Stream & Batch processing with Apache Flink™

Asterios Katsifodimos

TU Berlin

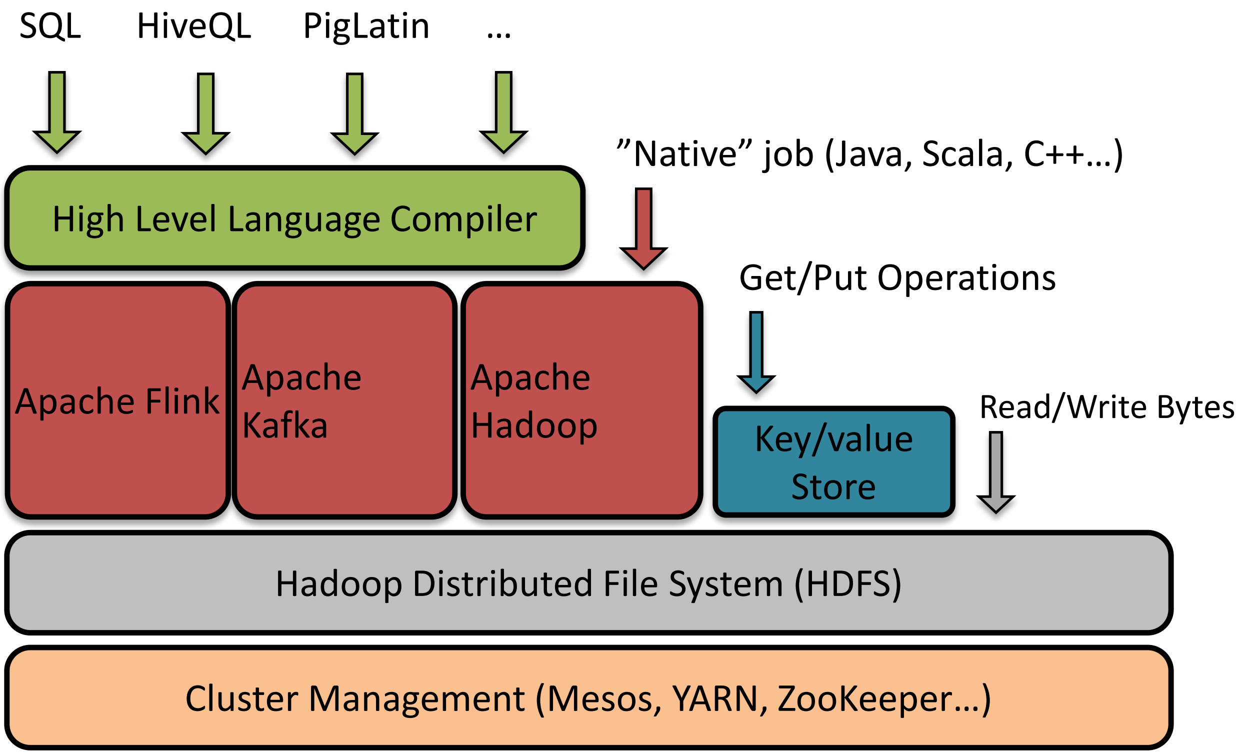
asterios.katsifodimos@tu-berlin.de

Overview of the Tutorial

- Introduction
- API's Overview

- While we are giving the talks:
  - Grab one USB Stick
  - Install Java and IntelliJ
  - Start IntelliJ
  - Extract and import the project (code/Template.zip)

A (narrow) view of the Big Data Zoo



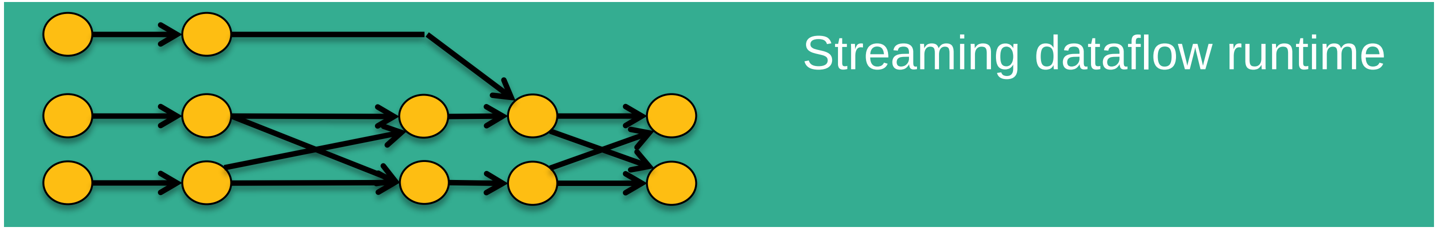
In this talk

- **Apache Flink Primer**
  - Architecture
  - Execution Engine
  - API Examples
- **Stream Processing with Apache Flink**
  - Flexible Windows/Stream Discretization
  - Exactly-once Processing & Fault Tolerance
- **The Road Ahead**
  - The Emma language

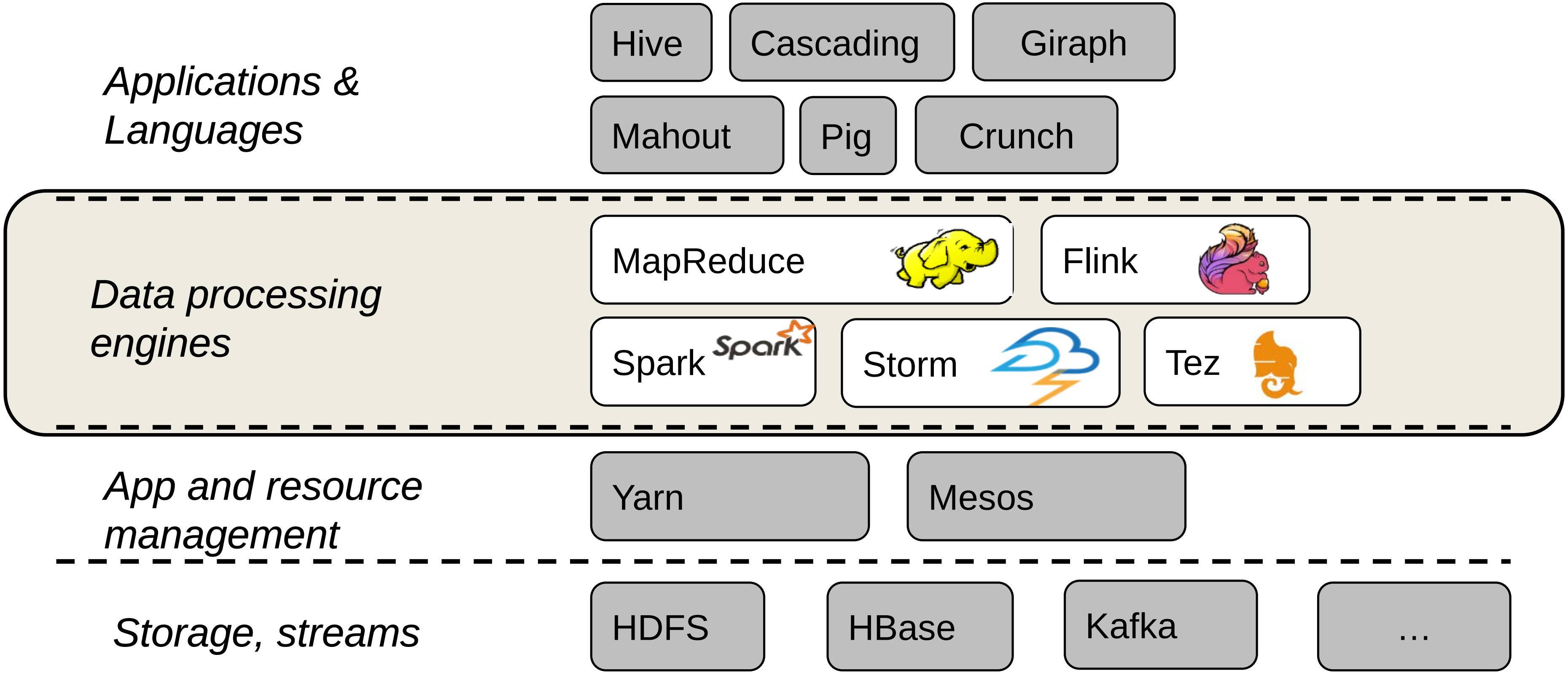
Apache Flink Primer

What is Flink?

A platform for distributed
batch and streaming analytics

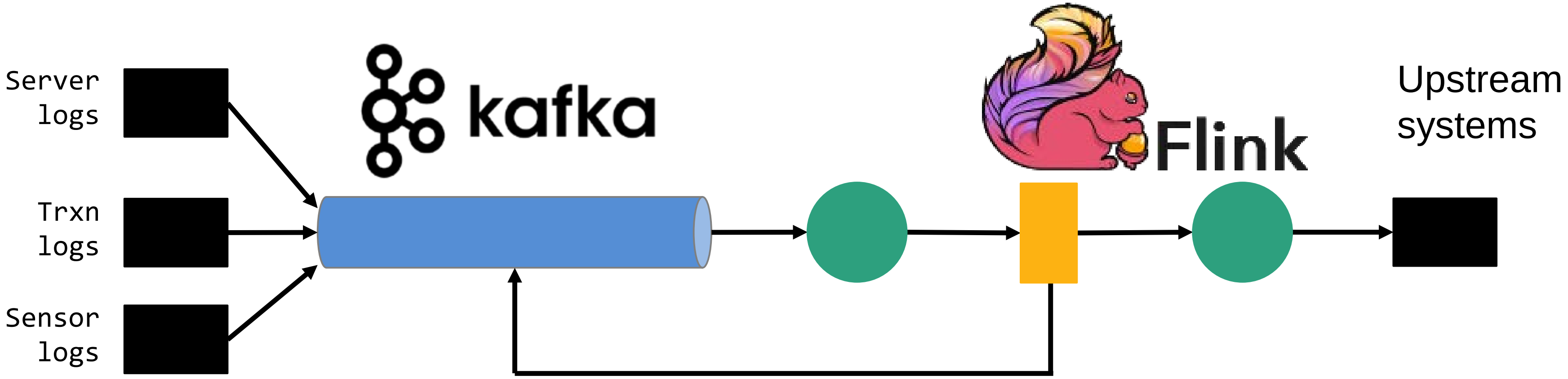


Flink in the Analytics Ecosystem



Where in my cluster does Flink fit?

Gathering | Integration | Analysis

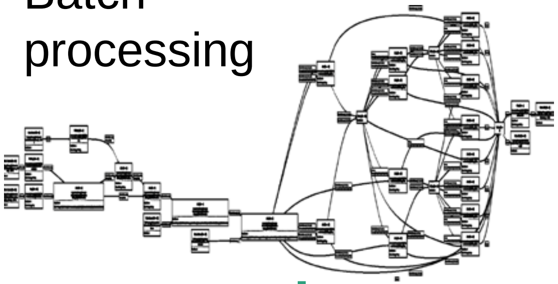


- Gather and backup streams
- Offer streams for consumption
- Provide stream recovery

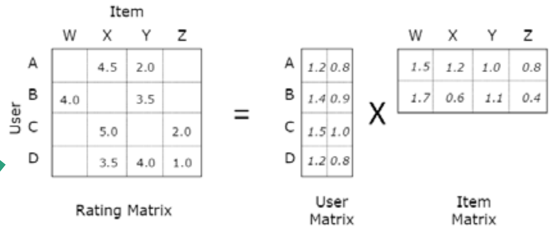
- Analyze and correlate streams
- Create derived streams and state
- Provide these to upstream systems

What can I do with it?

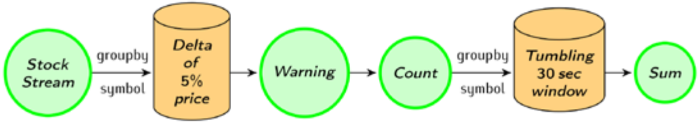
Batch processing



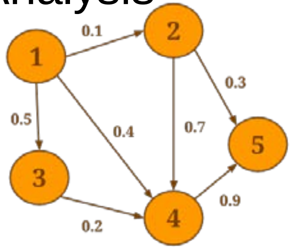
Machine Learning at scale



Stream processing



Graph Analysis

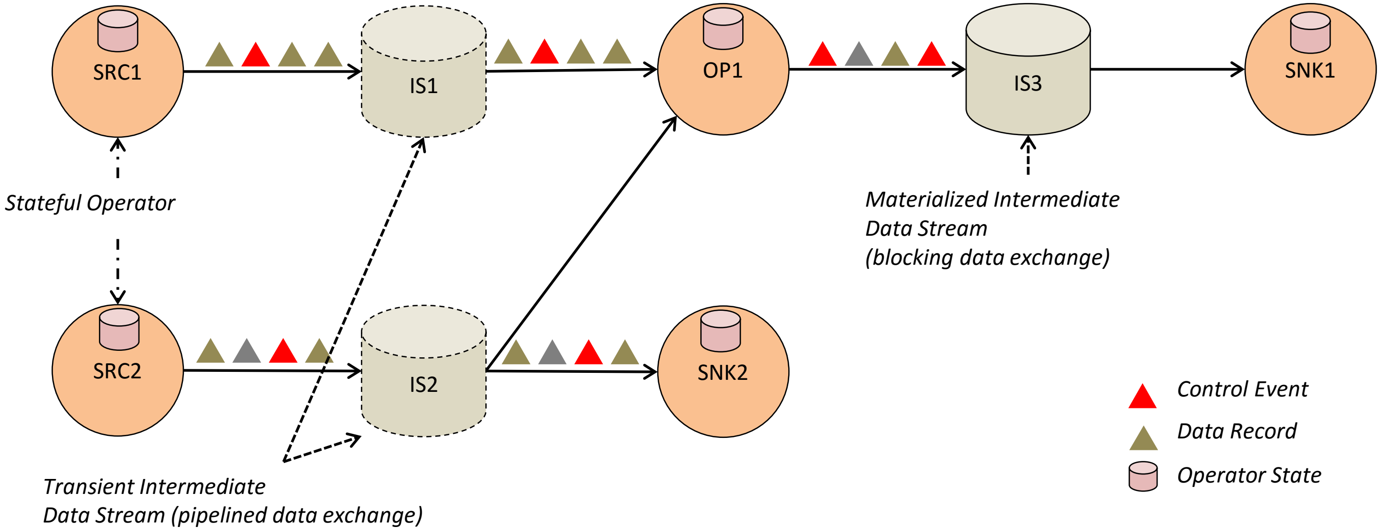


Flink

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

Execution Model

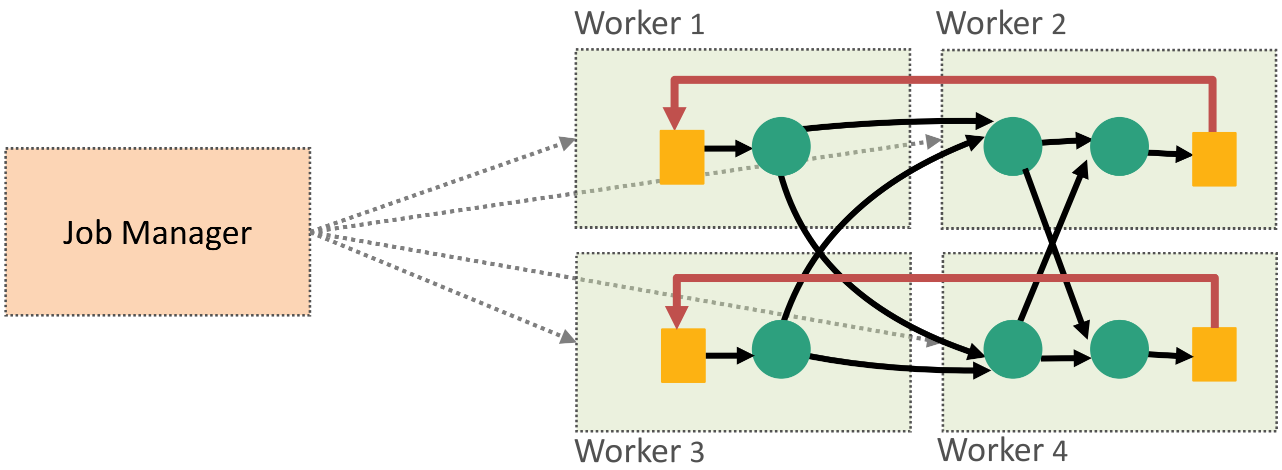
- Flink program = DAG* of operators and intermediate streams
- Operator = computation + state
- Intermediate streams = logical stream of records



Architecture

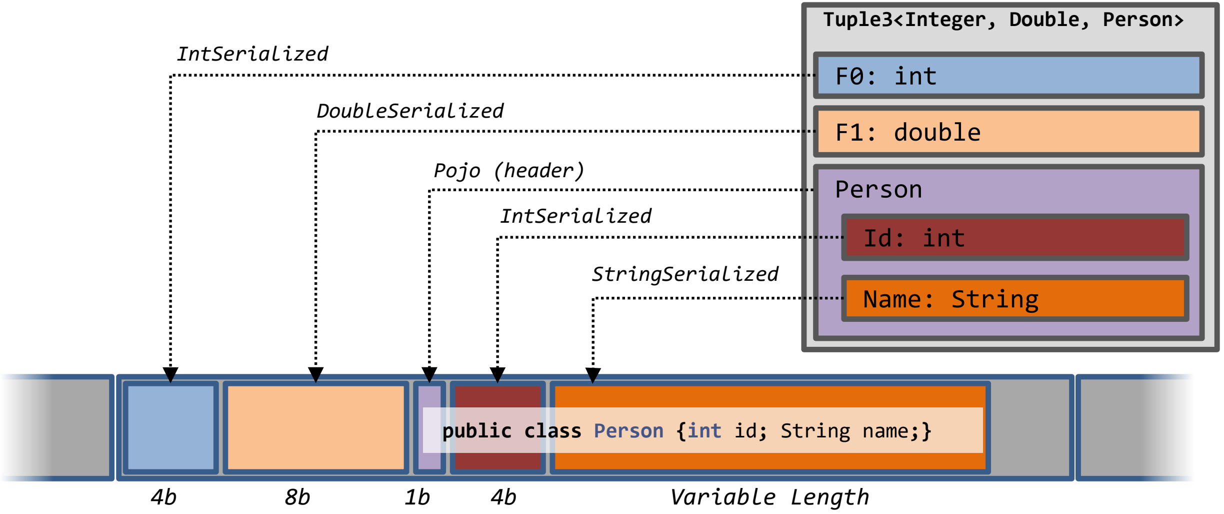
- Hybrid MapReduce and MPP database runtime

- Pipelined/Streaming engine
  - Complete DAG deployed



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



Stream Processing with Flink

Ingredients of a Streaming System

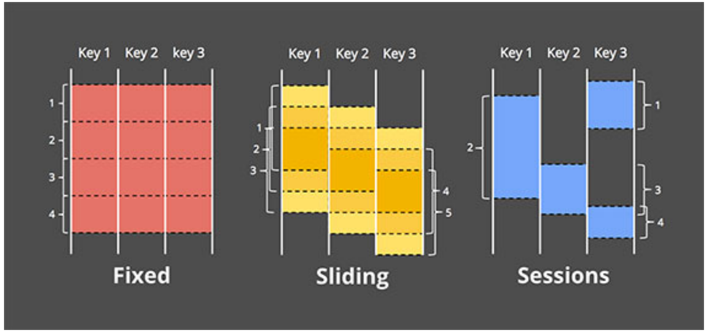
- Streaming Execution Engine
- Windowing (a.k.a Discretization)
- Fault Tolerance
- High Level Programming API (or language)

Ingredients of a Streaming System

- Streaming Execution Engine
- **Windowing (a.k.a Discertization)**
- ~~**Fault Tolerance**~~
- High Level Programming API (or language)

Stream Discretization

- Data is unbounded
  - Interested in a (recent) part of it e.g. last 10 days
- Most common windows around: time, and count
  - Mostly in sliding, fixed, and tumbling forms
- Need for data-driven window definitions
  - e.g., user sessions (periods of user activity followed by inactivity), price changes, etc.



The world beyond batch: Streaming 101, Tyler Akidau
https://beta.oreilly.com/ideas/the-world-beyond-batch-streaming-101
Great read!

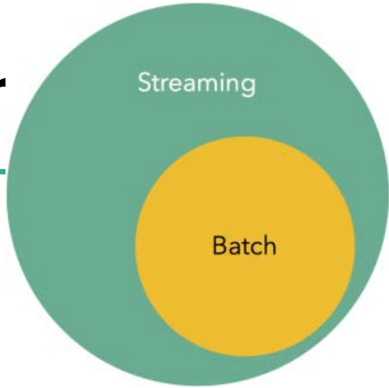
Flink's Windowing

- Windows can be any combination of (multiple) triggers & evictions
  - Arbitrary tumbling, sliding, session, etc. windows can be constructed.
- Common triggers/evictions part of the API
  - Time (processing vs. event time), Count
- Even more flexibility: *define your own UDF* trigger/eviction
- Examples:

```
dataStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(5)));
dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5)));
```

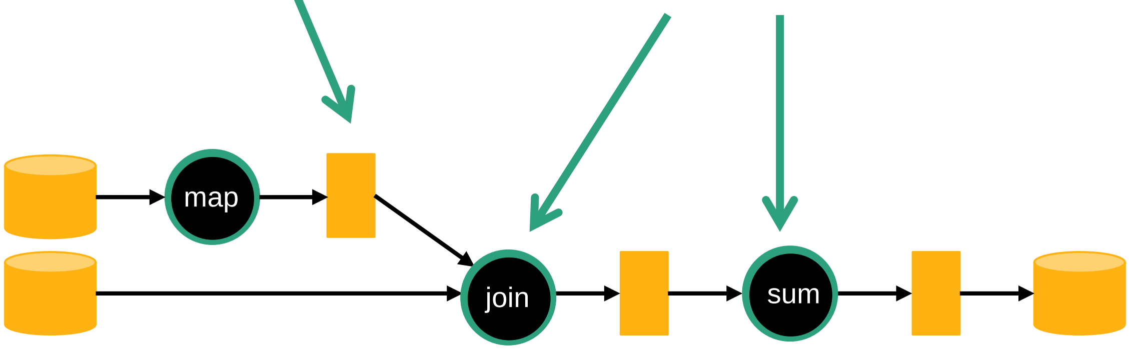
Batch vs. Streaming Analytics

Batch is a Special Case of Streamin

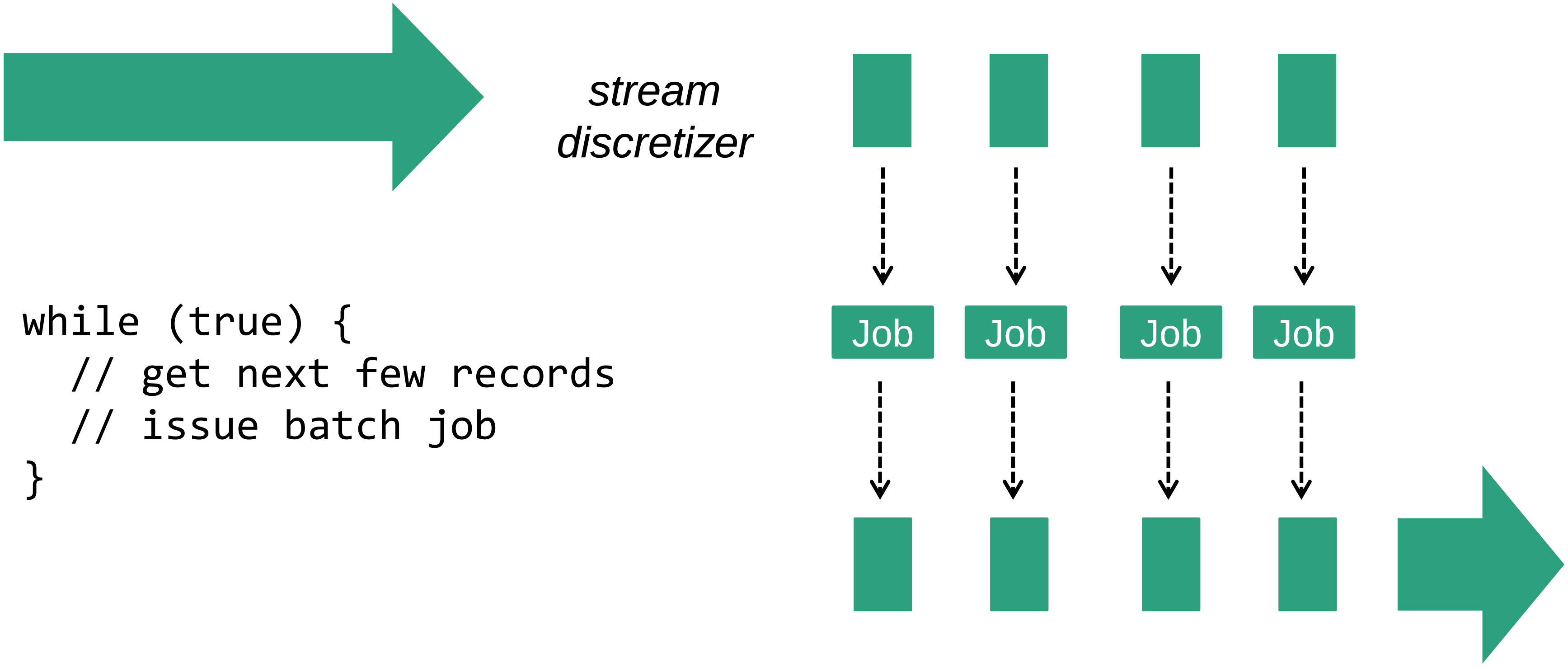


Lower-overhead fault-tolerance via replaying intermediate results

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



e.g.: Non-native streaming
(Spark, Hadoop, etc.)




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

Closing

tl;dr: what was this about?

- The case for Flink as a stream processor
  - Proper streaming engine foundation
  - Flexible Windowing
  - Fault Tolerance with exactly once guarantees
  - Integration with batch
  - Large (and growing!) community

FlinkForward

Berlin | September 12-14, 2016

Save the Date!

Flink Forward 2016 will take place September 12 - 14 at Kulturbrauerei Berlin.

We are currently working on our new website.
In the meantime, make sure to join our mailing list and **get notified** about Call for Papers and Ticket Sales.

Take a look at last year's conference here!

Photo: Palais © Palais Veranstaltungs GmbH

Thank you

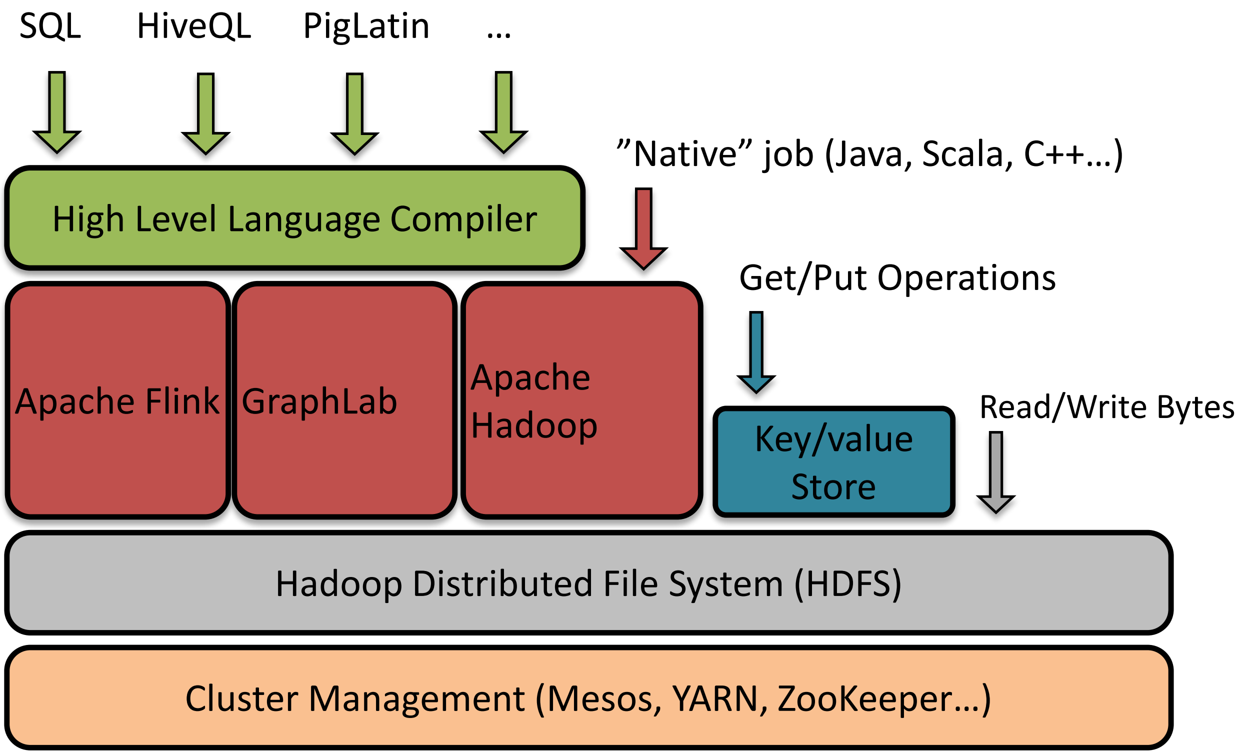
The road ahead

Parallel Data Analysis for non-geeks

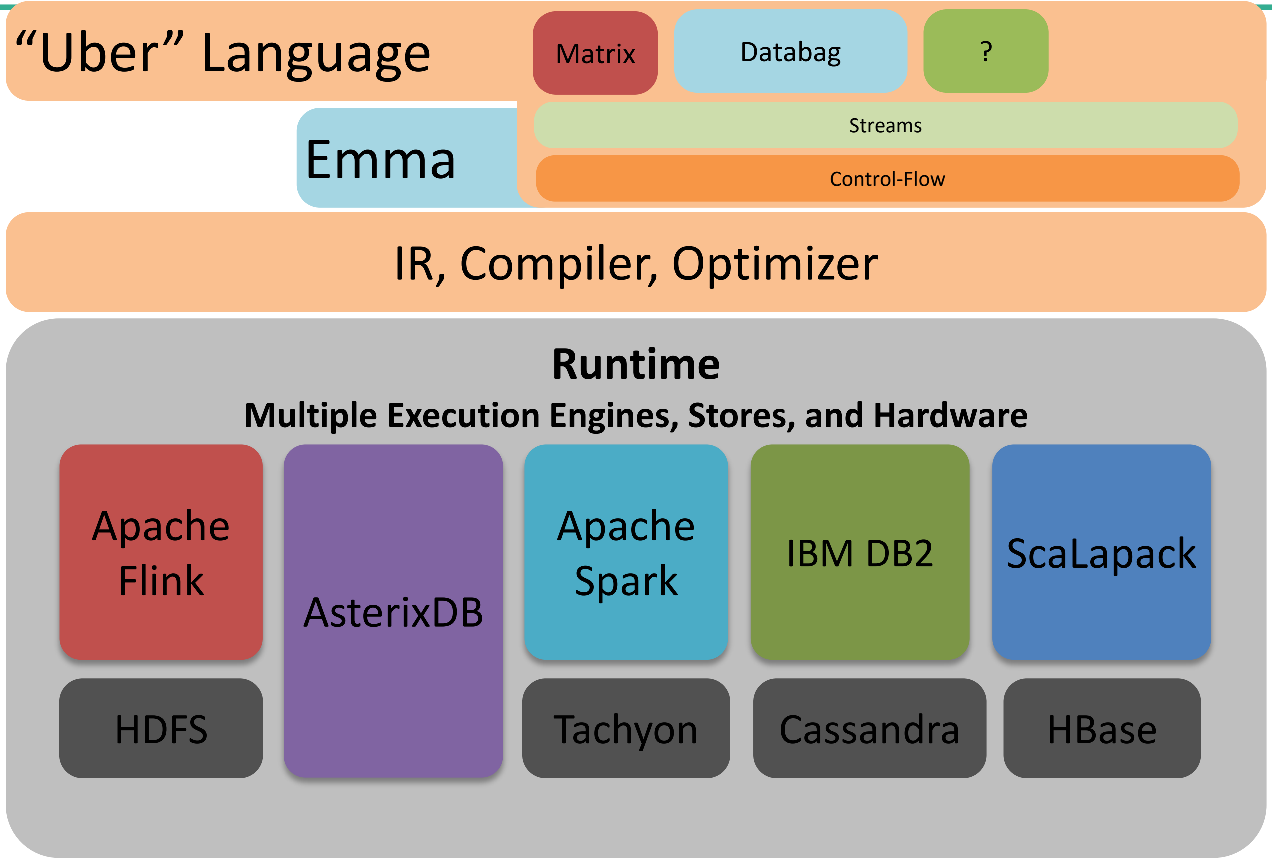
Querying Databases ➤ Data Analysis

- Tables ➤ Tables and unstructured files, matrices, logs, graphs
  - Multitude of data models
- Queries (SQL) ➤ Programs (Java, SQL, R, Scala, Python, R, etc.)
  - Iterative processing, control flow, general object manipulation, user defined functions
- Data Loading ➤ Files dumped in a (H)DFS
  - Schema on read
- Proprietary ➤ Open source
  - Multitude of Systems

A (narrow) view of the Big Data Zoo



Database Mosaics



Emma: Key Features

```
... // initialize points and clusters
while (change > epsilon) {
  val clusters = (for (p <- points) yield {
    val c = ctrds.minBy(distanceTo(p)).get
    Solution(c.id, p.p)
  }).groupBy(_.cid)
  // compute new centroids
  val newCtrds = for (clr <- clusters) yield {
    val sum = (for (p <- clr.values) yield p.pos).sum()
    val cnt = (for (p <- clr.values) yield p.pos).cnt()
    Point(c.key, sum / cnt)
  }
  // compute the total change in all centroids
  change = {
    val distances = for (
      x <- ctrds;
      y <- newCtrds; if x.id == y.id) yield dist(x, y)
    distances.sum()
  }
  // use the new centroids in the next iteration
  ctrds = newCtrds
}
... // finalize result
```

Deeply Embedded in Scala

- Relax! This is not a new language

Core type: **DataBag**

- a.k.a. **RDD** (Spark) / **DataSet** (Flink)
- Based on **union** algebra & **folds**

Scala **for**-comprehensions

- Instead of **join**, **cross**
- Like **Select-From-Where** in SQL

Nesting

- Group values of type **DataBag**
- Ubiquitous abstraction for computation

DataBag expressions as coarse-grained parallelism contracts

- Top-level: mapped to a dataflow API
- Everything else (mostly) untouched

Thank you

If you find this exciting,

get involved on Flink‘s mailing list

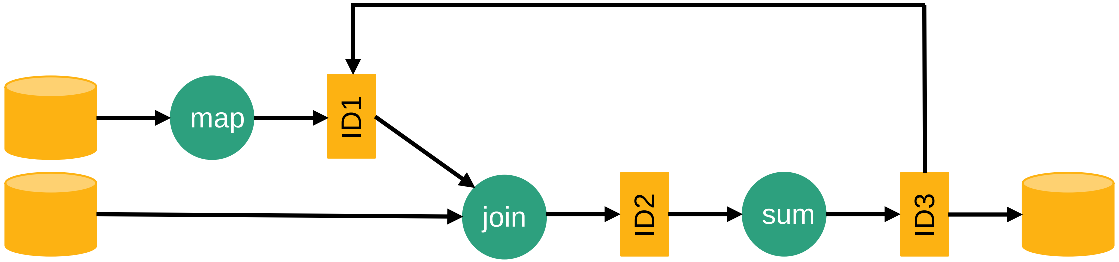
Subscribe to news@flink.apache.org,
follow flink.apache.org/blog, and
@ApacheFlink on Twitter

Want to try out Emma? Drop me an email:
asterios.katsifodimos@tu-berlin.de

Appendix

Iterative processing in Flink

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



Exactly once approaches

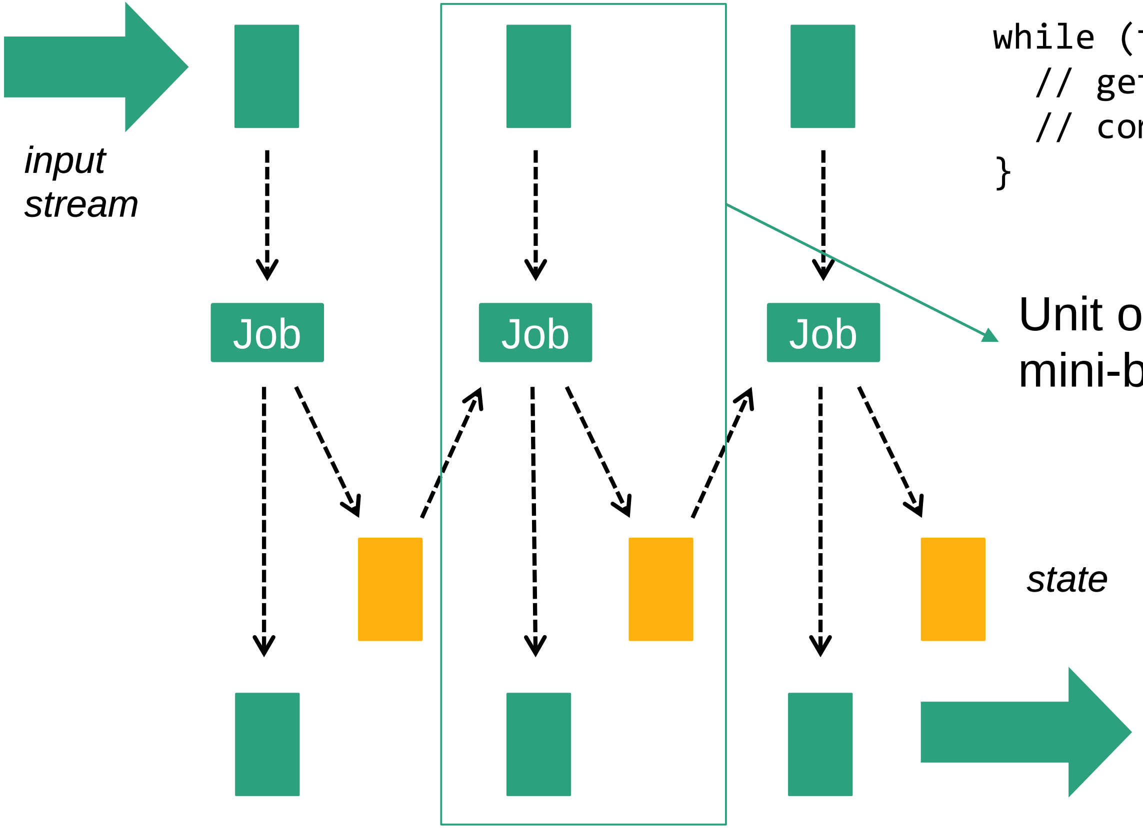
- Discretized streams – mini-batching (Spark Streaming)
  - Treat streaming as a series of small atomic computations
  - “Fast track” to fault tolerance, but does not separate business logic from recovery
- MillWheel (Google Cloud Dataflow)
  - State update and derived events committed as atomic transaction 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

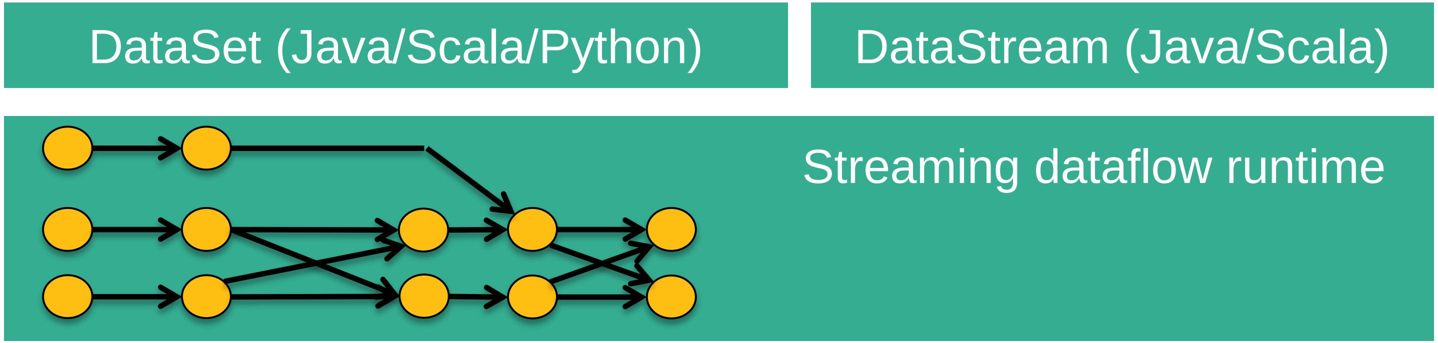
Exactly once approaches

- 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

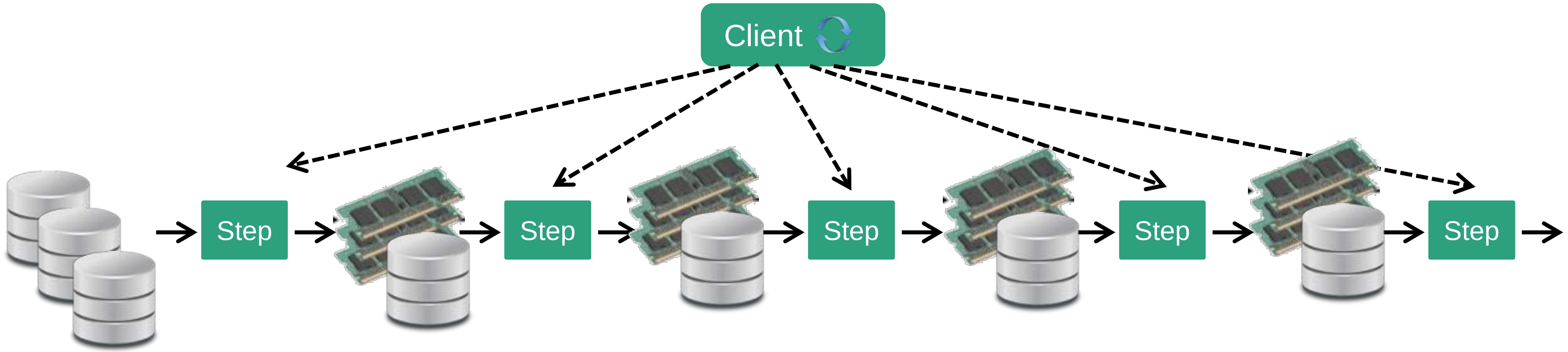
Integration with batch

- 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
}
```



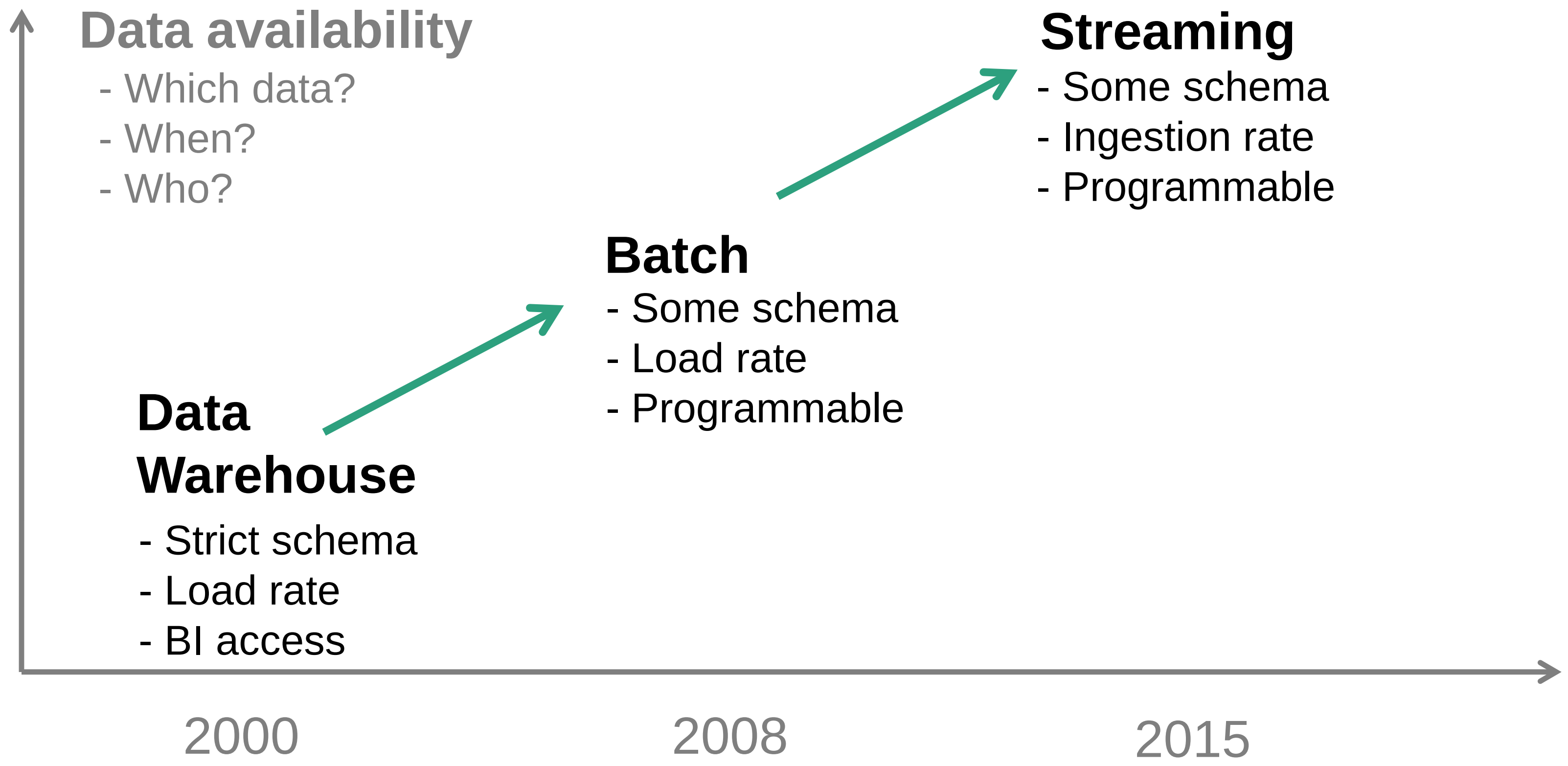
What is Operator State?

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

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

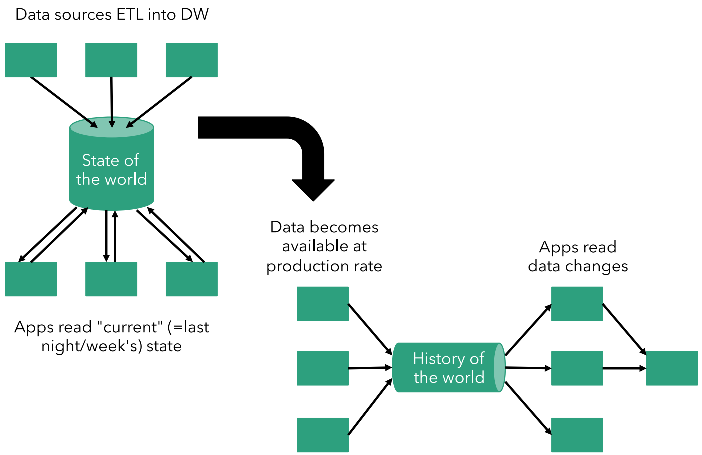
- Fault tolerance mechanism:
  - Back up and restored state stored in a backend (HDFS, Ignite, Cassandra, …)
  - After restore: replay stream from the last checkpoint

Why streaming



What does streaming enable?

1. Data integration



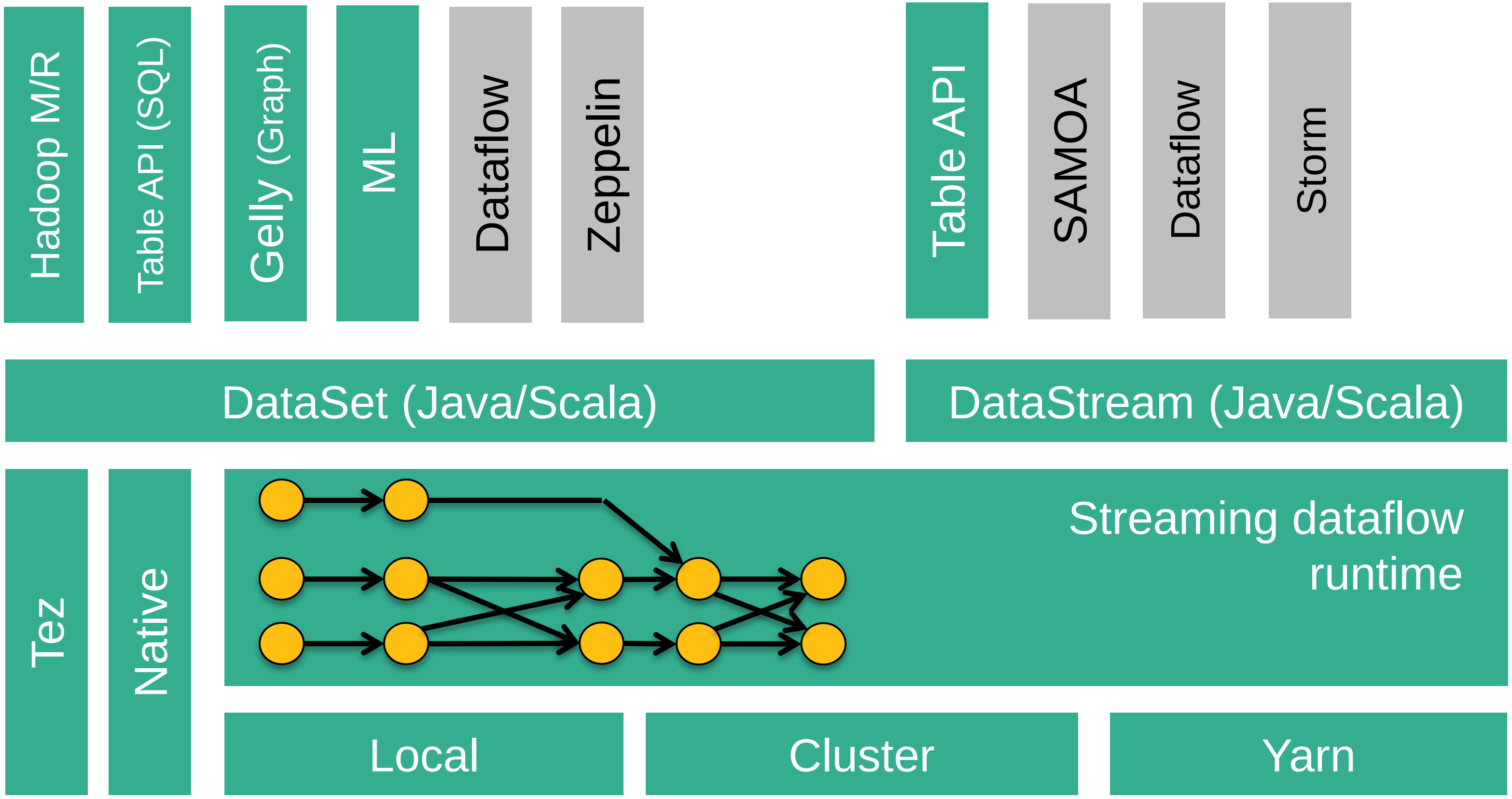
cf. Kleppmann: "Turning the DB inside out with Samza"

2. Low latency applications

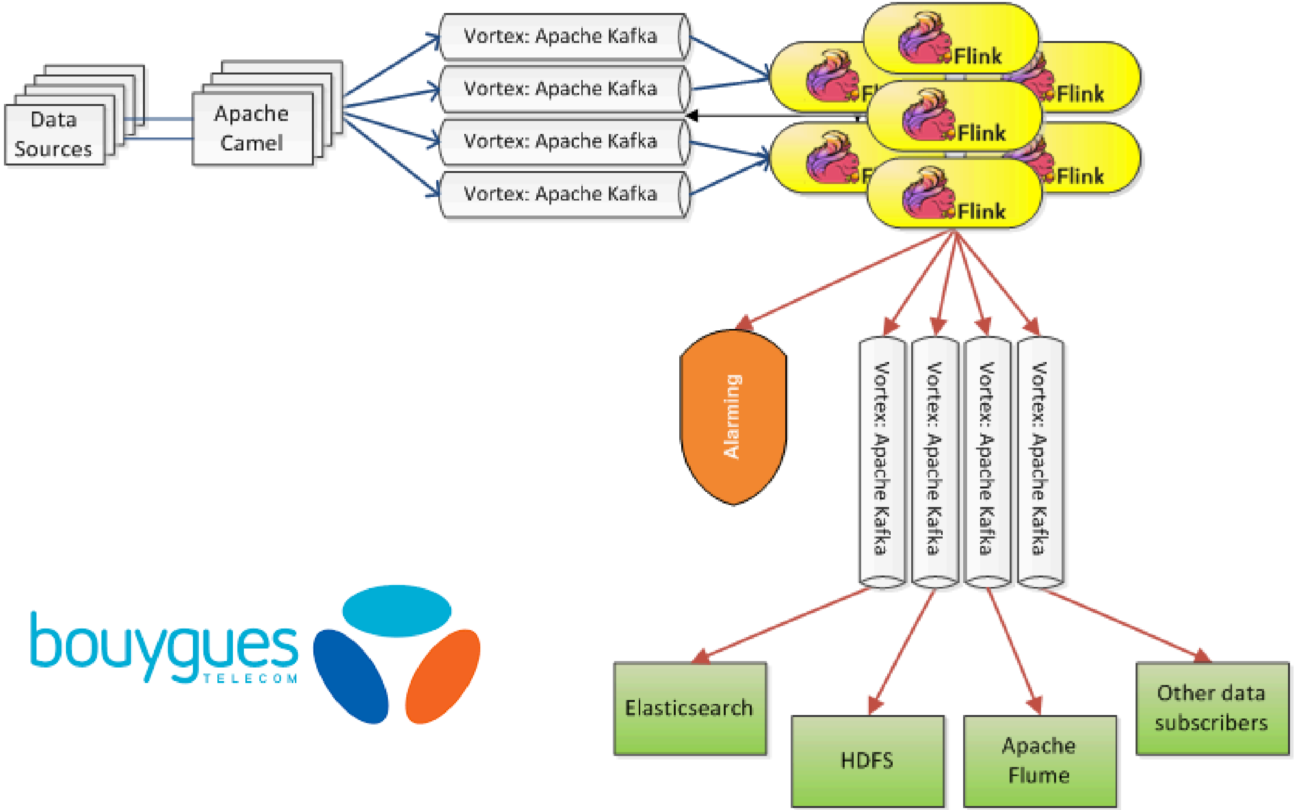
- Fresh recommendations, fraud detection, etc
- Internet of Things, intelligent manufacturing
- Results “right here, right now”

3. Batch < Streaming

The Stack

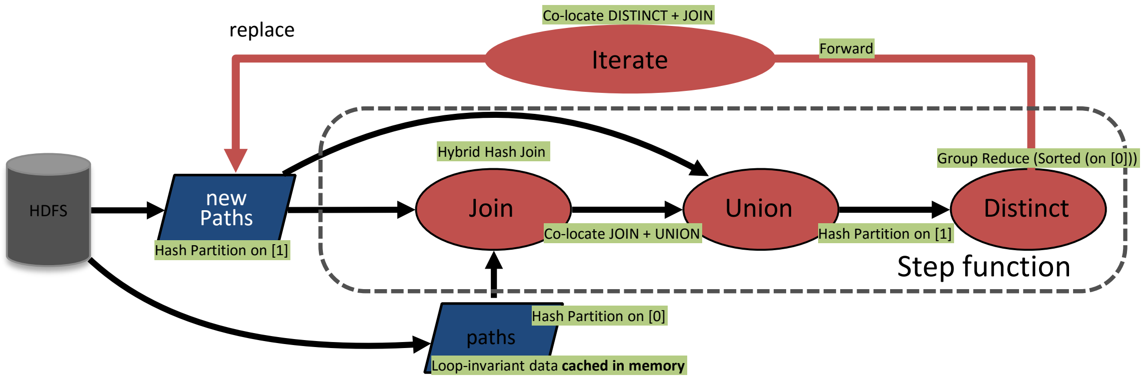

Hadoop M/R
Table API (SQL)
Gelly (Graph)
ML
Dataflow
Zeppelin
Table API
SAMOA
Dataflow
Storm
DataSet (Java/Scala)
DataStream (Java/Scala)
Tez
Native
Streaming dataflow runtime
Local
Cluster
Yarn

Example: Bouygues Telecom



bouygues TELECOM

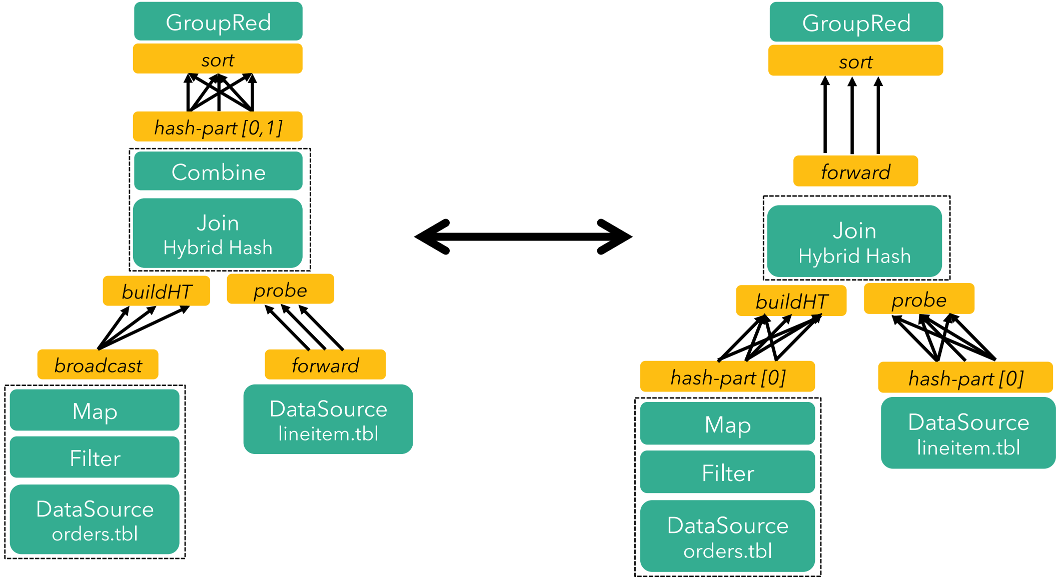
Flink Optimizer



- 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

Cost-based optimizer



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

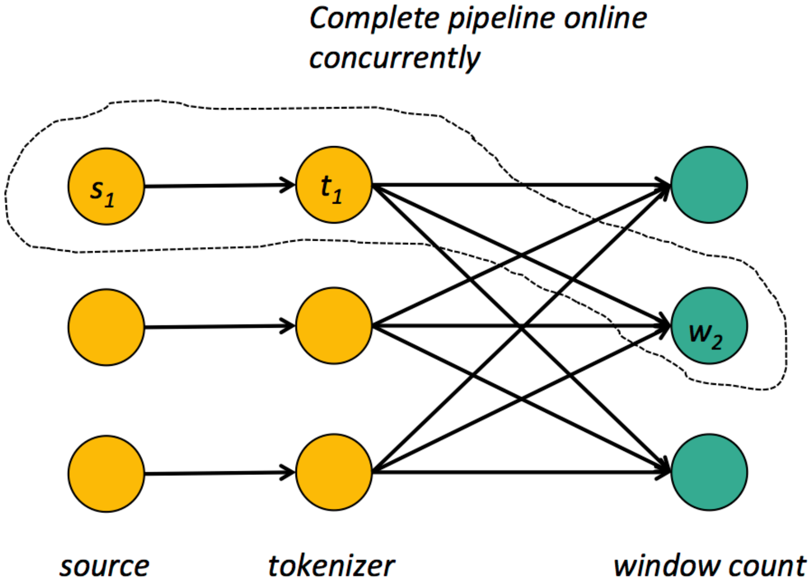
App development

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

Large deployments

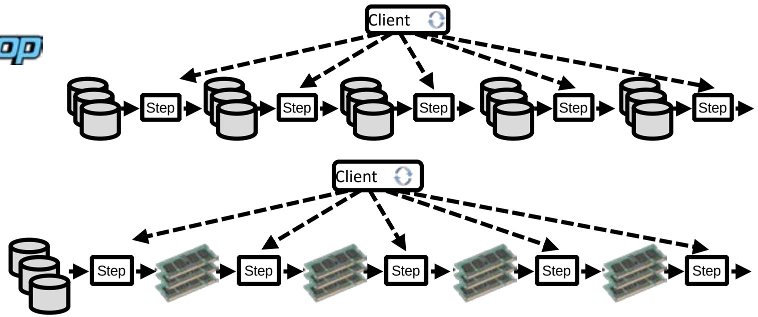
Pipelining

Basic building block to “keep the data moving”



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

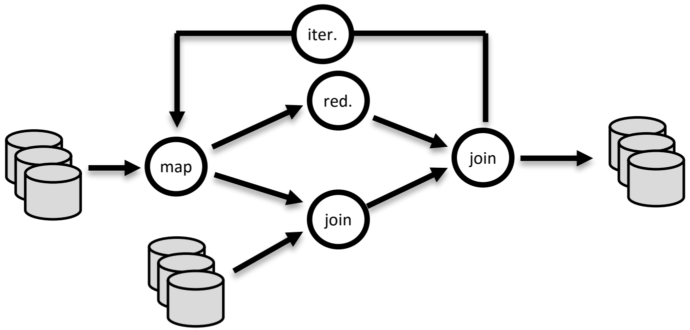
Built-in vs. driver-based looping



Loop outside the system, in driver program

Iterative program looks like many independent jobs

Flink

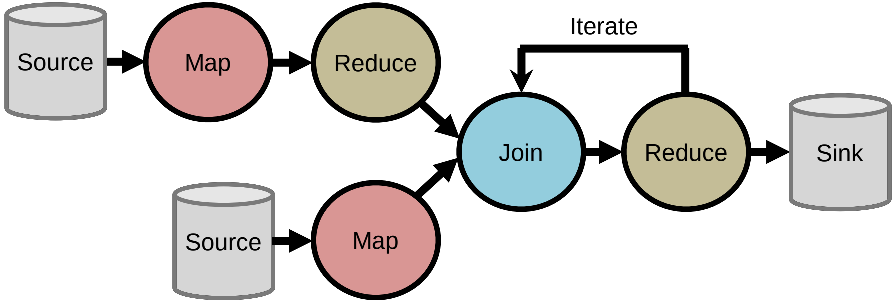


Dataflows with feedback edges

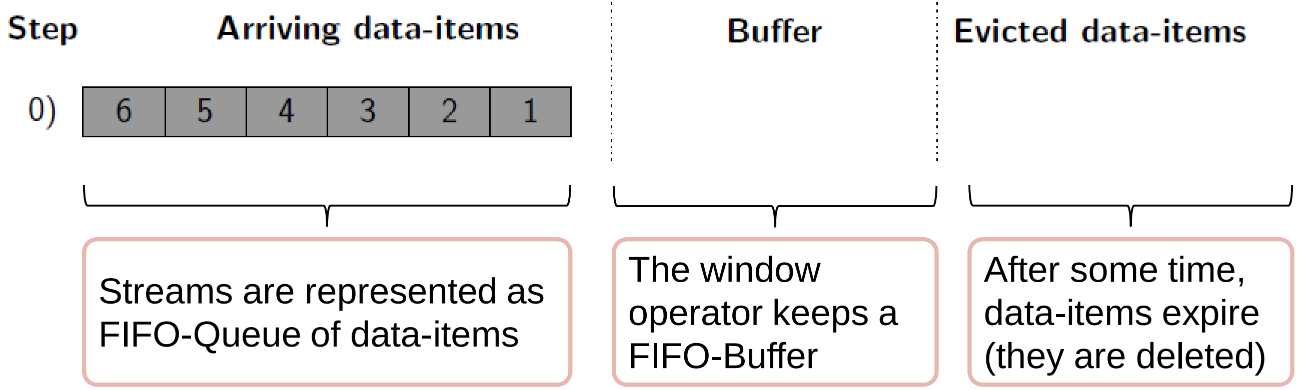
System is iteration-aware, can optimize the job

Rich set of operators

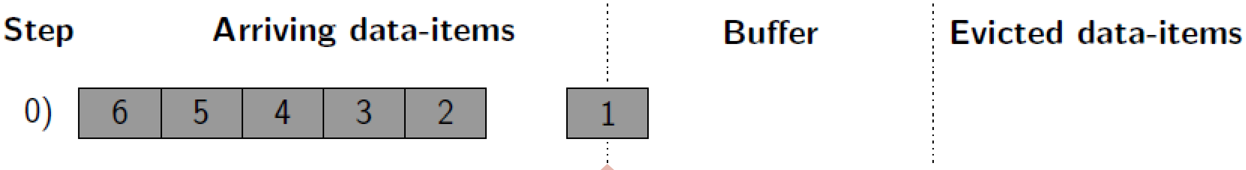
Map, Reduce, Join, CoGroup, Union, Iterate, Delta Iterate, Filter, FlatMap, GroupReduce, Project, Aggregate, Distinct, Vertex-Update, Accumulators, …



Stream Discretization Policies

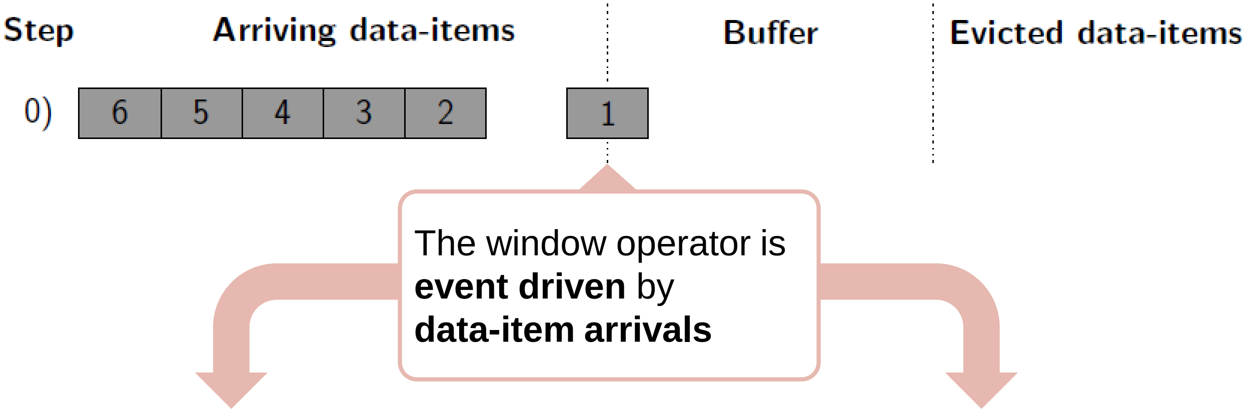


Stream Discretization Policies



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

Stream Discretization Policies



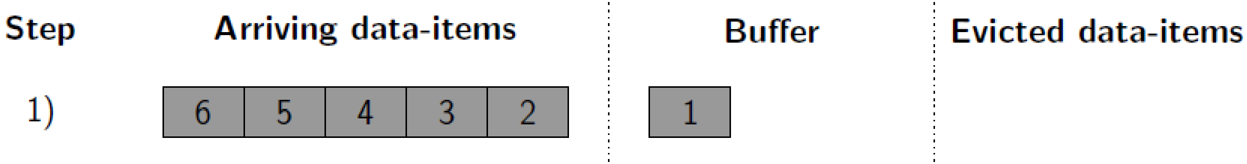
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

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



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

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

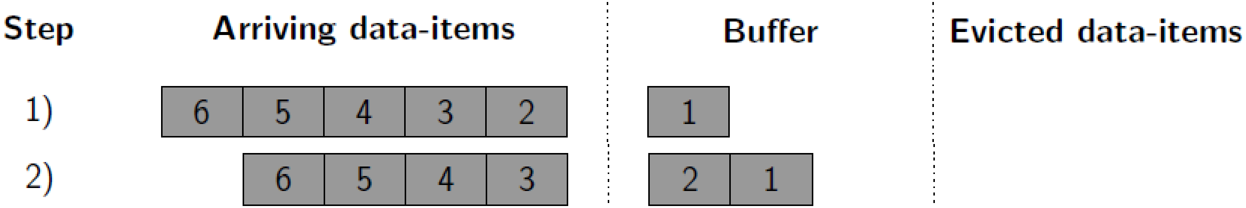
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

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



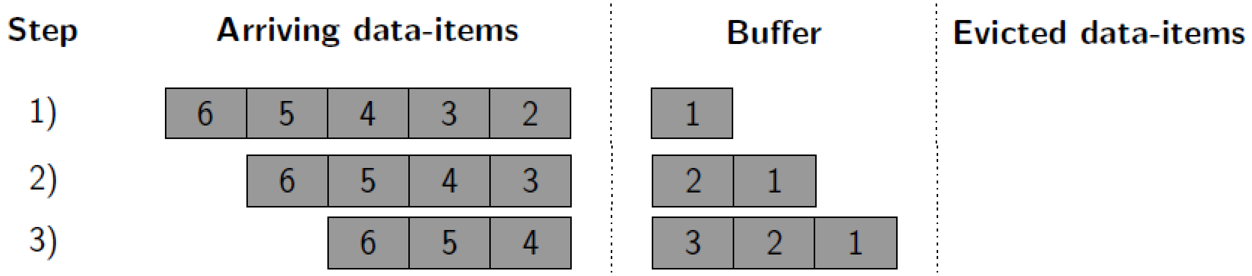
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

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



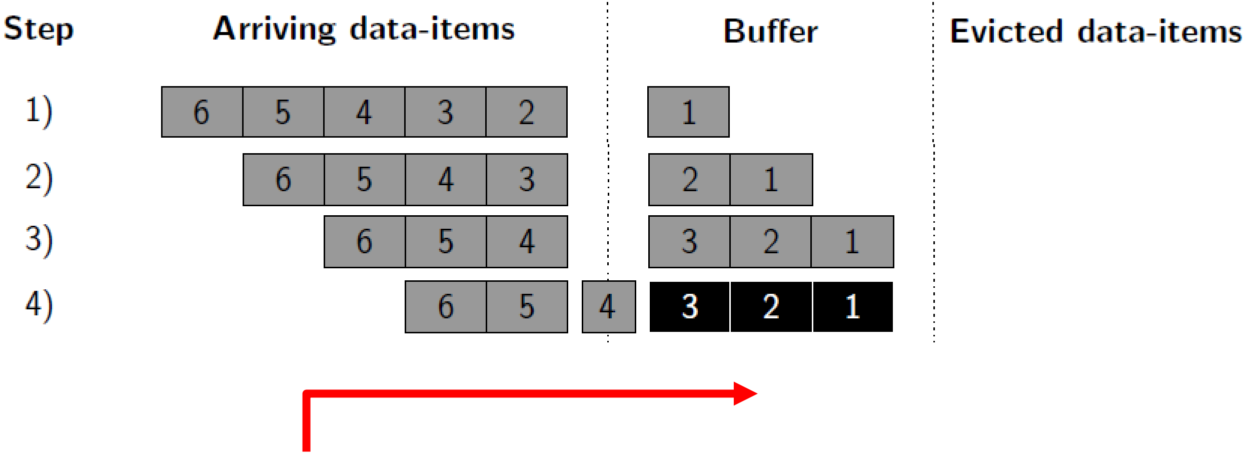
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

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



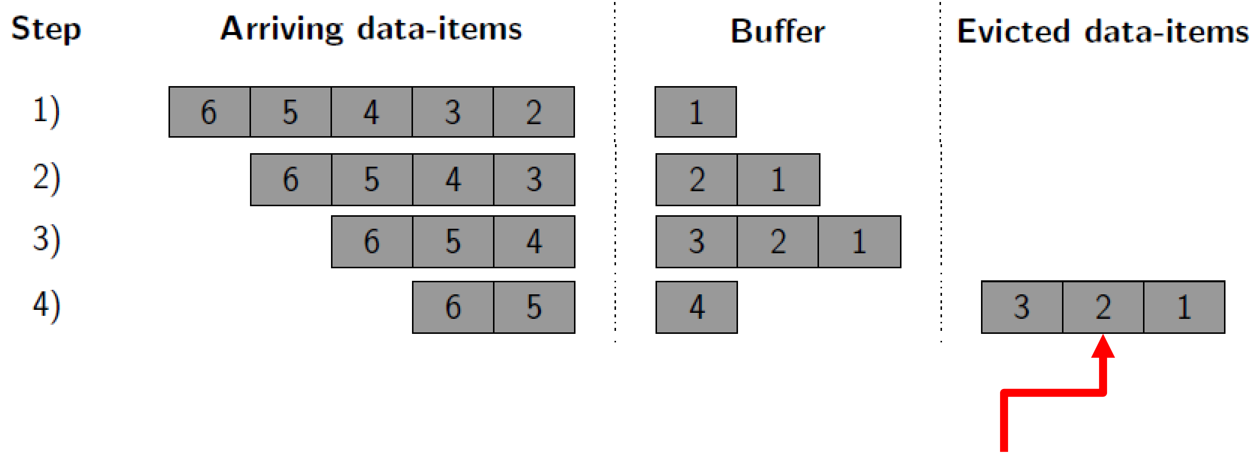
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

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



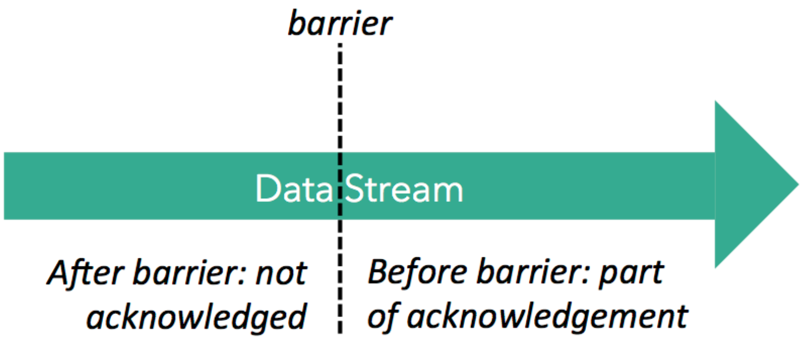
1.) Trigger Policies (TPs)
Specify when the aggregate is executed on the current buffer content.

Define the moment that results are emitted.

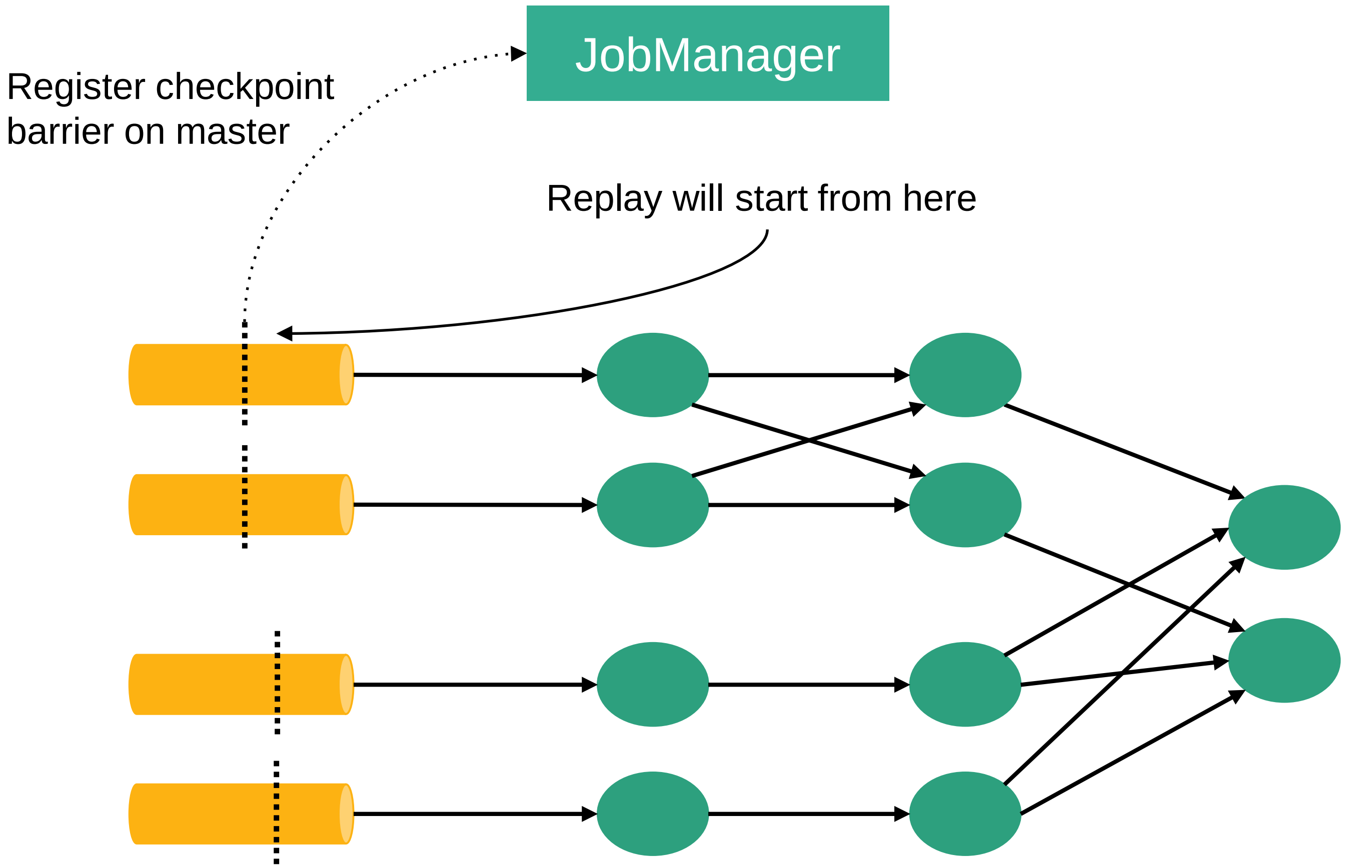
2.) Eviction Policies (EPs)
Specify when data-items are removed from the buffer.

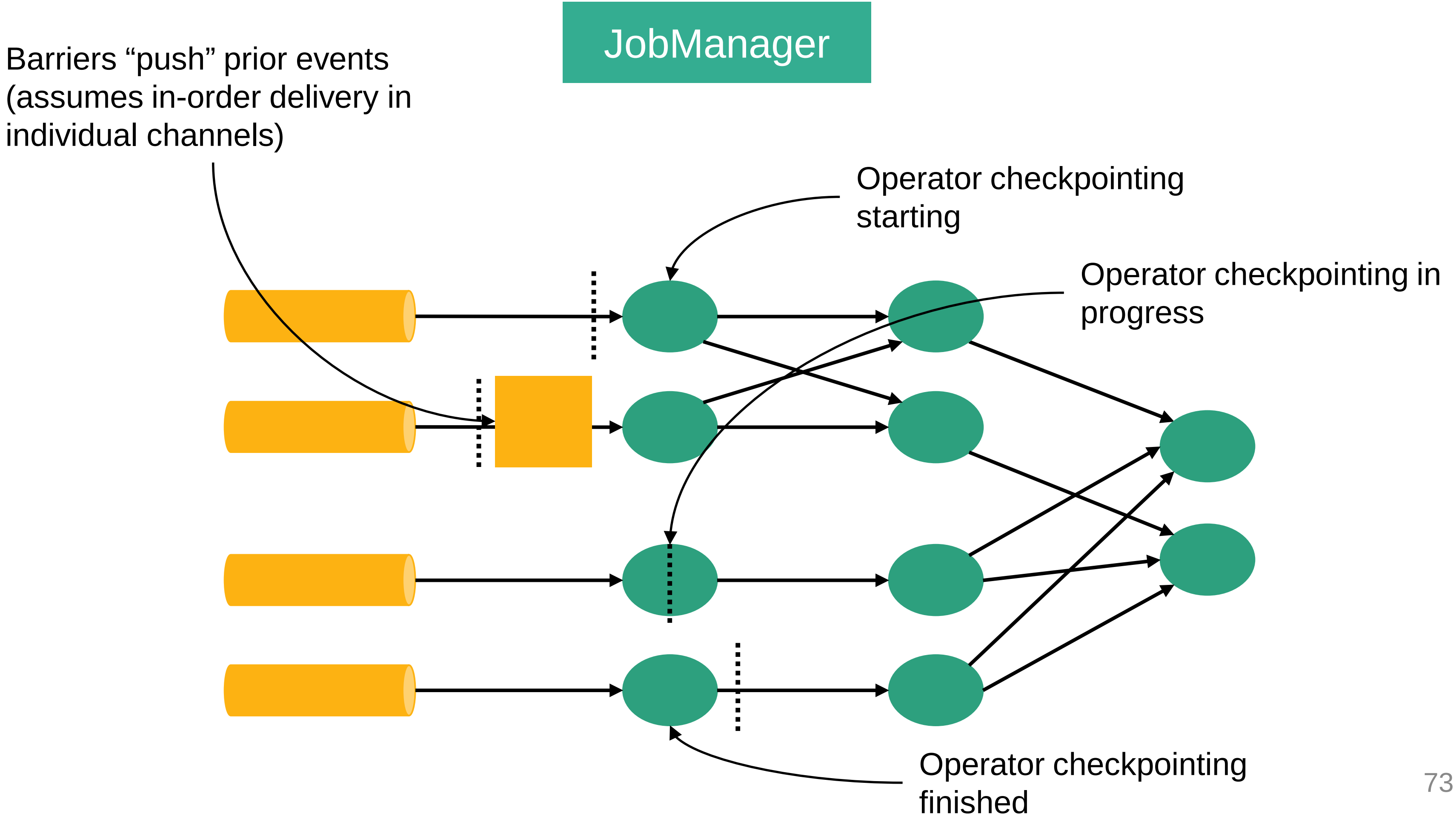
Defines the size of a window.

Distributed snapshots in Flink



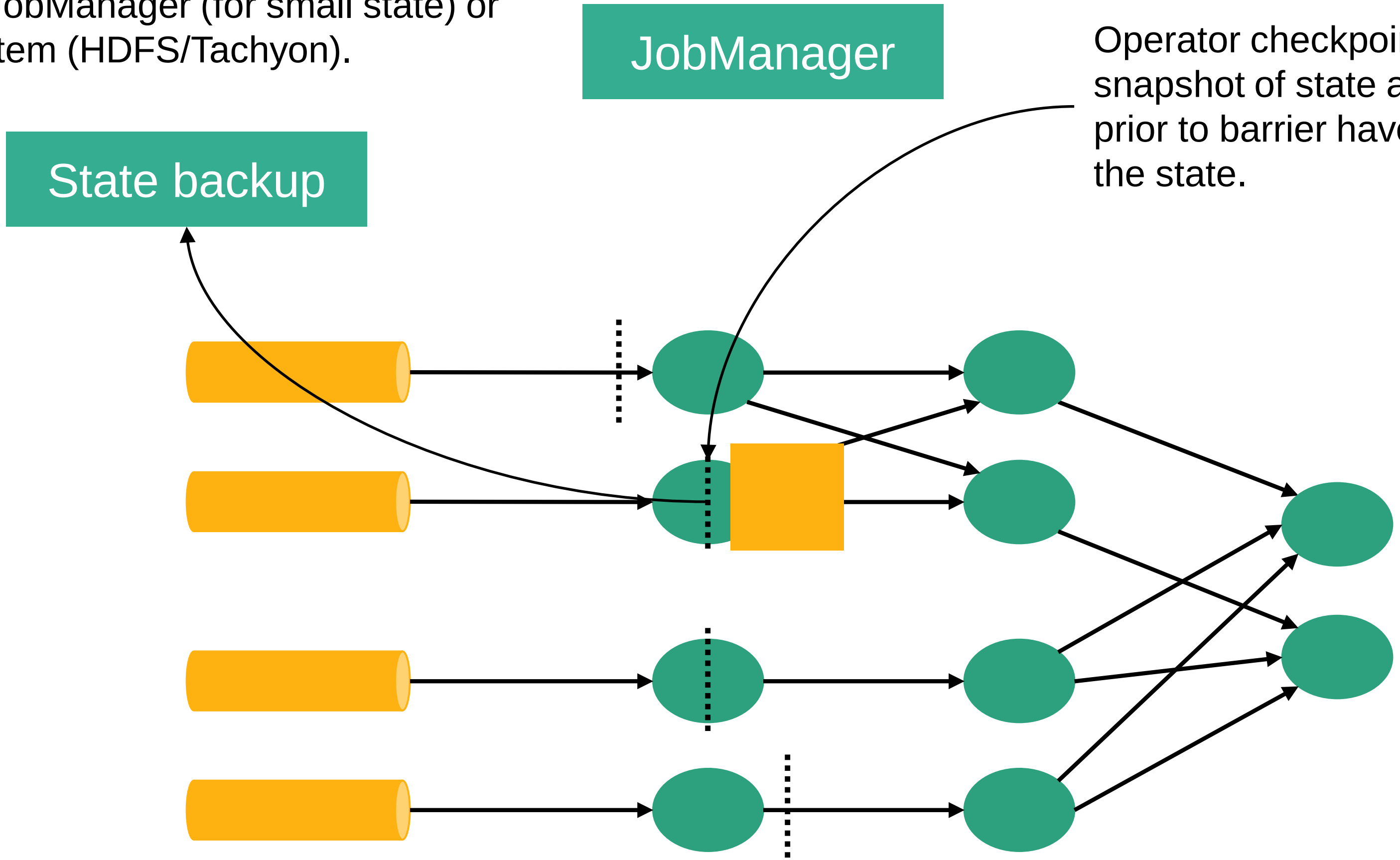
Super-impose checkpointing mechanism on execution instead of using execution as the checkpointing mechanism

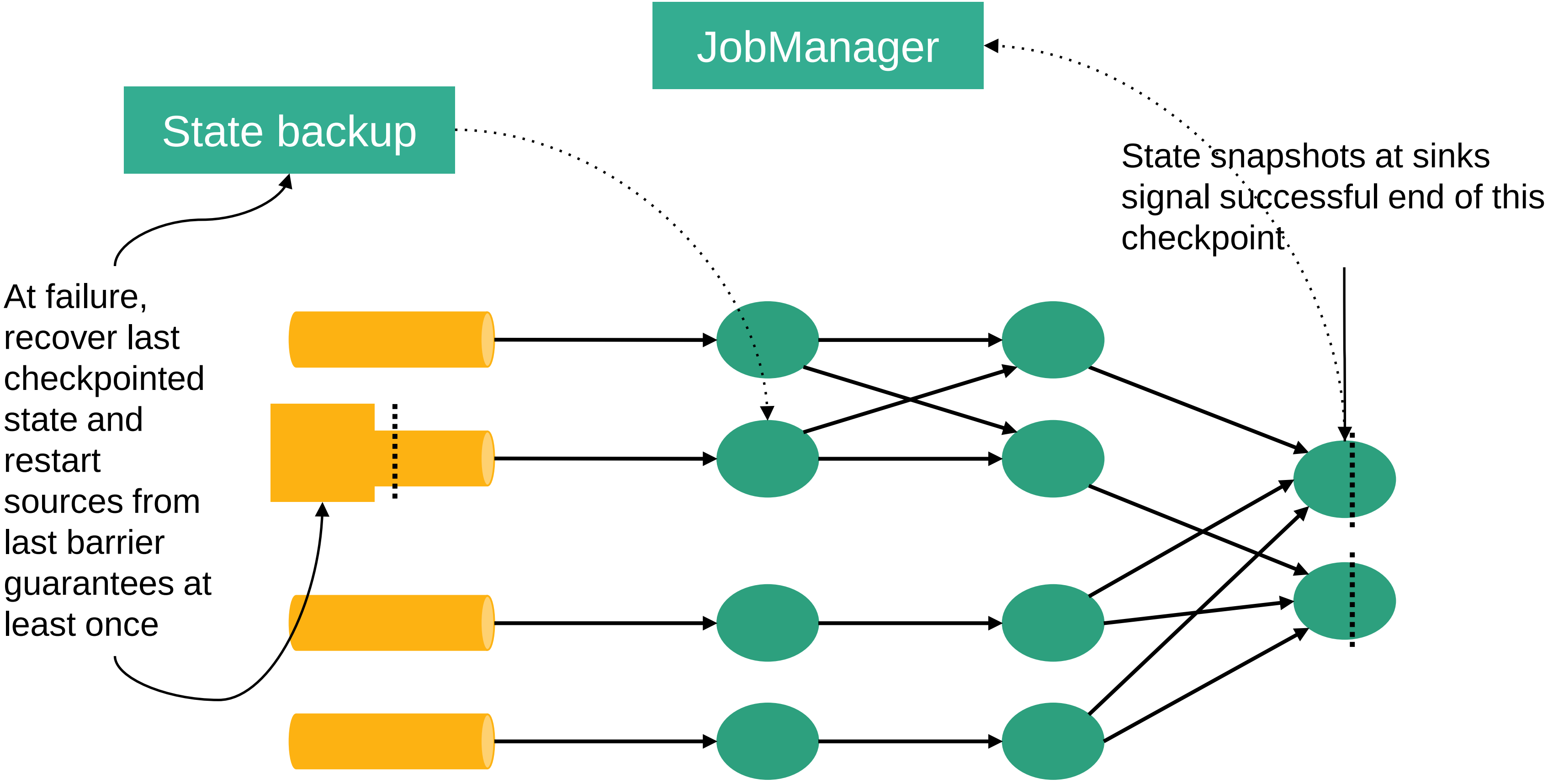

JobManager
Register checkpoint
barrier on master
Replay will start from here


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

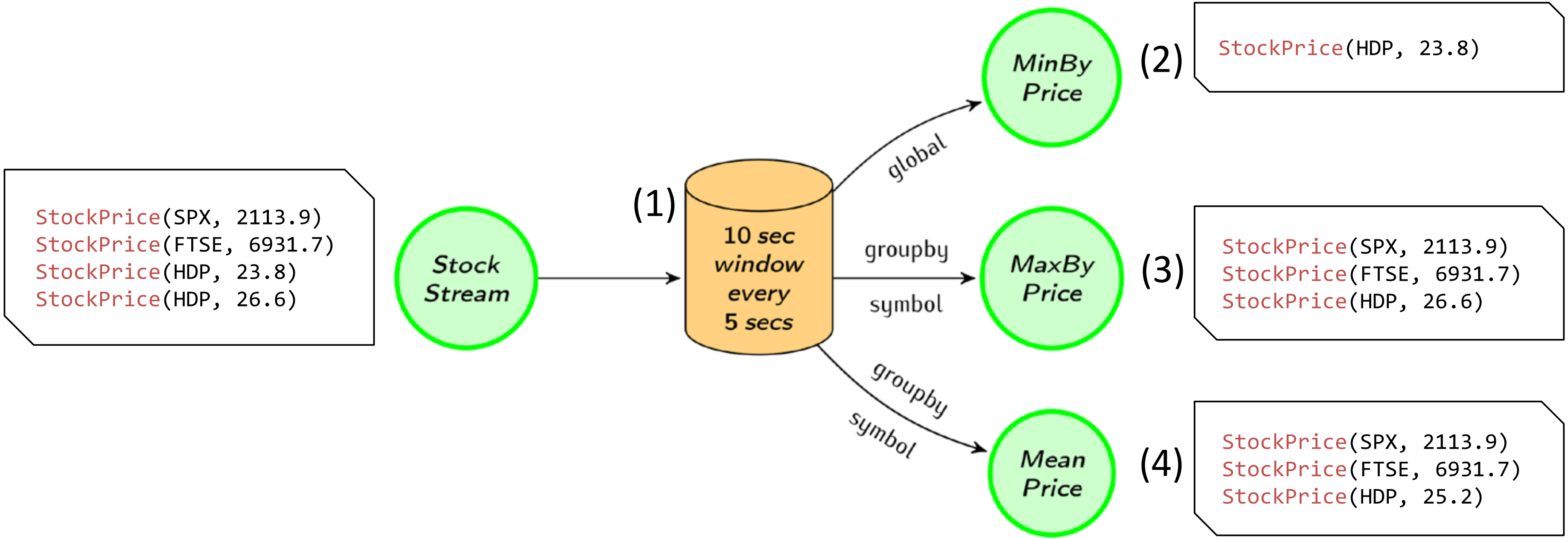
Pluggable mechanism. Currently either JobManager (for small state) or file system (HDFS/Tachyon).

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




JobManager
State backup
State snapshots at sinks signal successful end of this checkpoint
At failure, recover last checkpointed state and restart sources from last barrier guarantees at least once

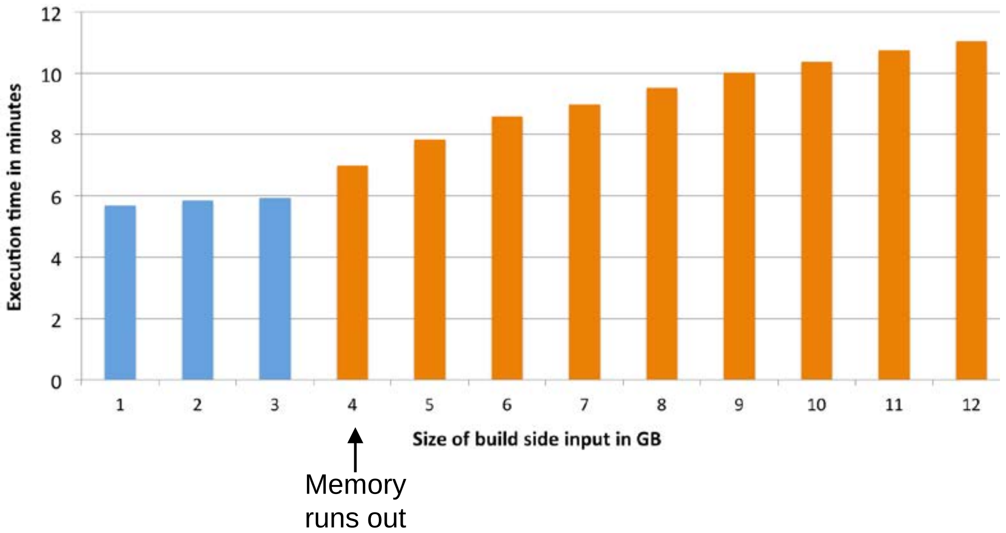
Example Analysis: Windowed Aggregation




```
(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)



Quiz: guess the algorithm!

```
... // initialize
while (theta) {
  newCntrds = points
    .map(findNearestCntrd)
    .map( (c, p) => (c, (p, 1L)) )
    .reduceByKey( (x, y) =>
      (x._1 + y._1, x._2 + y._2) )
    .map( x => Centroid(x._1, x._2._1 / x._2._2) )

  bcCntrs = sc.broadcast(newCntrds.collect())
}
```

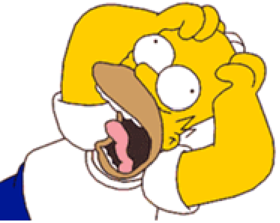
```
... // initialize
val cntrds = centroids.iterate(theta) { currCntrds =>
  val newCntrds = points
    .map(findNearestCntrd).withBcSet(currCntrds, "cntrds")
    .map( (c, p) => (c, p, 1L) )
    .groupBy(0).reduce( (x, y) =>
      (x._1, x._2 + y._2, x._3 + y._3) )
    .map( x => Centroid(x._1, x._2 / x._3) )

  currCntrds
}
```

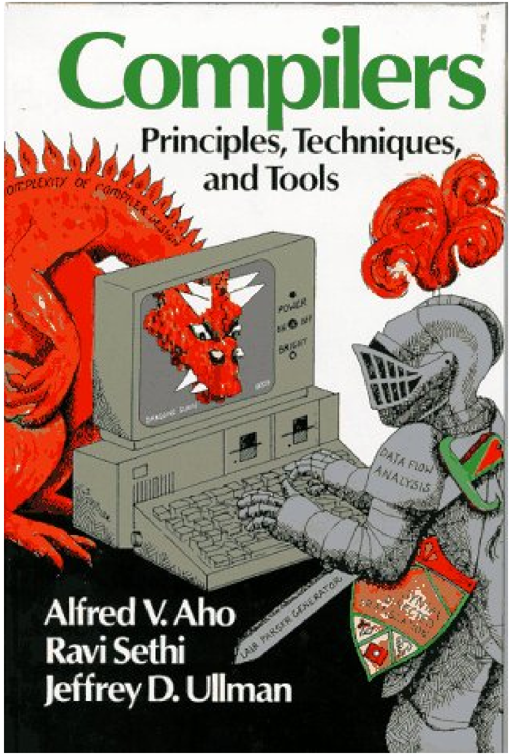
Problem Statement

- Runtime-centric evolution of the APIs results in
  - Too much low-level aspects exposed
  - Hard to teach people how and when to use them
  - Affects productivity
  - Neglects optimization potential

- **We are hard-coding execution plans!**
- Back in the 70s? Can we do better?



Compilers to the Rescue!



- Deep language embedding
- A holistic view of the complete data analysis enables
  - Parallelism transparency (SPJ + nesting)
  - Advanced Optimizations

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