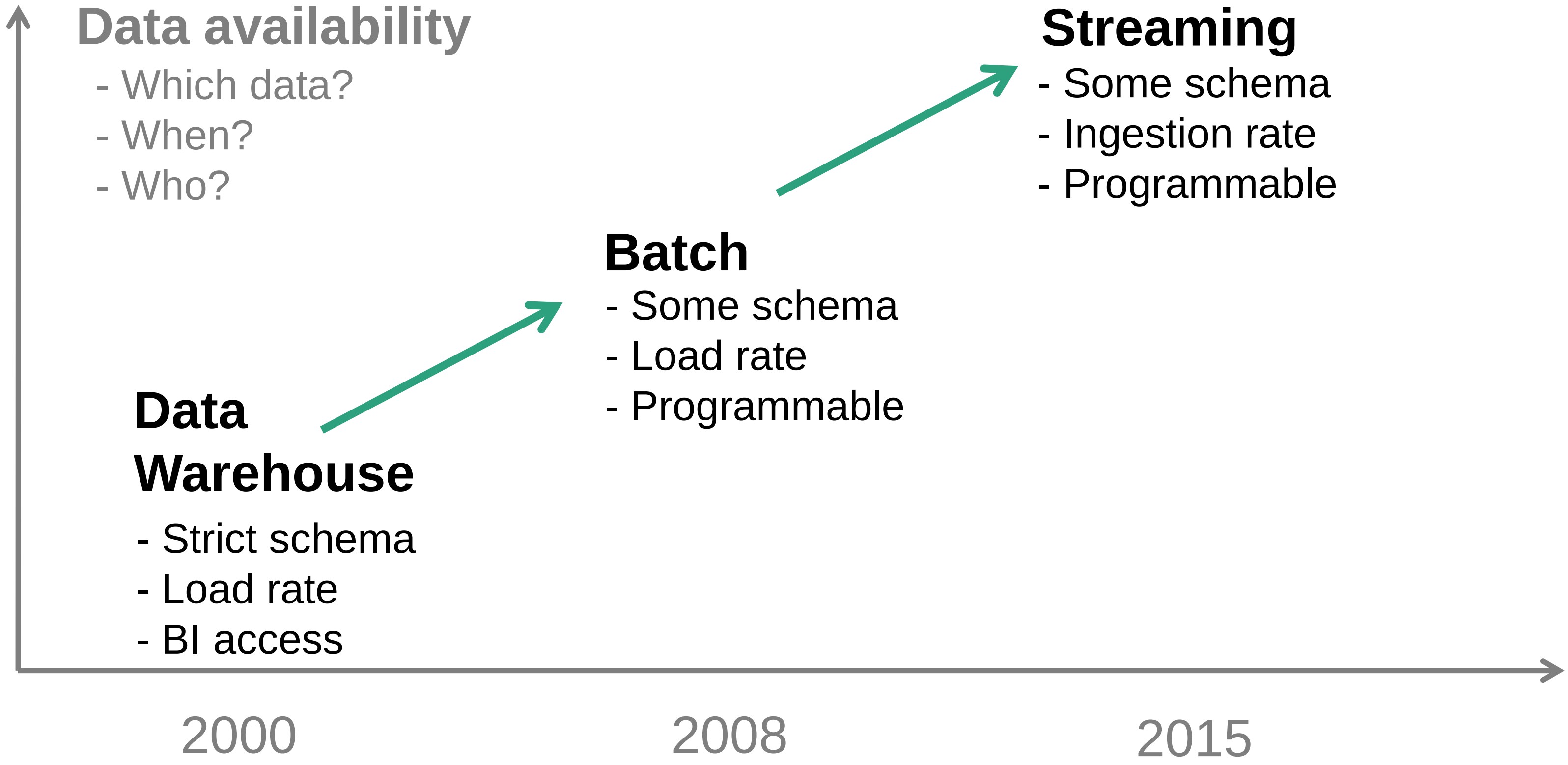
- Fault tolerance mechanism:
  - Back up and restored state stored in a backend (HDFS, Ignite, Cassandra, …)
  - After restore: replay stream from the last checkpoint

e.g.: Non-native streaming



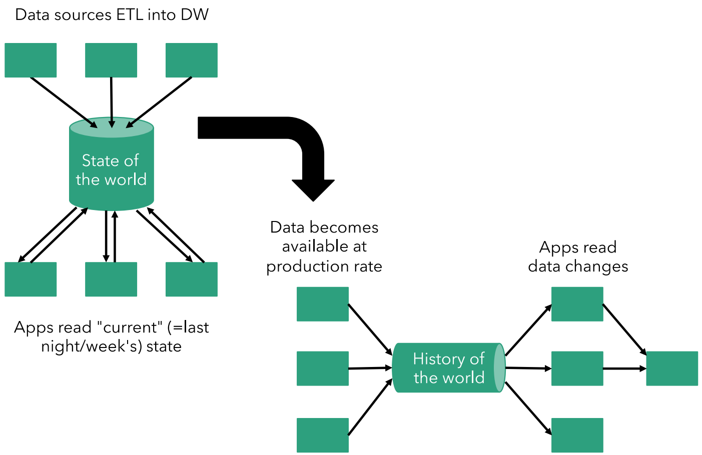

```
while (true) {
  // get next few records
  // issue batch job
}
```

Why streaming



What does streaming enable?

1. Data integration



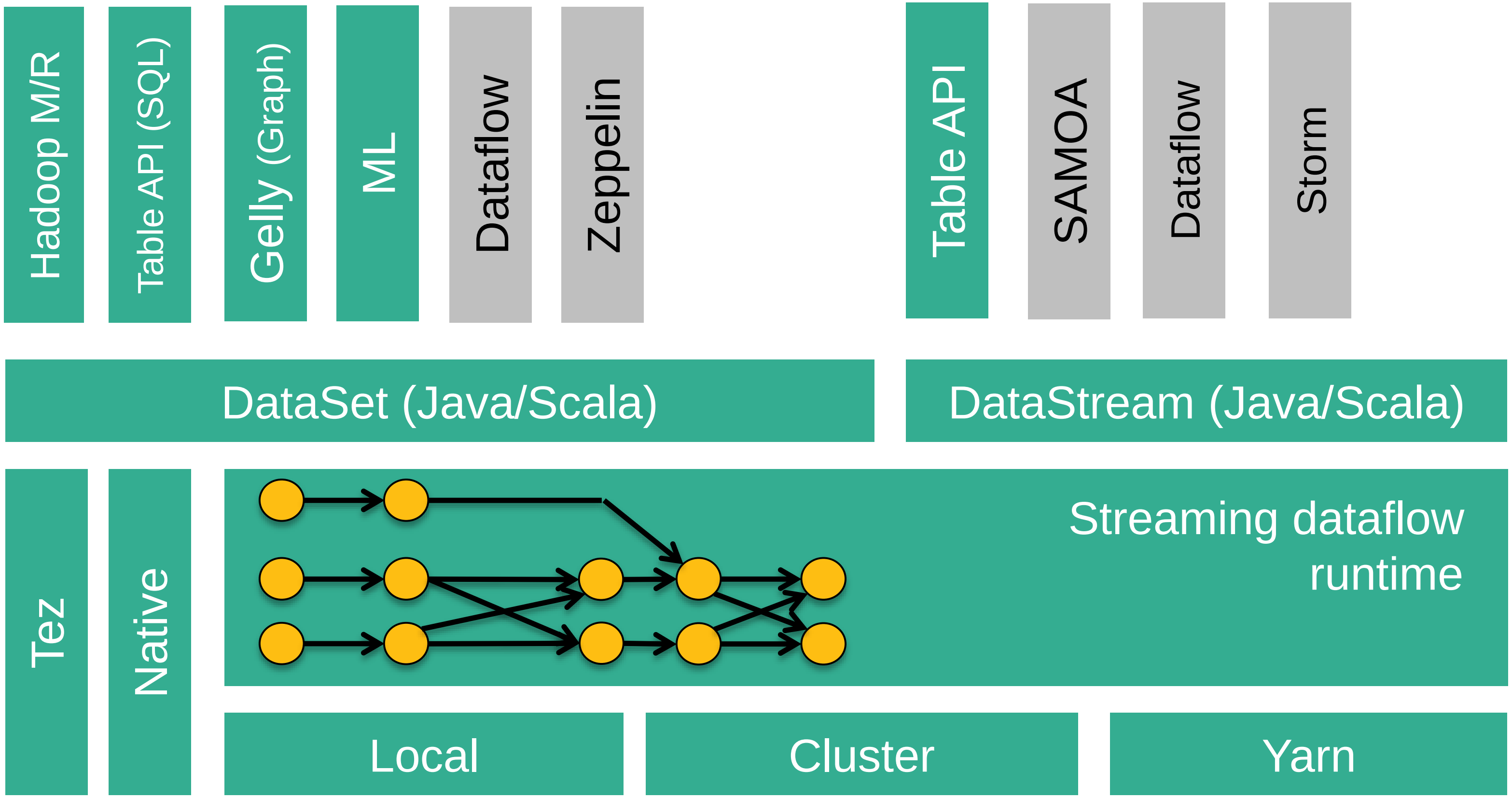
cf. Kleppmann: "Turning the DB inside out with Samza"

2. Low latency applications

- Fresh recommendations, fraud detection, etc
- Internet of Things, intelligent manufacturing
- Results “right here, right now”

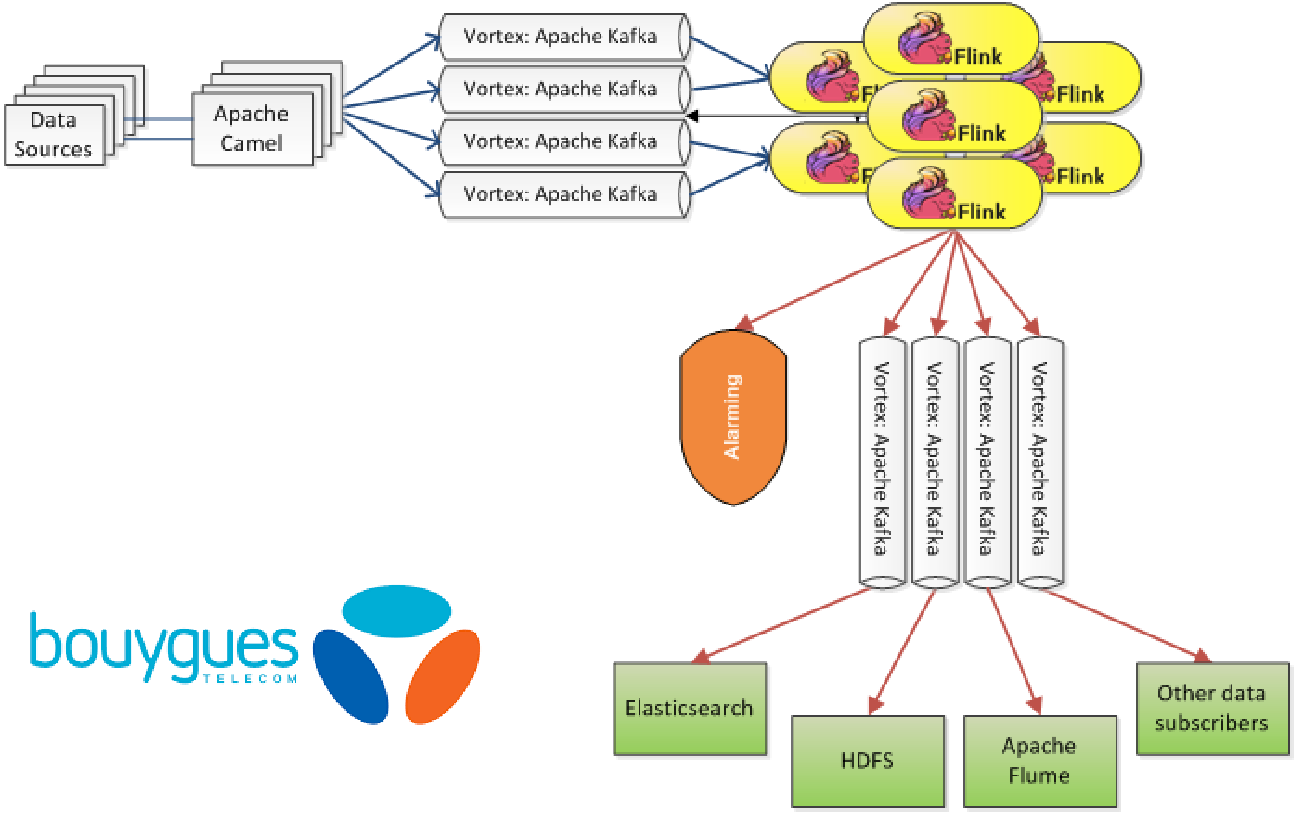
3. Batch < Streaming

The Stack

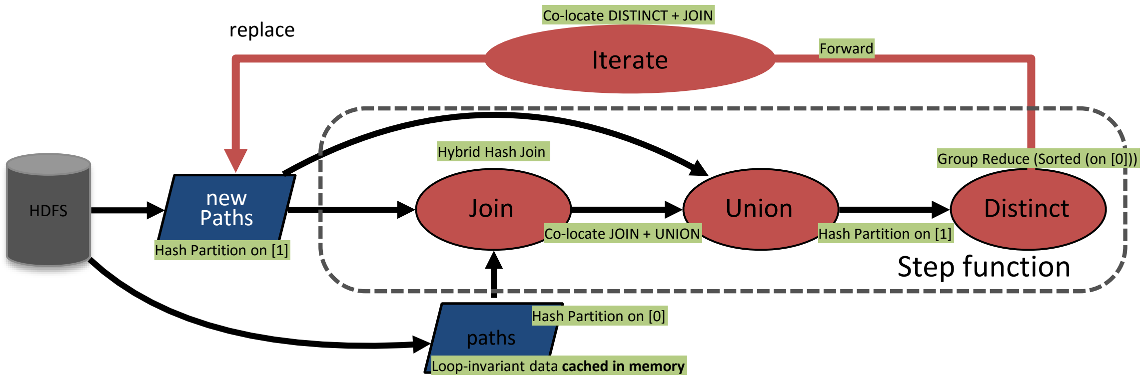


Hadoop M/R
Table API (SQL)
Gelly (Graph)
ML
Dataflow
Zeppelin
Table API
SAMOA
Dataflow
Storm
DataSet (Java/Scala)
DataStream (Java/Scala)
Tez
Native
Streaming dataflow runtime
Local
Cluster
Yarn

Example: Bouygues Telecom



Flink Optimizer



- What you write is ***not*** what is executed
- No need to hardcode execution strategies

- Flink Optimizer decides:
  - Pipelines and dam/barrier placement
  - Sort- vs. hash- based execution
  - Data exchange (partition vs. broadcast)
  - Data partitioning steps
  - In-memory caching

What is a stream processor?

1. Pipelining
2. Stream replay

Basics

3. Operator state
4. Backup and restore

State

5. High-level APIs
6. Integration with batch

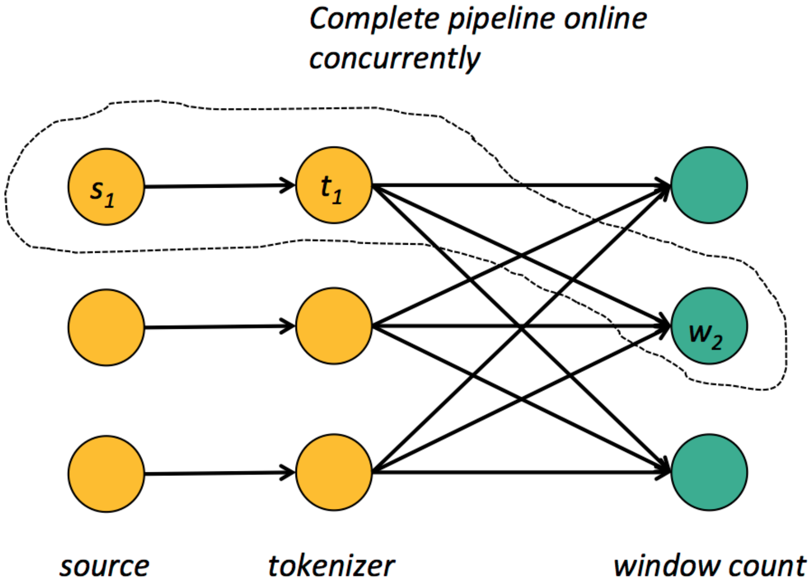
App development

7. High availability
8. Scale-in and scale-out

Large deployments

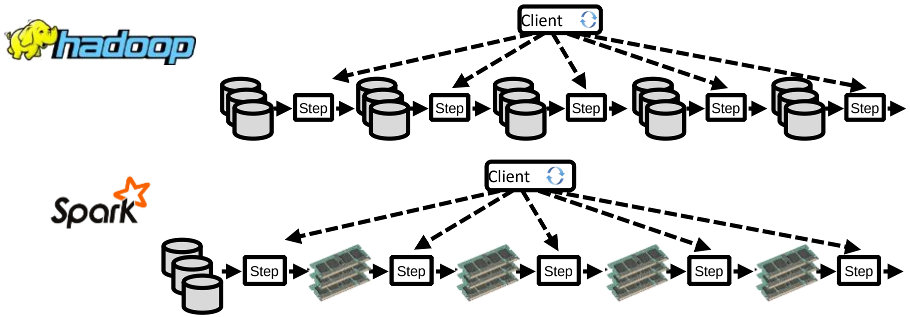
Pipelining

Basic building block to "keep the data moving"



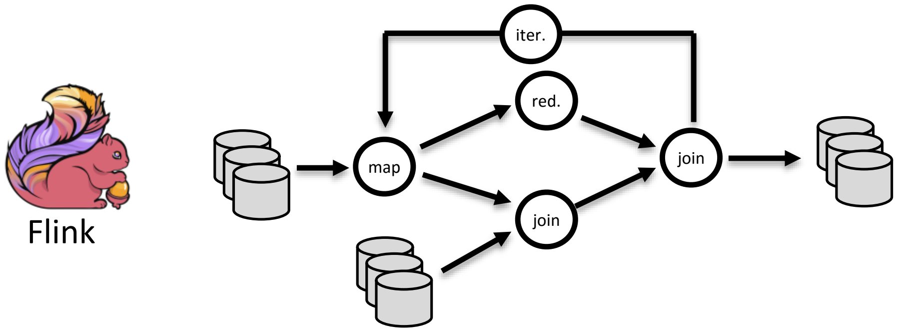
Note: pipelined systems do not usually transfer individual tuples, but buffers that batch several tuples!

Built-in vs. driver-based looping



Loop outside the system, in driver program

Iterative program looks like many independent jobs

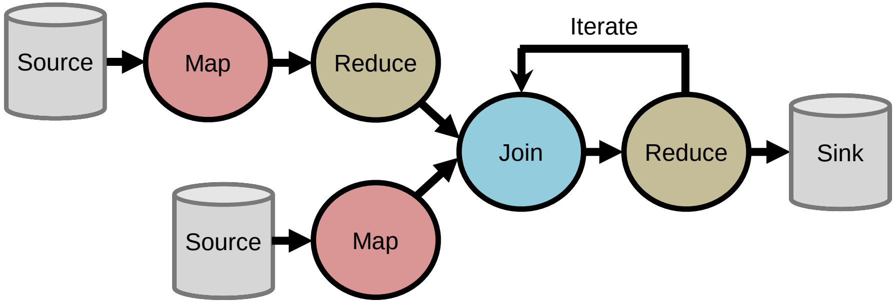


Dataflows with feedback edges

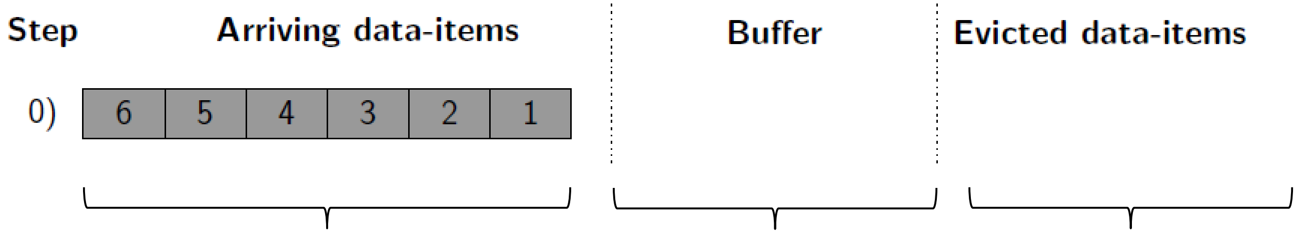
System is iteration-aware, can optimize the job

Rich set of operators

Map, Reduce, Join, CoGroup, Union, Iterate, Delta Iterate, Filter, FlatMap, GroupReduce, Project, Aggregate, Distinct, Vertex-Update, Accumulators, …



Stream Discretization Policies

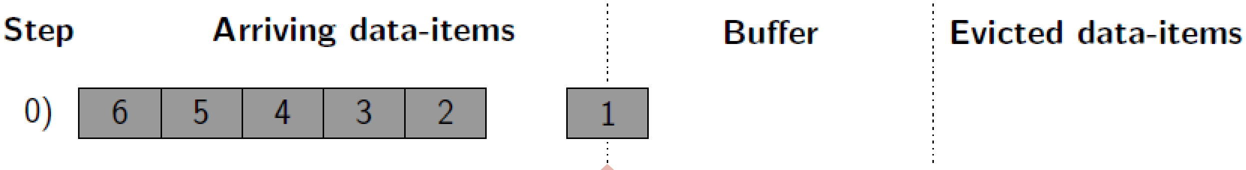


Streams are represented as FIFO-Queue of data-items

The window operator keeps a FIFO-Buffer

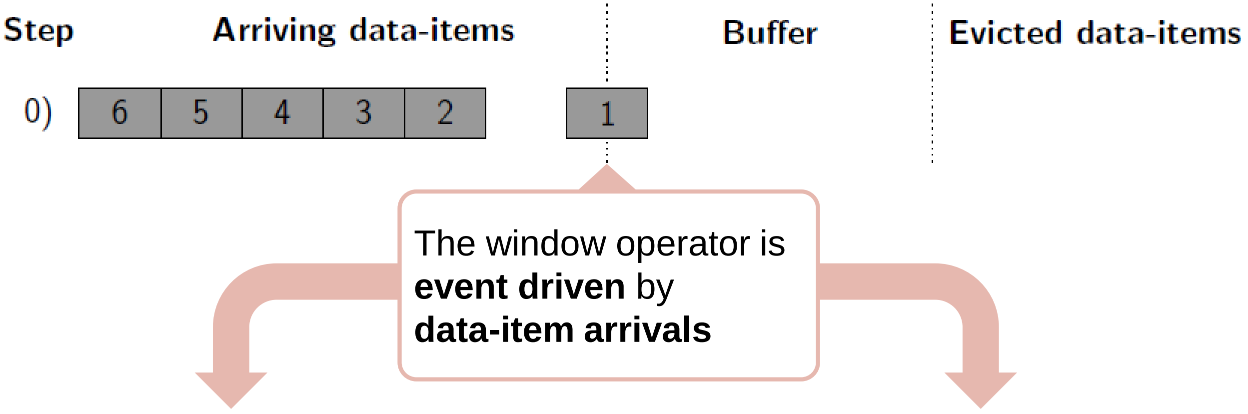
After some time, data-items expire (they are deleted)

Stream Discretization Policies



The window operator is **event driven** by **data-item arrival**

Stream Discretization Policies



1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

Define the moment that results are emitted.

2.) Eviction Policies (EPs)
Specify when data-items are removed from the buffer.

Defines the size of a window.

Stream Discretization Policies

Step	Arriving data-items	Buffer	Evicted data-items
1)	6 5 4 3 2	1	

Query Example (tumbling/fixed window of size 3):

```
dataStream.window(Count.of(3))
```

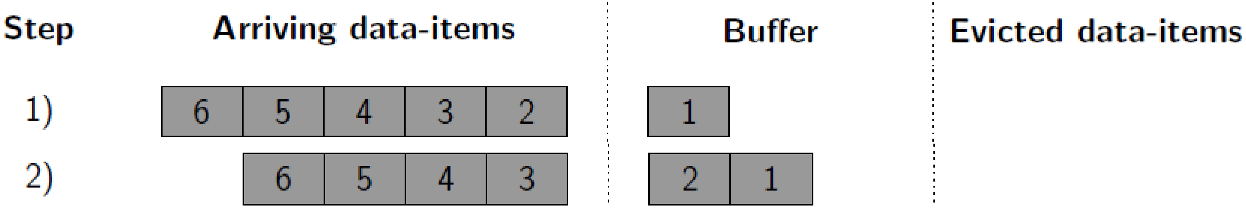
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

Define the moment that results are emitted.

2.) Eviction Policies (EPs)
Specify when data-items are removed from the buffer.

Defines the size of a window.

Stream Discretization Policies



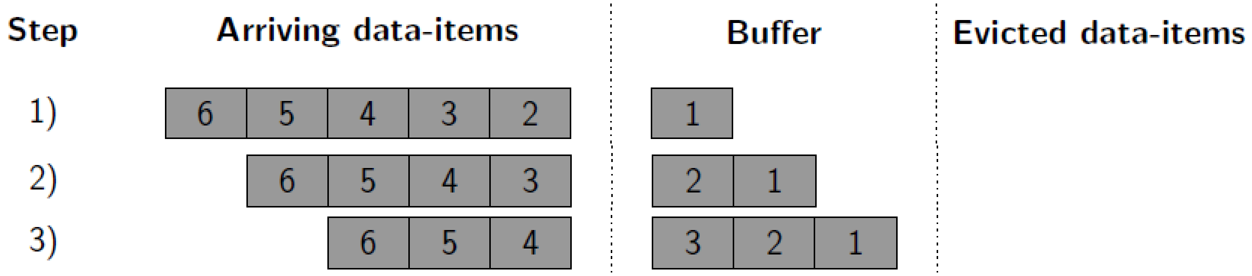
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

Define the moment that results are emitted.

2.) Eviction Policies (EPs)
Specify when data-items are removed from the buffer.

Defines the size of a window.

Stream Discretization Policies



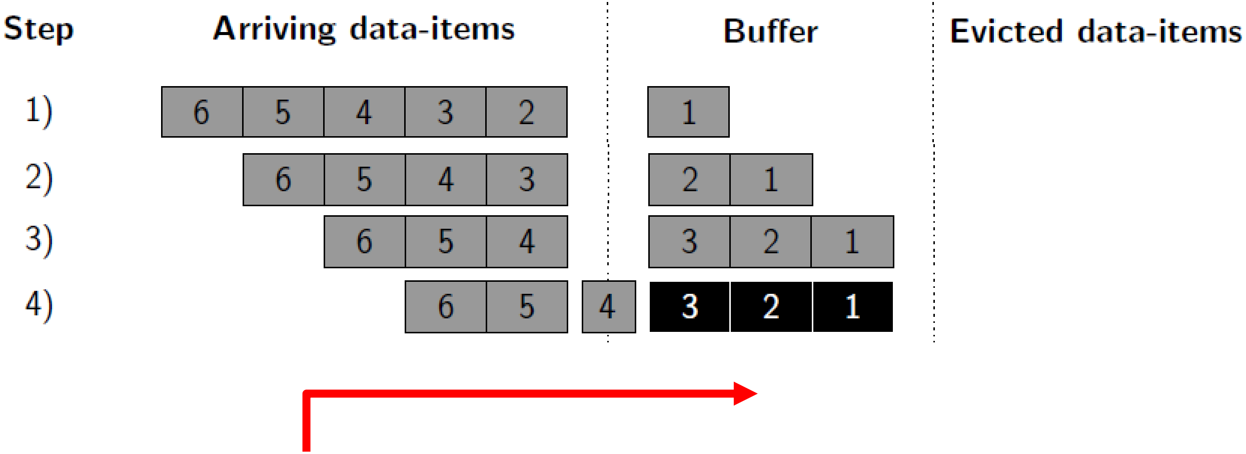
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

Define the moment that results are emitted.

2.) Eviction Policies (EPs)
Specify when data-items are removed from the buffer.

Defines the size of a window.

Stream Discretization Policies



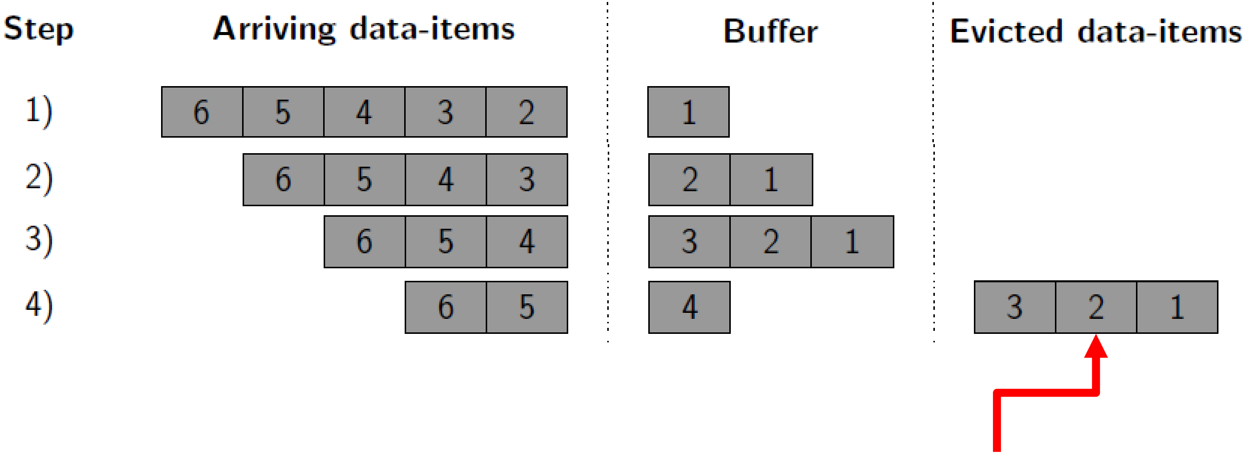
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

Define the moment that results are emitted.

2.) Eviction Policies (EPs)
Specify when data-items are removed from the buffer.

Defines the size of a window.

Stream Discretization Policies



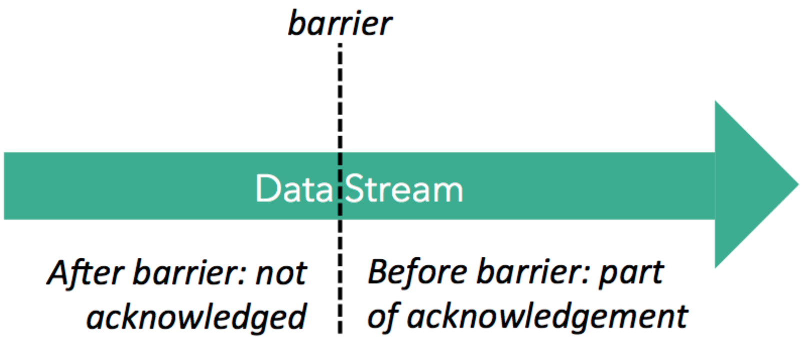
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

Define the moment that results are emitted.

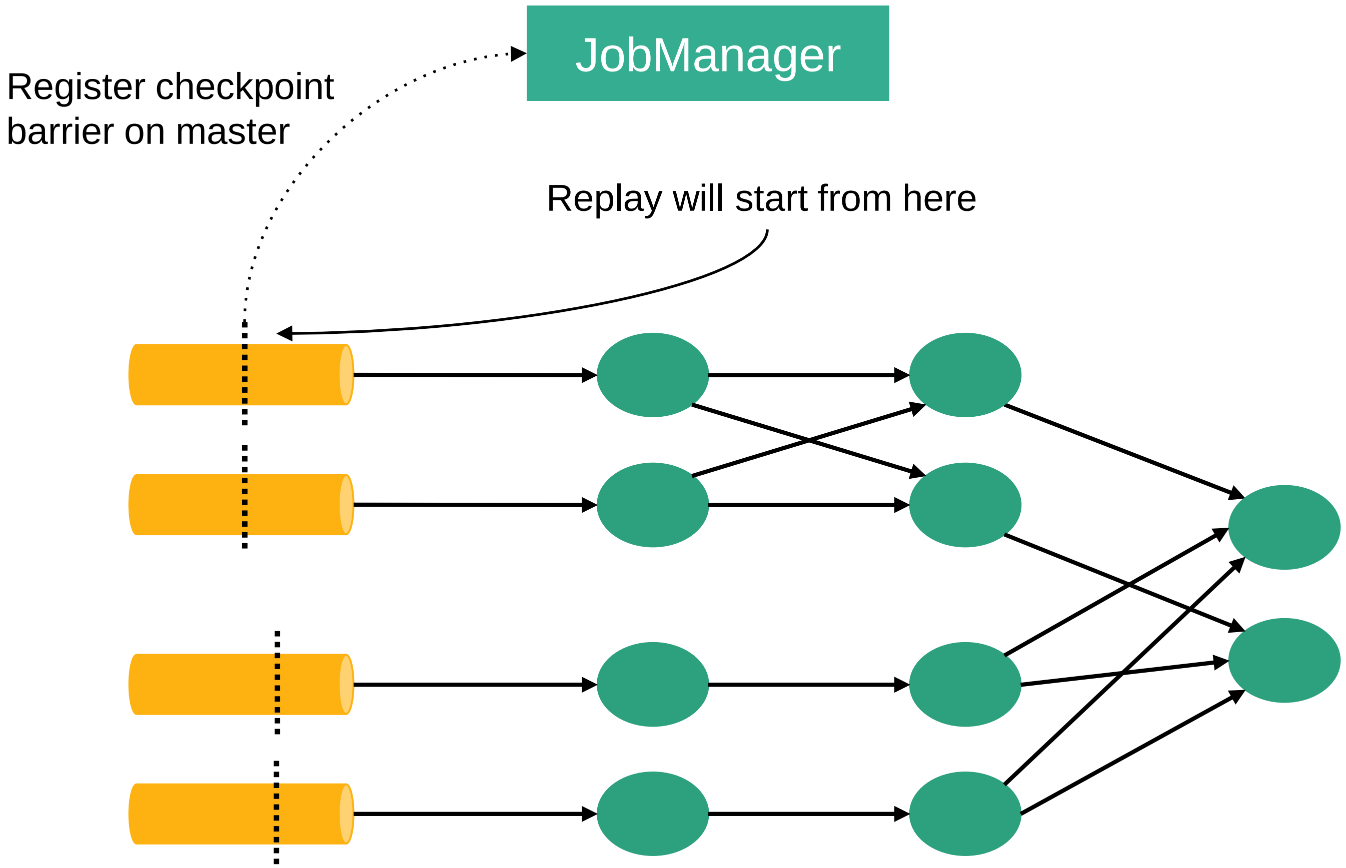
2.) Eviction Policies (EPs)
Specify when data-items are removed from the buffer.

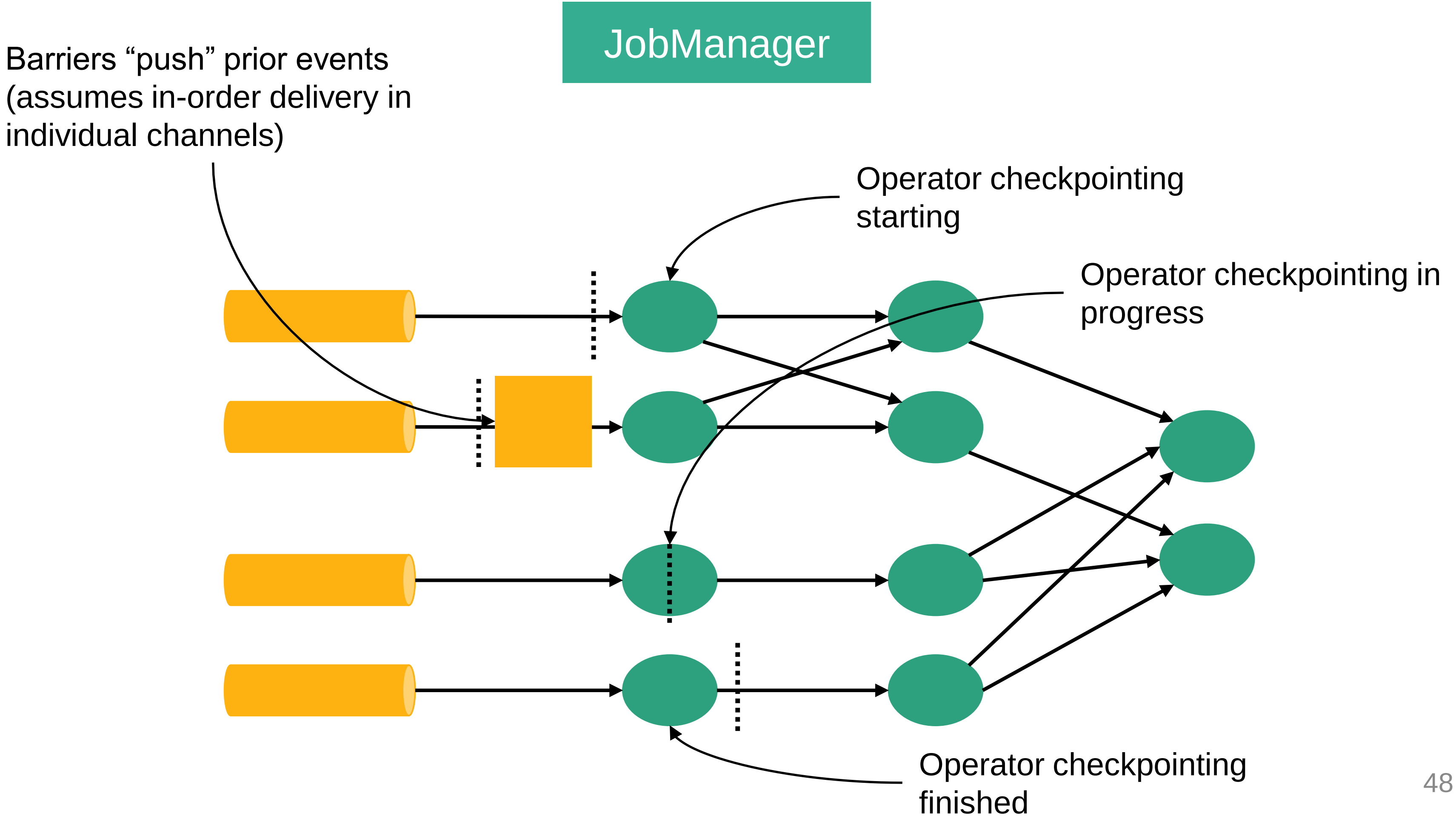
Defines the size of a window.

Distributed snapshots in Flink



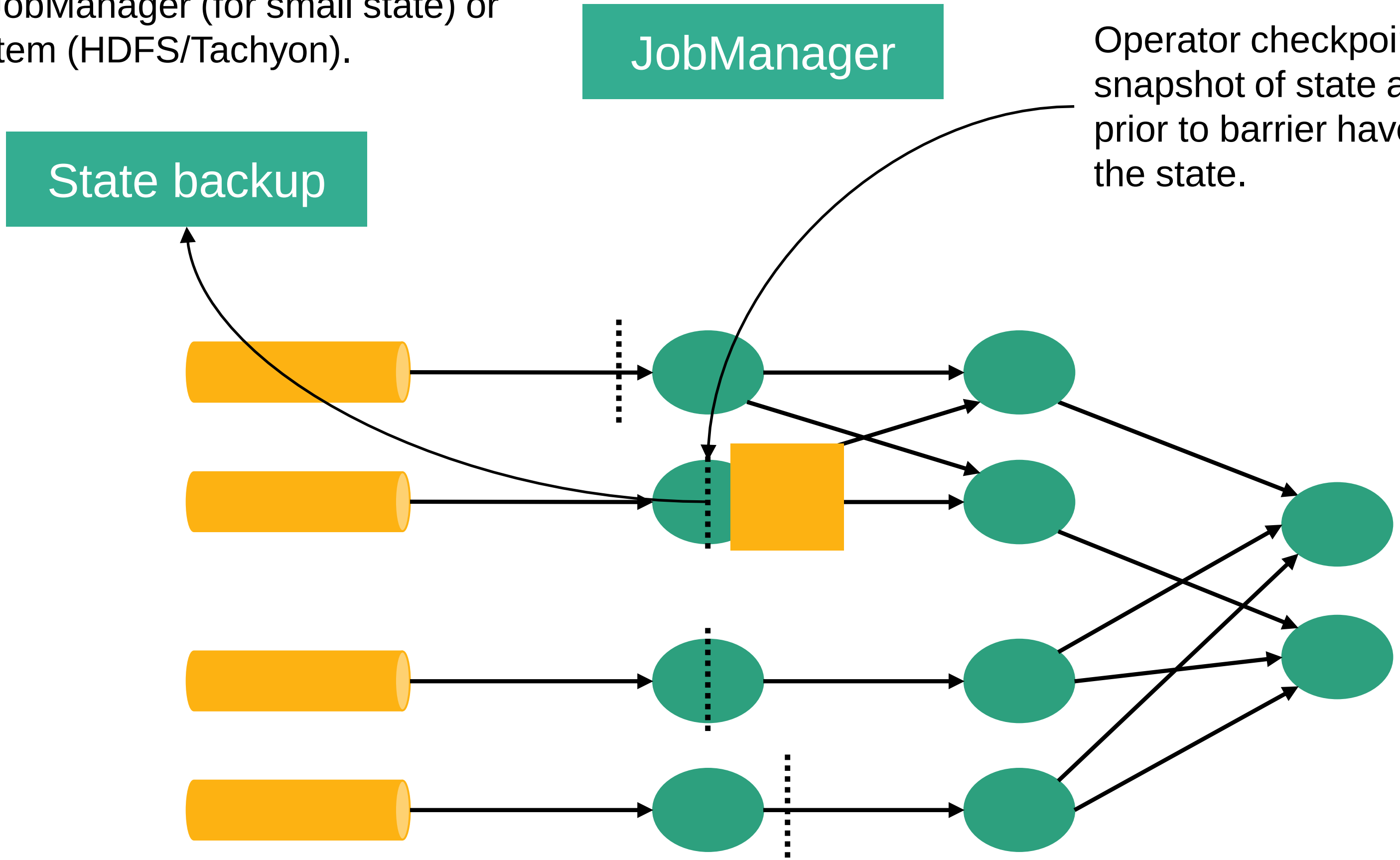
Super-impose checkpointing mechanism on execution instead of using execution as the checkpointing mechanism


JobManager
Register checkpoint
barrier on master
Replay will start from here

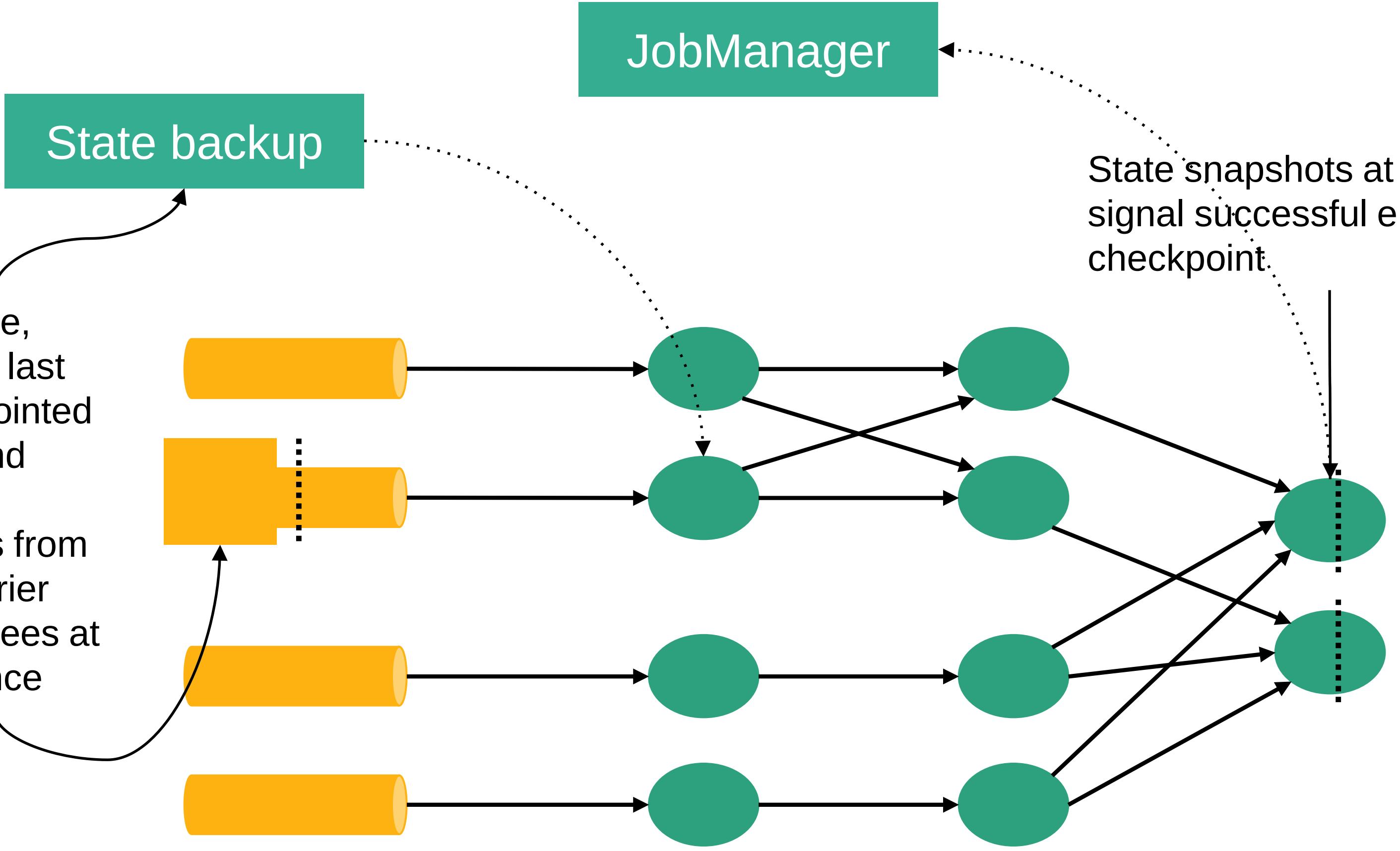

JobManager
Barriers “push” prior events (assumes in-order delivery in individual channels)
Operator checkpointing starting
Operator checkpointing in progress
Operator checkpointing finished

Pluggable mechanism. Currently either JobManager (for small state) or file system (HDFS/Tachyon).

Operator checkpointing takes snapshot of state after data prior to barrier have updated the state.

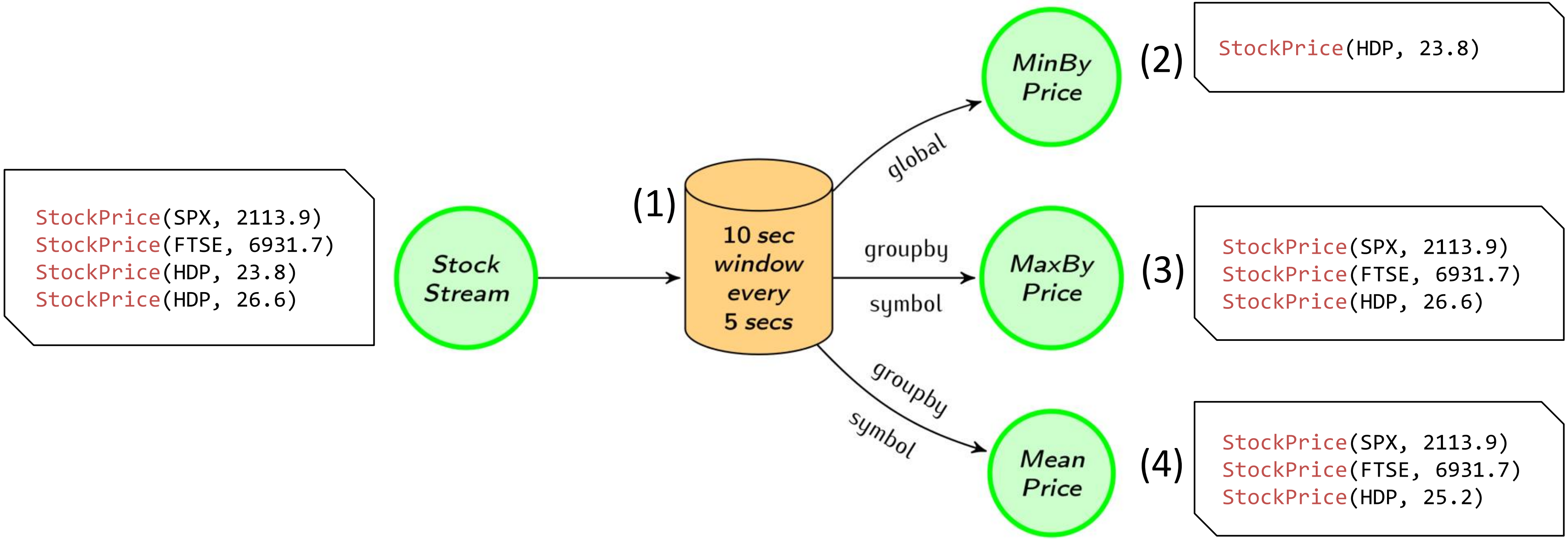


At failure, recover last checkpointed state and restart sources from last barrier guarantees at least once



State snapshots at sinks signal successful end of this checkpoint

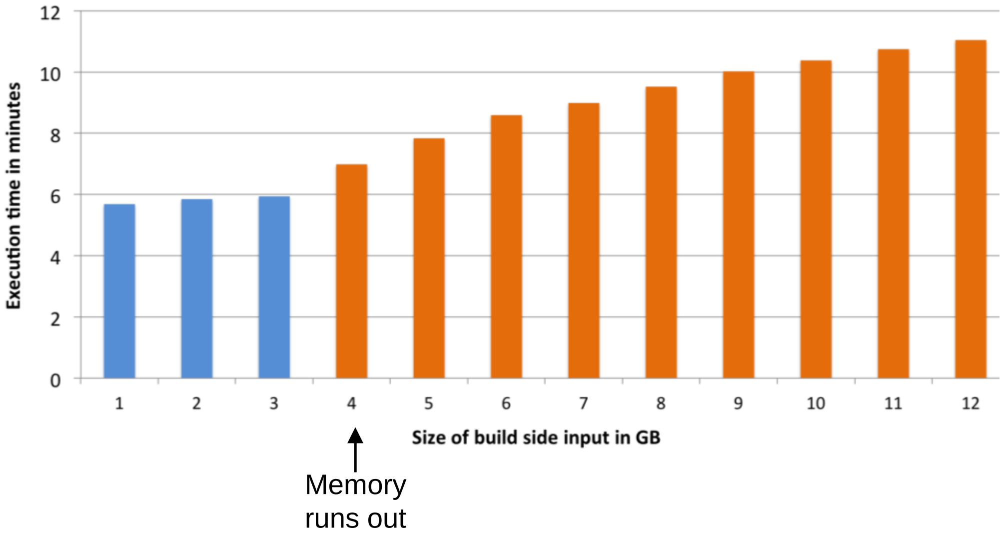
Example Analysis: Windowed Aggregation




```
(1) val windowedStream = stockStream.window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))
(2) val lowest = windowedStream.minBy("price")
(3) val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
(4) val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)
```

Managed Memory

- Language APIs automatically converts objects to tuples
  - Tuples mapped to pages of bytes
  - Operators work on pages
- Full control over memory, out-of-core enabled
- Operators (e.g., Hybrid Hash Join) address individual fields (not deserialize whole object)



Desiderate for Stream Processors

1. Pipelining
2. Stream replay

Basics

3. Operator state
4. Backup and restore

State

5. High-level APIs
6. Integration with batch

App development

7. High availability
8. Scale-in and scale-out

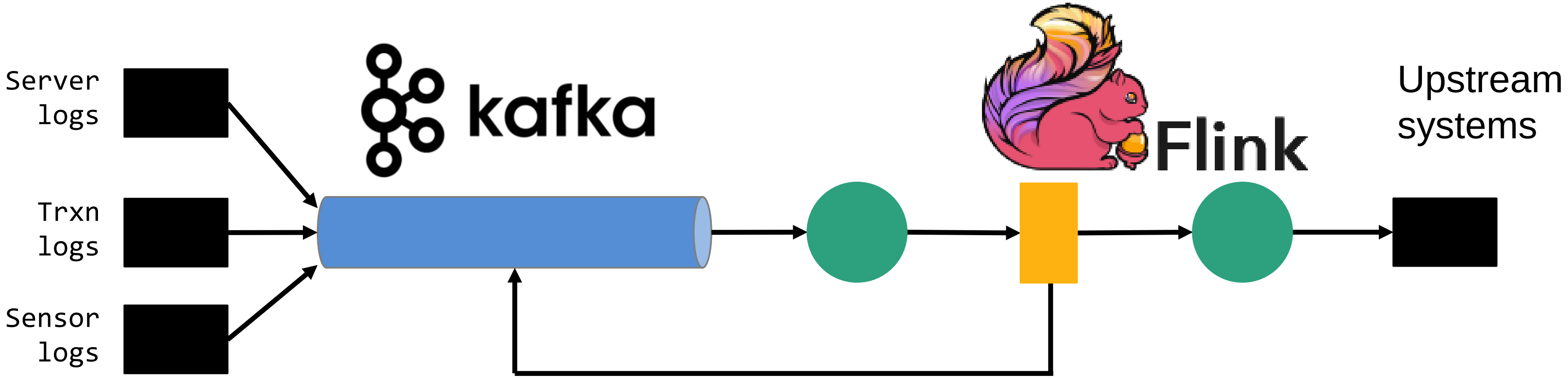
Large deployments

Benefits of Flink's approach

- Data processing does not block
  - Can checkpoint at any interval you like to balance overhead/recovery time
- Separates business logic from recovery
  - Checkpointing interval is a config parameter, not a variable in the program (as in discretization)
- Can support richer windows
  - Session windows, event time, etc.
- Best of all worlds: true streaming latency, exactly-once semantics, and low overhead for recovery

Where in my cluster does Flink fit?

Gathering | Integration | Analysis



- Gather and backup streams
- Offer streams for consumption
- Provide stream recovery

- Analyze and correlate streams
- Create derived streams and state
- Provide these to upstream systems