Apache Flink

Stephan Ewen

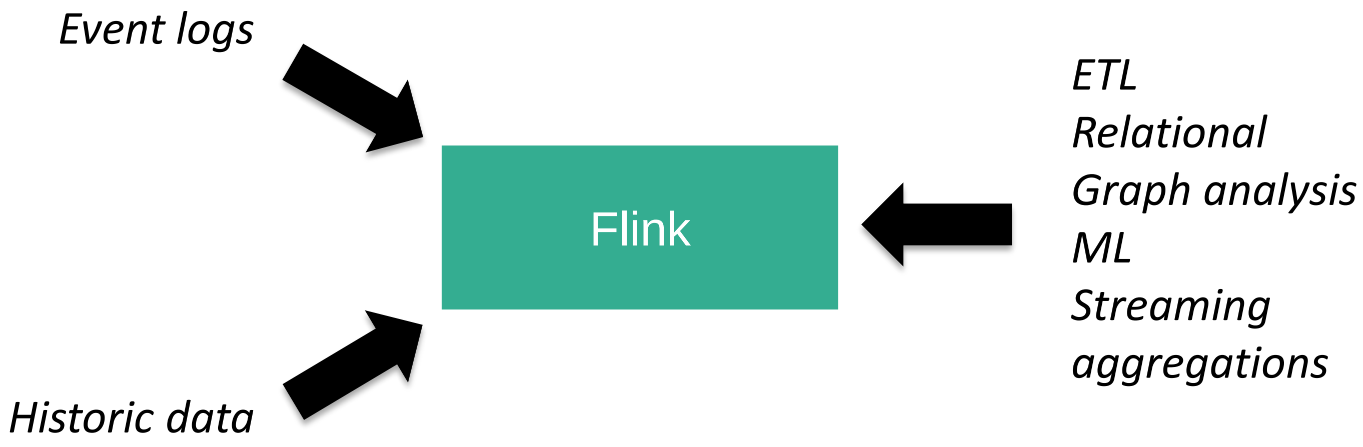
Flink committer

co-founder / CTO @ *data Artisans*

@StephanEwen

What is Apache Flink?

A "use-case complete" framework to unify batch & stream processing



Event logs

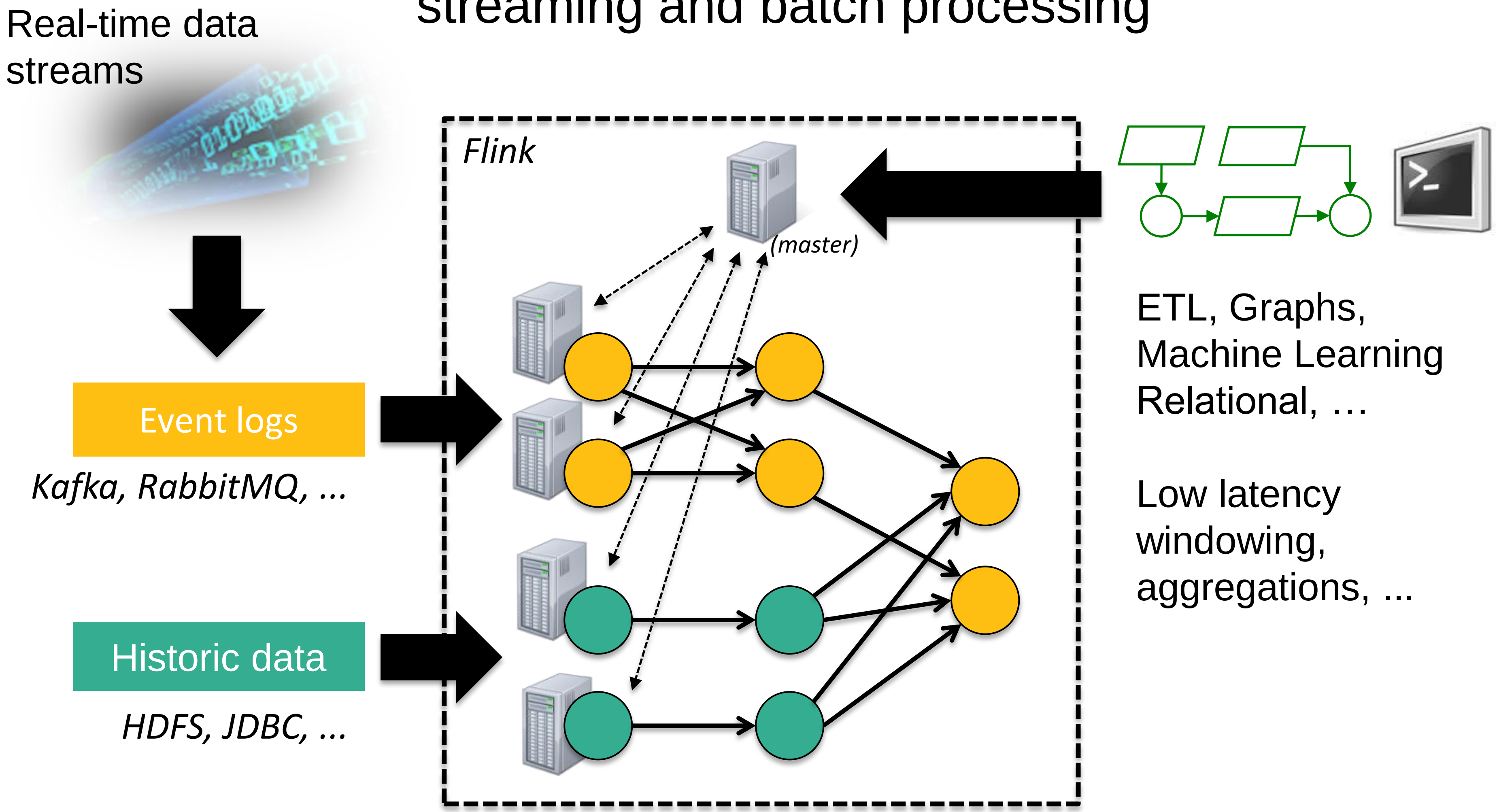
Historic data

Flink

ETL
Relational
Graph analysis
ML
Streaming aggregations

What is Apache Flink?

An engine that puts equal emphasis on streaming and batch processing



Real-time data streams

Event logs

Kafka, RabbitMQ, ...

Historic data

HDFS, JDBC, ...

ETL, Graphs, Machine Learning Relational, …

Low latency windowing, aggregations, ...

What is Apache Flink?

- Large-scale data processing engine
- Easy and powerful APIs for *batch and real-time streaming* analysis (Java / Scala)
- Backed by a robust execution backend
  - with true streaming capabilities,
  - sophisticated windowing mechanisms,
  - custom memory manager,
  - native iteration execution,
  - and a cost-based optimizer.

Cornerpoints of Flink Design

Flexible Data Streaming Engine

→ *Low Latency Steam Proc.*
→ *Highly flexible windows*

Robust Algorithms on Managed Memory

→ *No OutOfMemory Errors*
→ *Scales to very large JVMs*
→ *Efficient an robust processing*

High-level APIs, beyond key/value pairs

→ *Java/Scala/Python (upcoming)*
→ *Relational-style optimizer*

Pipelined Execution of Batch Programs

→ *Better shuffle performance*
→ *Scales to very large groups*

Active Library Development

→ *Graphs / Machine Learning*
→ *Streaming ML (coming)*

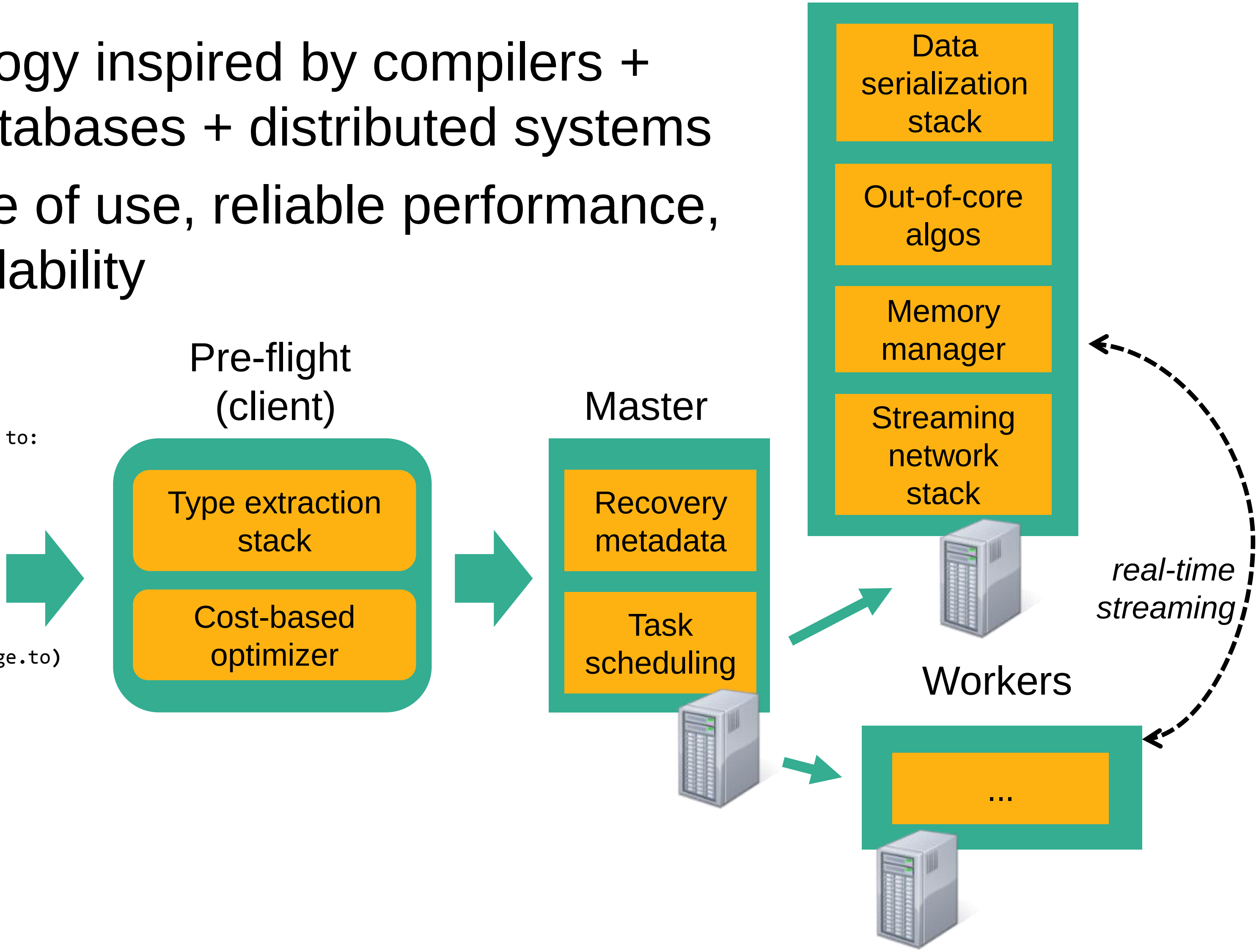
Native Iterations

→ *Very fast Graph Processing*
→ *Stateful Iterations for ML*

Technology inside Flink

- Technology inspired by compilers + MPP databases + distributed systems
- For ease of use, reliable performance, and scalability

```
case class Path (from: Long, to:
Long)
val tc = edges.iterate(10) {
  paths: DataSet[Path] =>
    val next = paths
      .join(edges)
      .where("to")
      .equalTo("from") {
        (path, edge) =>
          Path(path.from, edge.to)
      }
      .union(paths)
      .distinct()
    next
  }
```



Pre-flight (client)

Type extraction stack

Cost-based optimizer

Master

Recovery metadata

Task scheduling

Data serialization stack

Out-of-core algos

Memory manager

Streaming network stack

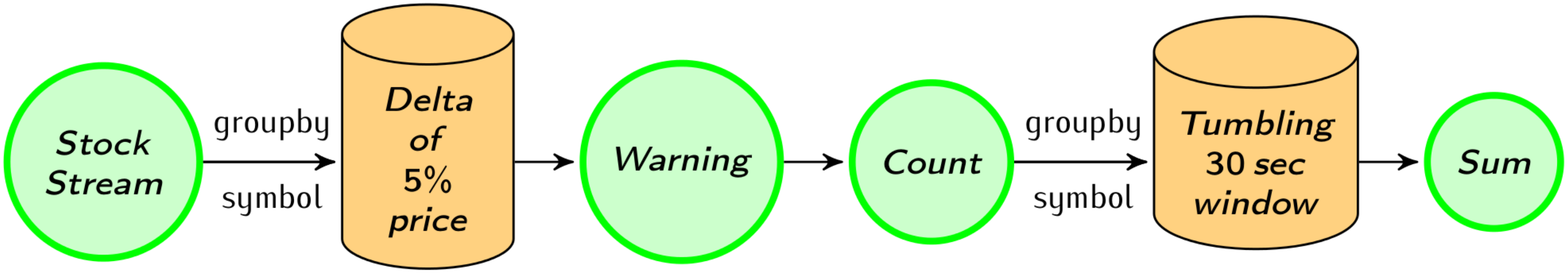
Workers

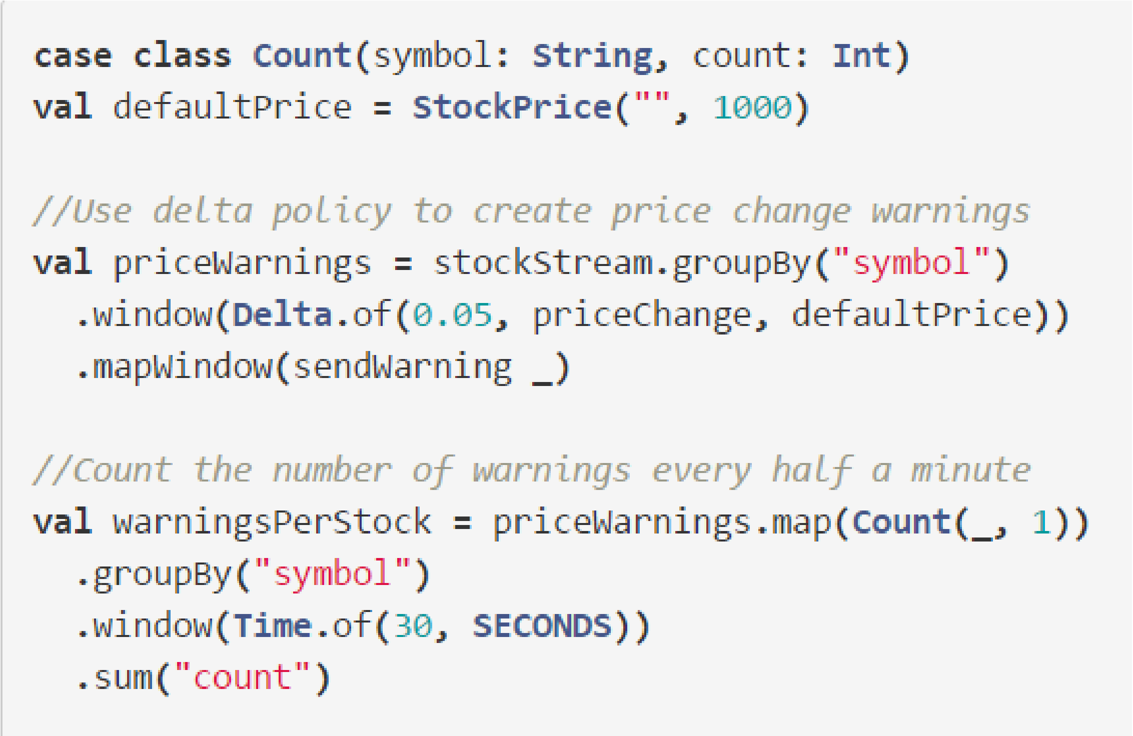
...

real-time streaming

What can you do with Flink?

Streaming Data Analysis





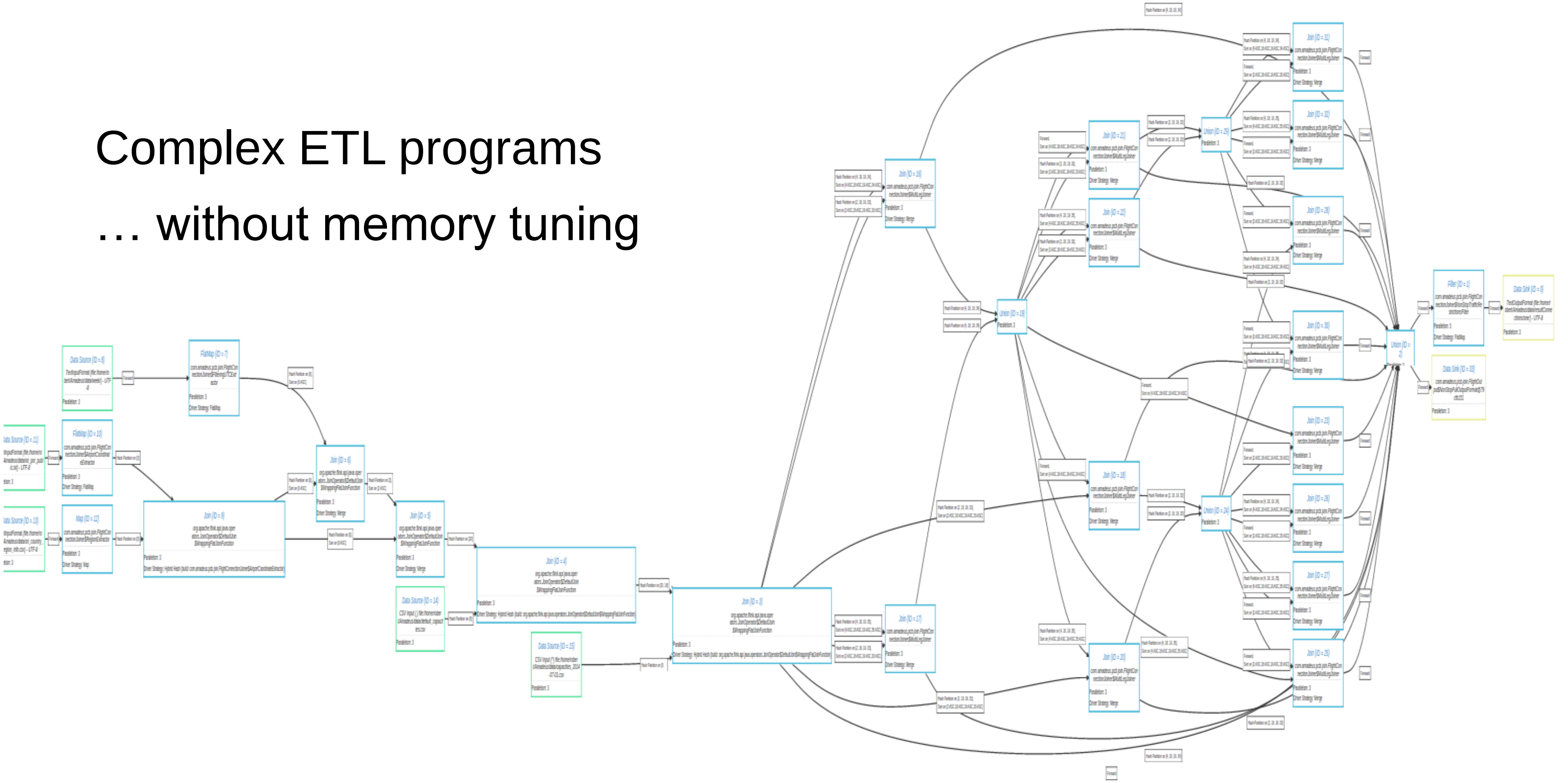
```
case class Count(symbol: String, count: Int)
val defaultPrice = StockPrice("", 1000)

//Use delta policy to create price change warnings
val priceWarnings = stockStream.groupBy("symbol")
  .window(Delta.of(0.05, priceChange, defaultPrice))
  .mapWindow(sendWarning _)

//Count the number of warnings every half a minute
val warningsPerStock = priceWarnings.map(Count(_, 1))
  .groupBy("symbol")
  .window(Time.of(30, SECONDS))
  .sum("count")
```

More at: http://flink.apache.org/news/2015/02/09/streaming-example.html

Heavy Data Pipelines

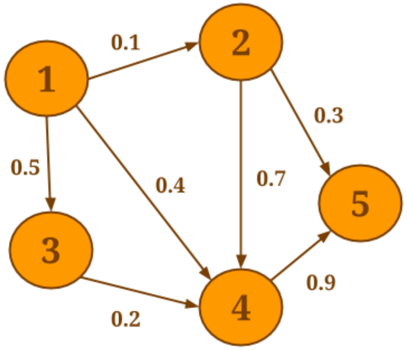
Complex ETL programs

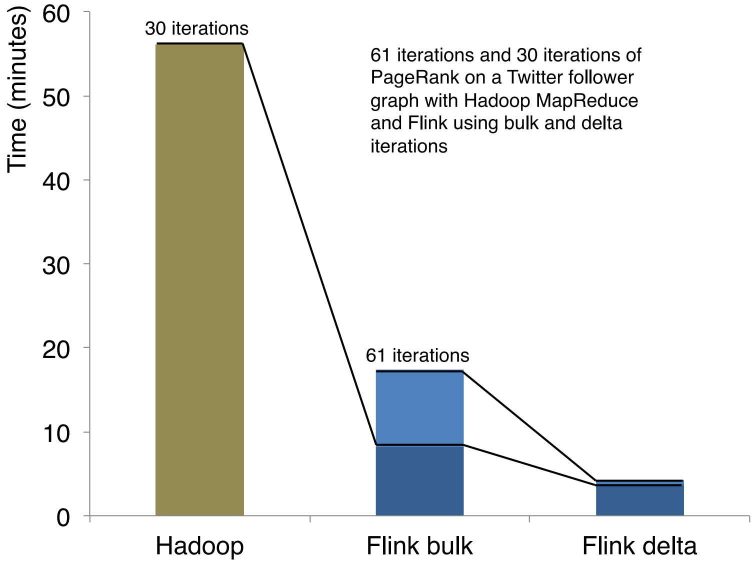
… without memory tuning


Data Source (ID = 8)
FlatMap (ID = 7)
Data Source (ID = 11)
FlatMap (ID = 10)
Data Source (ID = 13)
Map (ID = 12)
Join (ID = 9)
Join (ID = 6)
Join (ID = 5)
Data Source (ID = 14)
Join (ID = 4)
Data Source (ID = 15)
Join (ID = 3)
Forward
Hash Partition on [0]
Sort on [0:ASC]
Parallelism: 3
Driver Strategy: FlatMap
Driver Strategy: Map
Driver Strategy: Merge
com.amadeus.pcb.join.FlightConnectionJoiner$FilteringUTCExtractor
com.amadeus.pcb.join.FlightConnectionJoiner$AirportCoordinateExtractor
com.amadeus.pcb.join.FlightConnectionJoiner$RegionExtractor
org.apache.flink.api.java.operators.JoinOperator$DefaultJoin$WrappingFlatJoinFunction
Driver Strategy: Hybrid Hash (build: com.amadeus.pcb.join.FlightConnectionJoiner$AirportCoordinateExtractor)

Very fast graph analysis

… and mix and match
ETL-style and graph analysis
in one program





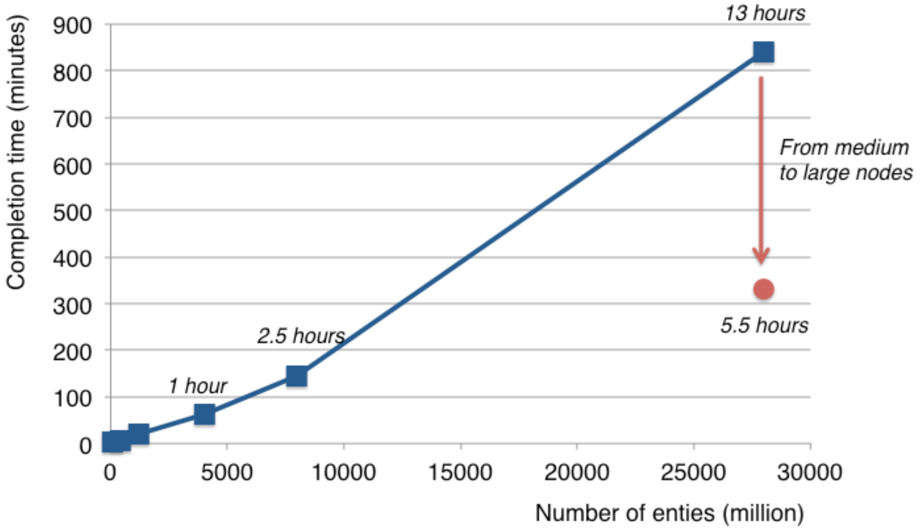
61 iterations and 30 iterations of PageRank on a Twitter follower graph with Hadoop MapReduce and Flink using bulk and delta iterations

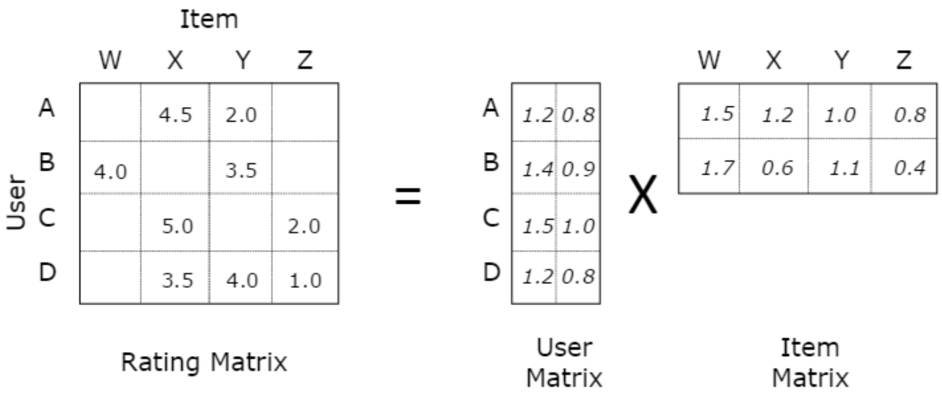
Performance competitive
with dedicated graph
analysis systems

More at: http://data-artisans.com/data-analysis-with-flink.html

Large-Scale Machine Learning

Factorizing a matrix with 28 billion ratings for recommendations





$$\begin{array}{c|cccc} & W & X & Y & Z \\ \hline A & & 4.5 & 2.0 & \\ B & 4.0 & & 3.5 & \\ C & & 5.0 & & 2.0 \\ D & & 3.5 & 4.0 & 1.0 \end{array} = \begin{array}{c|cc} A & 1.2 & 0.8 \\ B & 1.4 & 0.9 \\ C & 1.5 & 1.0 \\ D & 1.2 & 0.8 \end{array} \times \begin{array}{cccc} W & X & Y & Z \\ \hline 1.5 & 1.2 & 1.0 & 0.8 \\ 1.7 & 0.6 & 1.1 & 0.4 \end{array}$$

(Scale of Netflix or Spotify)

More at: http://data-artisans.com/computing-recommendations-with-flink.html

How do you use Flink?

Example: WordCount

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

val env = ExecutionEnvironment.getExecutionEnvironment()

val lines = env.readTextFile(...)

lines
   .flatMap {line => line.split(" ").map(word => Word(word,1))}
   .groupBy("word").sum("frequency”)
   .print()

env.execute()
```

Flink has mirrored Java and Scala APIs that offer the same functionality, including by-name addressing.

Example: Window WordCount

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

val env = StreamExecutionEnvironment.getExecutionEnvironment()

val lines = env.fromSocketStream(...)

lines
   .flatMap {line => line.split(" ").map(word => Word(word,1))}
   .window(Count.of(100)).every(Count.of(10))
   .groupBy("word").sum("frequency").print()

env.execute()
```

Flink API in a Nutshell

- map, flatMap, filter, groupBy, reduce, reduceGroup, aggregate, join, coGroup, cross, project, distinct, union, iterate, iterateDelta, ...
- All Hadoop input formats are supported
- API similar for data sets and data streams with slightly different operator semantics
- Window functions for data streams
- Counters, accumulators, and broadcast variables

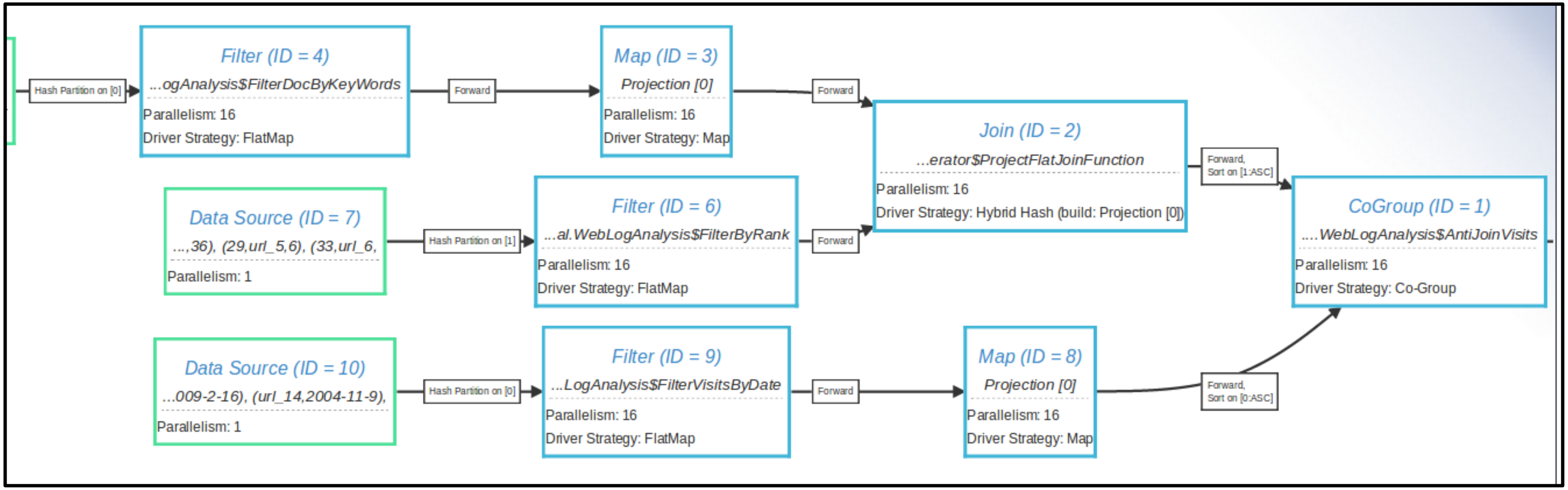
Defining windows

- Trigger policy
  - When to trigger the computation on current window
- Eviction policy
  - When data points should leave the window
  - Defines window width/size
- E.g., count-based policy
  - evict when #elements > n
  - start a new window every n-th element
- Built-in: Count, Time, Delta policies

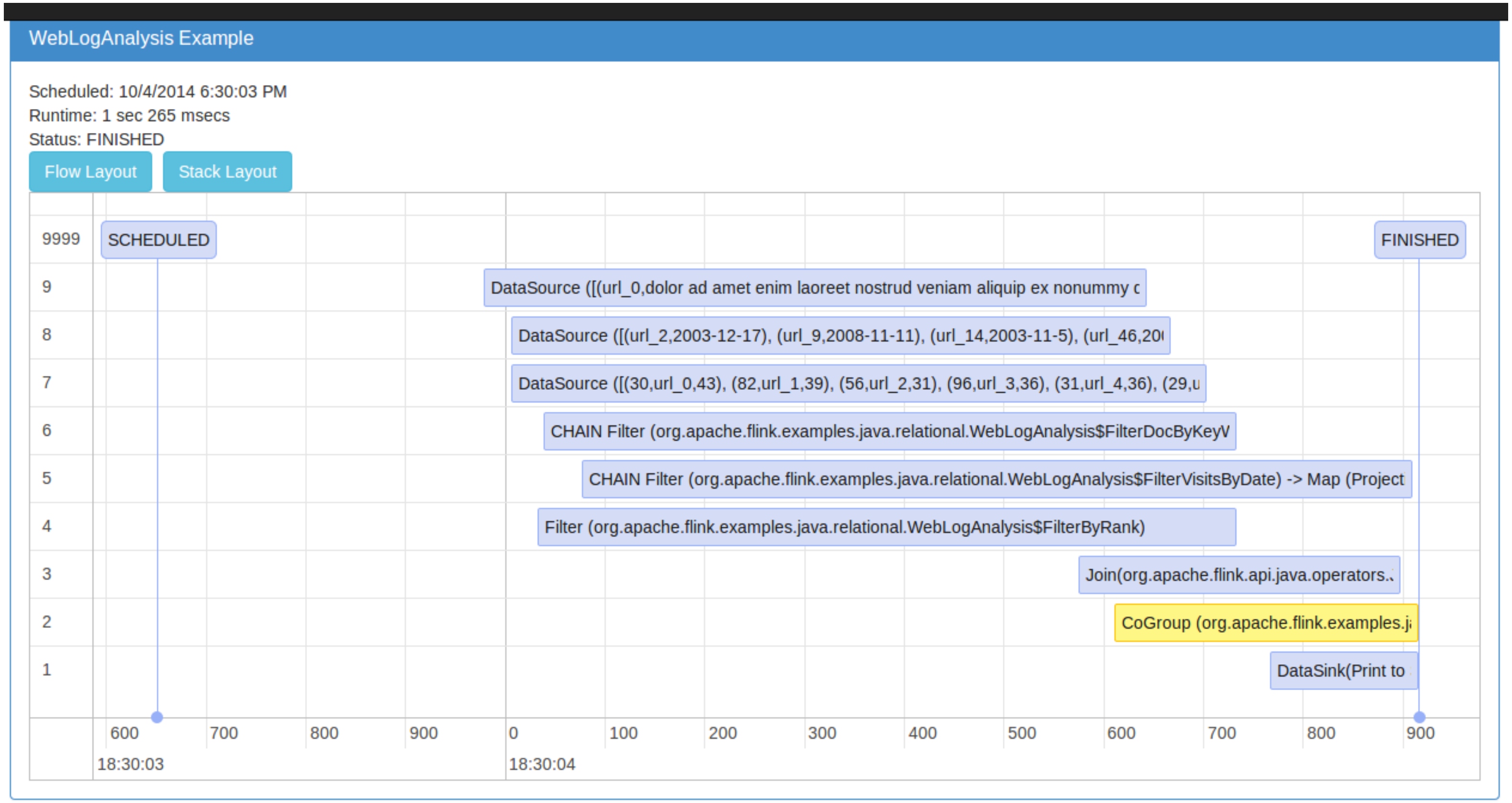
Table API

```
val customers = envreadCsvFile(…).as('id, 'mktSegment)
      .filter( 'mktSegment === "AUTOMOBILE" )

val orders = env.readCsvFile(…)
      .filter( o => dateFormat.parse(o.orderDate).before(date) )
      .as('orderId, 'custId, 'orderDate, 'shipPrio)

val items = orders
      .join(customers).where('custId === 'id)
      .join(lineitems).where('orderId === 'id)
      .select('orderId,'orderDate,'shipPrio,
          'extdPrice * (Literal(1.0f) - 'discount) as 'revenue)

val result = items
      .groupBy('orderId, 'orderDate, 'shipPrio)
      .select('orderId, 'revenue.sum, 'orderDate, 'shipPrio)
```

Visualization tools



Visualization tools

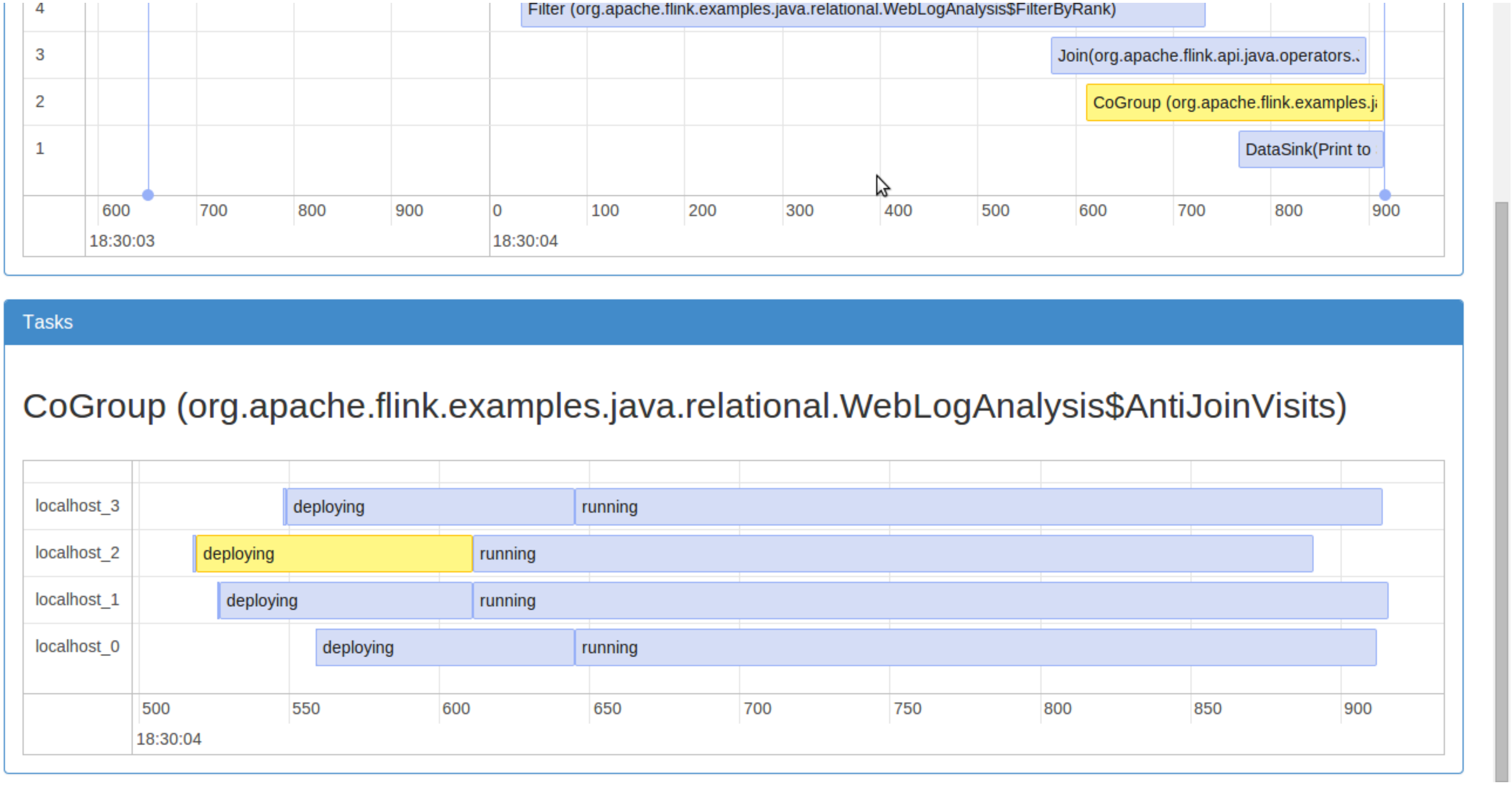


WebLogAnalysis Example

Scheduled: 10/4/2014 6:30:03 PM
Runtime: 1 sec 265 msecs
Status: FINISHED

Flow Layout
Stack Layout
9999
SCHEDULED
FINISHED
9
DataSource ([(url_0,dolor ad amet enim laoreet nostrud veniam aliquip ex nonummy c
8
DataSource ([(url_2,2003-12-17), (url_9,2008-11-11), (url_14,2003-11-5), (url_46,20
7
DataSource ([(30,url_0,43), (82,url_1,39), (56,url_2,31), (96,url_3,36), (31,url_4,36), (29,u
6
CHAIN Filter (org.apache.flink.examples.java.relational.WebLogAnalysis$FilterDocByKeyW
5
CHAIN Filter (org.apache.flink.examples.java.relational.WebLogAnalysis$FilterVisitsByDate) -> Map (Project
4
Filter (org.apache.flink.examples.java.relational.WebLogAnalysis$FilterByRank)
3
Join(org.apache.flink.api.java.operators..
2
CoGroup (org.apache.flink.examples.j
1
DataSink(Print to
600
700
800
900
0
100
200
300
400
500
600
700
800
900
18:30:03
18:30:04

Visualization tools



4
Filter (org.apache.flink.examples.java.relational.WebLogAnalysis$FilterByRank)
3
Join(org.apache.flink.api.java.operators..
2
CoGroup (org.apache.flink.examples.j
1
DataSink(Print to
600
700
800
900
0
100
200
300
400
500
600
700
800
900
18:30:03
18:30:04

Tasks

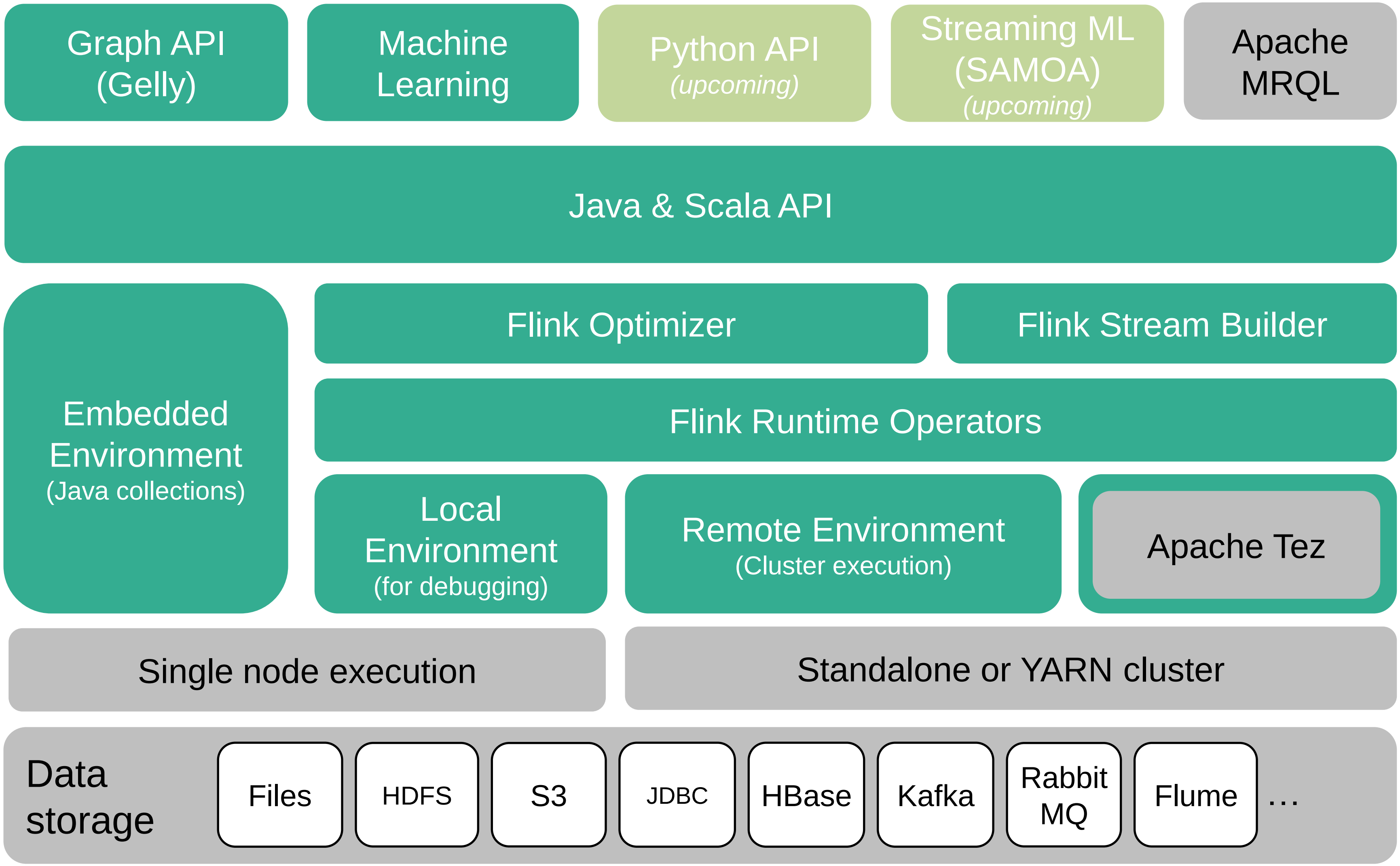
CoGroup (org.apache.flink.examples.java.relational.WebLogAnalysis$AntiJoinVisits)

Flink Architecture

The case for Apache Flink

- Performance and ease of use
  - Exploits in-memory and pipelining, language-embedded logical APIs
- Unified batch and real streaming
  - Batch and Stream APIs on top of a streaming engine
- A runtime that "just works" without tuning
  - C++ style memory management inside the JVM
- Predictable and dependable execution
  - Bird’s-eye view of what runs and how, and what failed and why

Flink stack



Graph API (Gelly)

Machine Learning

Python API *(upcoming)*

Streaming ML (SAMOA) *(upcoming)*

Apache MRQL

Java & Scala API

Embedded Environment (Java collections)

Flink Optimizer

Flink Stream Builder

Flink Runtime Operators

Local Environment (for debugging)

Remote Environment (Cluster execution)

Apache Tez

Single node execution

Standalone or YARN cluster

Data storage

Files

HDFS

S3

JDBC

HBase

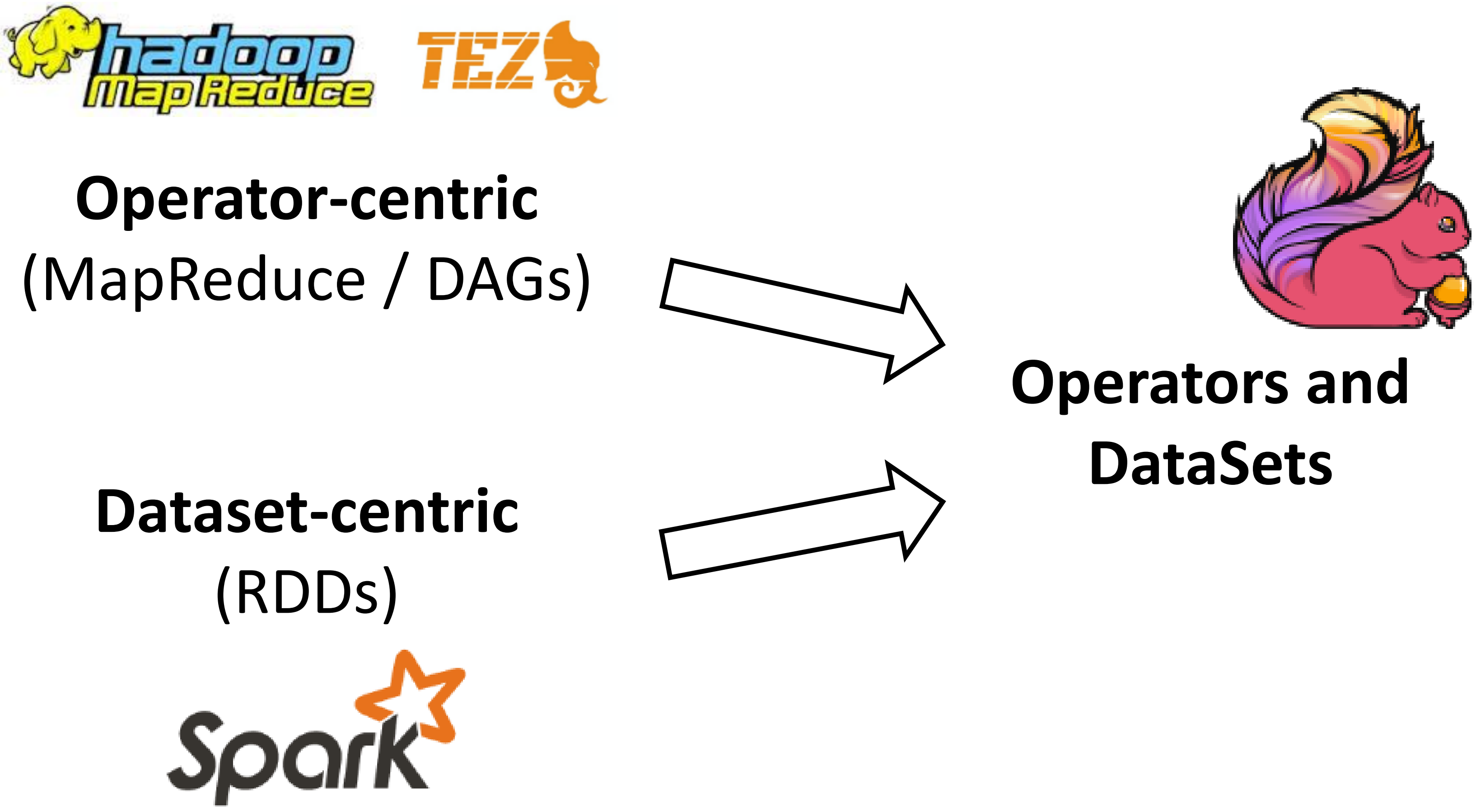
Kafka

Rabbit MQ

Flume

…

Evolution of Architectures



Operator-centric
(MapReduce / DAGs)

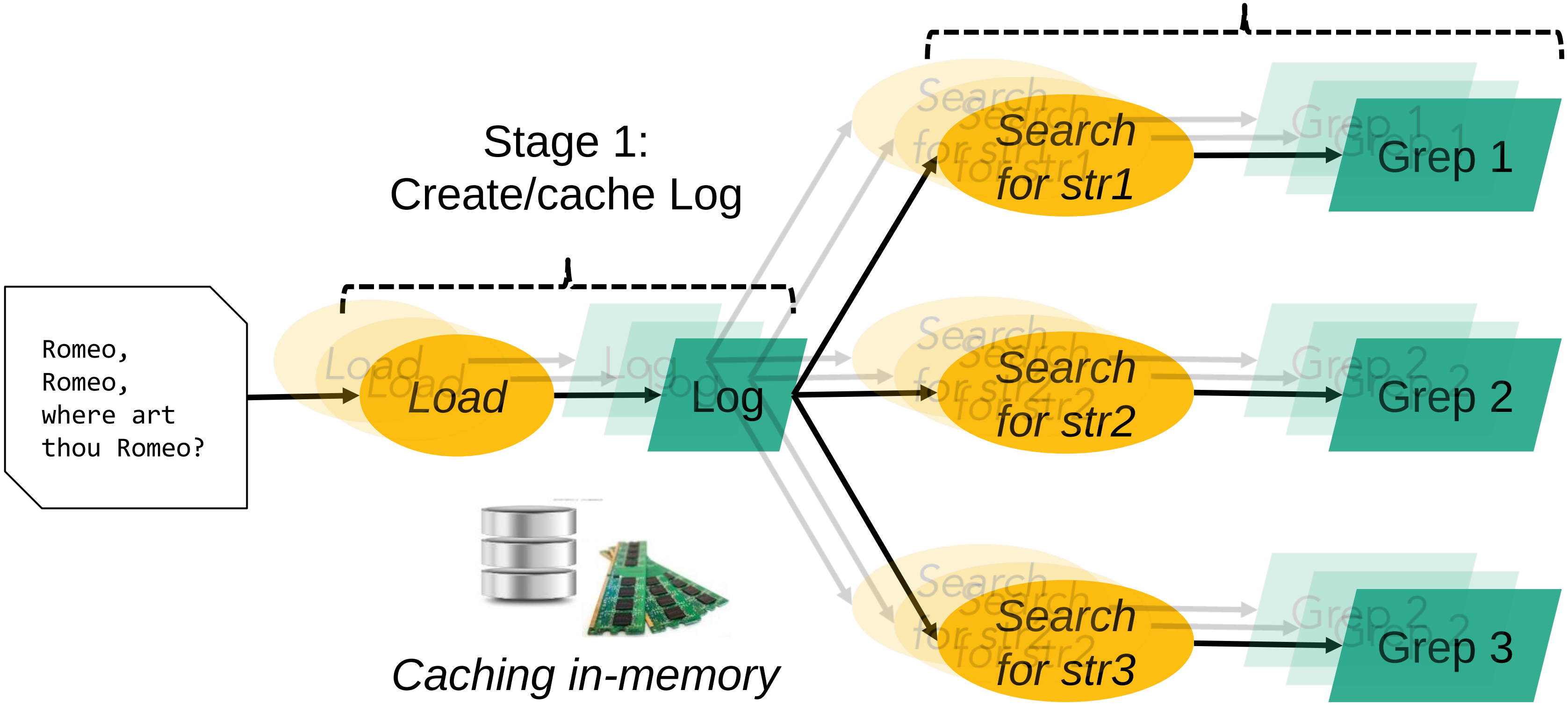
Dataset-centric
(RDDs)

Operators and DataSets

Staged (batch) execution

Subseqent stages:
Grep log for matches

Stage 1:
Create/cache Log



Romeo,
Romeo,
where art
thou Romeo?

Load

Log

Search for str1

Search for str2

Search for str3

Grep 1

Grep 2

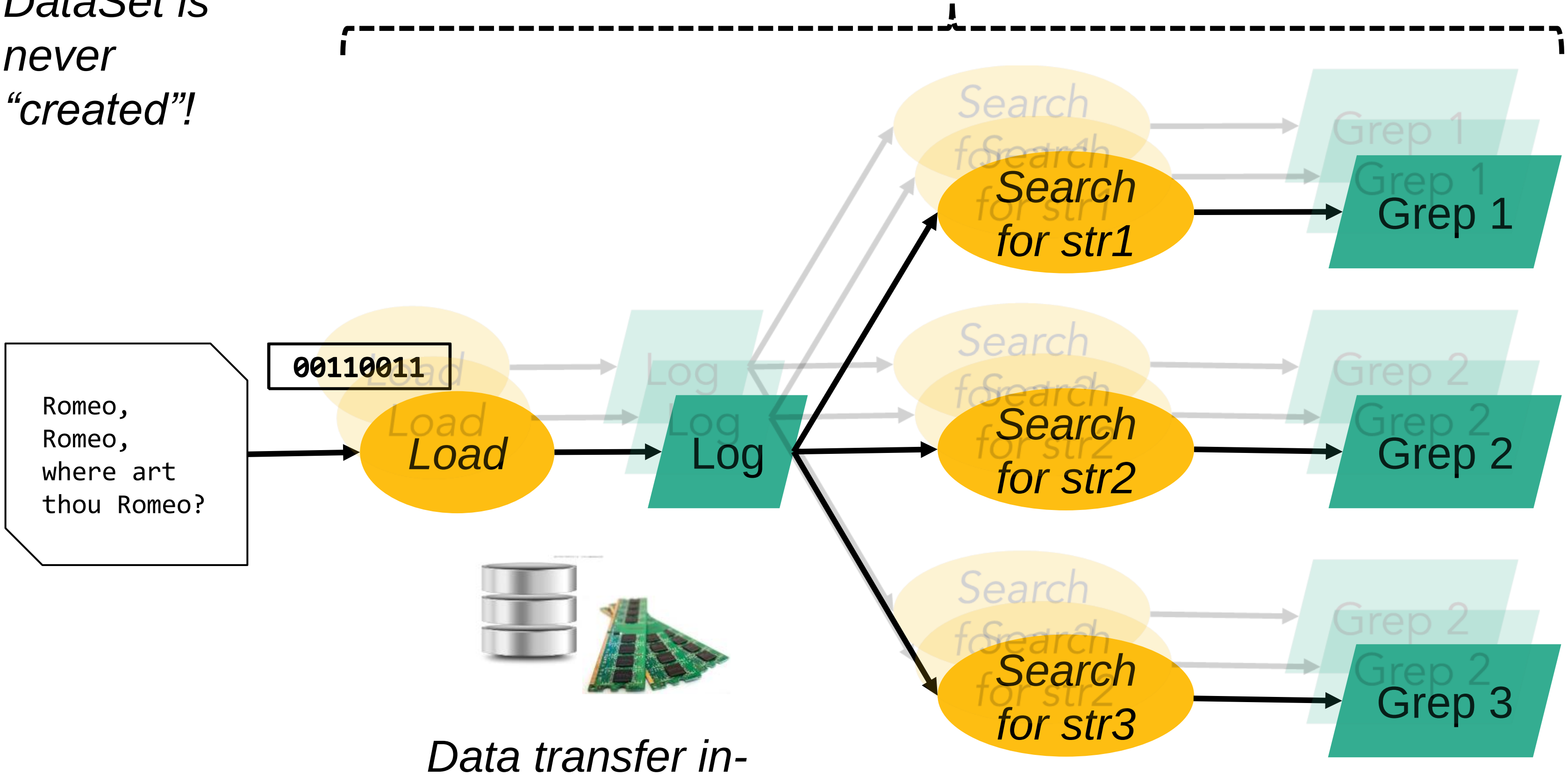
Grep 3

Caching in-memory and disk if needed

Pipelined execution

Note: *Log DataSet is never "created"!*

Stage 1:
Deploy and start operators



Data transfer in-memory and disk if needed

Pipelining in Flink

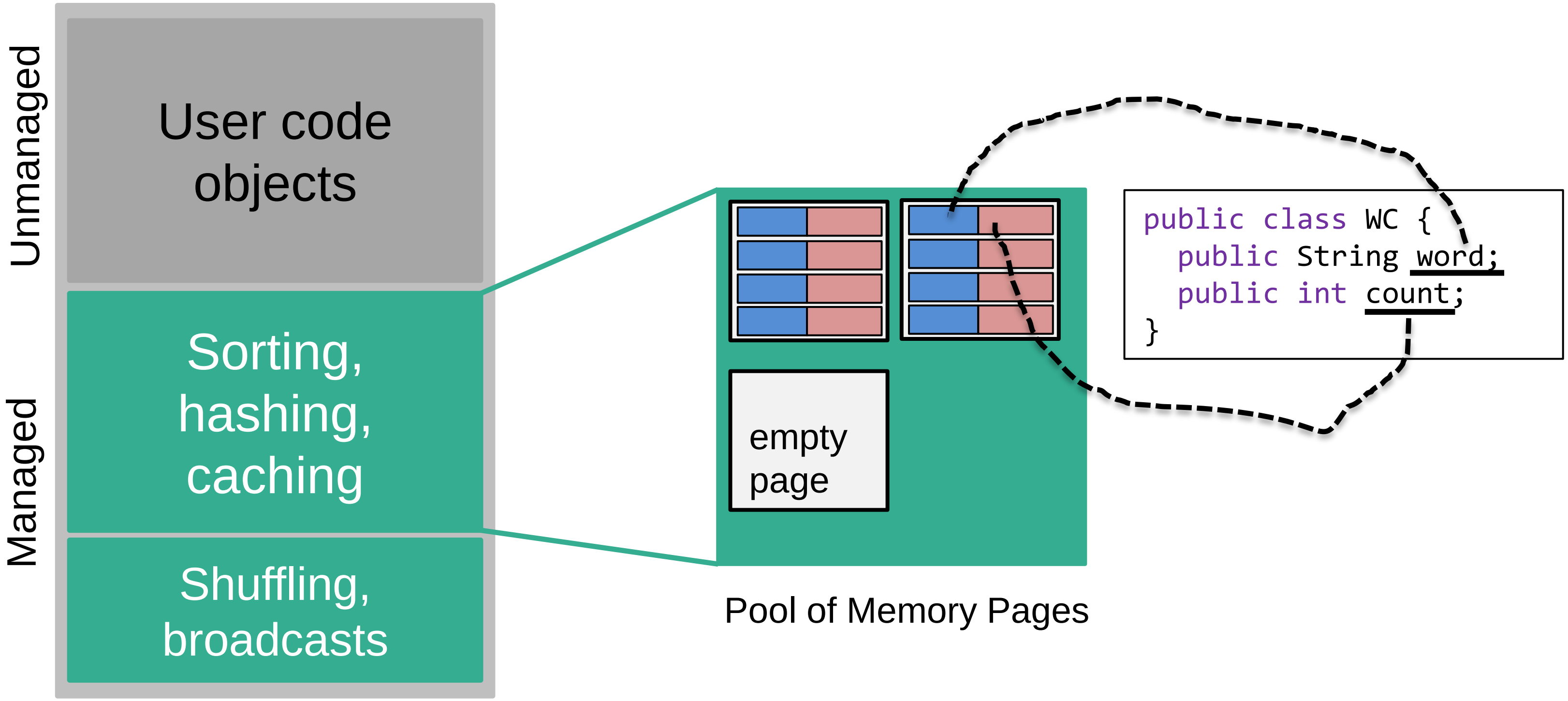
- Currently the default mode of operation
  - Much better performance in many cases – no need to materialize large data sets
  - Supports both batch and real-time streaming

- Currently evolving into a hybrid engine
  - Batch will use combination of blocking and pipelining
  - Streaming will use pipelining
  - Interactive will use blocking

Memory management

Memory management in Flink

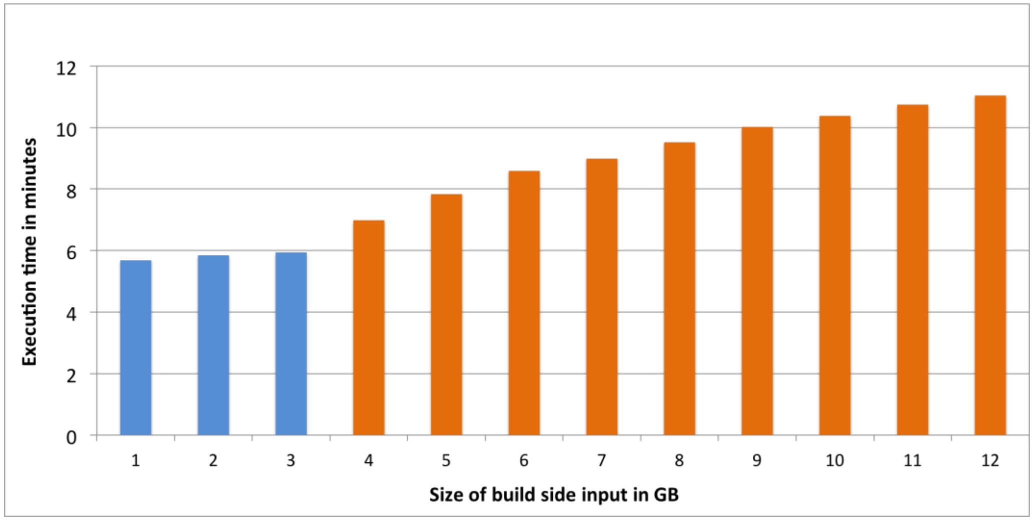
Flink contains its own memory management stack. Memory is allocated, de-allocated, and used strictly using an internal buffer pool implementation. To do that, Flink contains its own type extraction and serialization components.



```
public class WC {
  public String word;
  public int count;
}
```

More at: https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=53741525

Smooth out-of-core performance

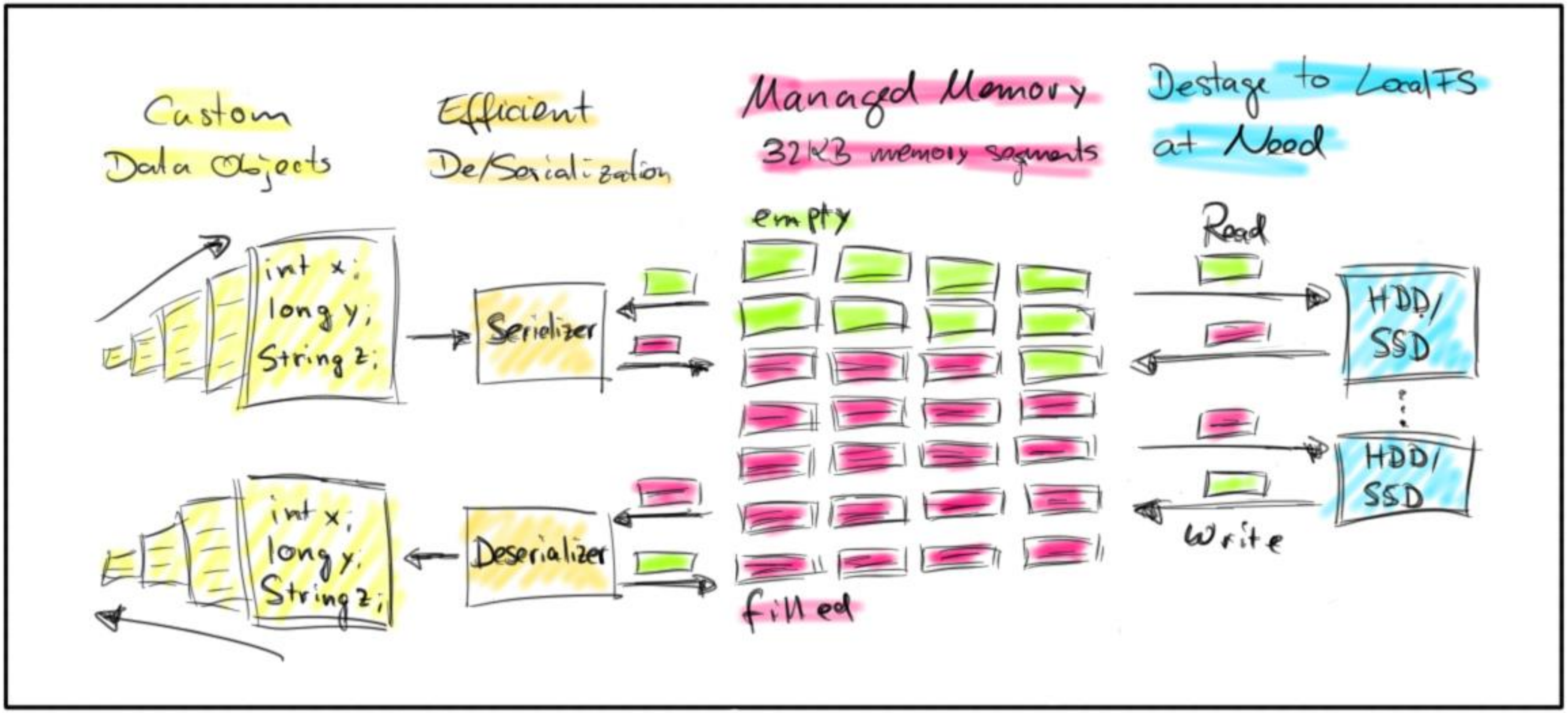


Single-core join of large Java objects beyond memory (4 GB)

Blue bars are in-memory, orange bars (partially) out-of-core

More at: http://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html

Paged Memory Management



Configuring Flink

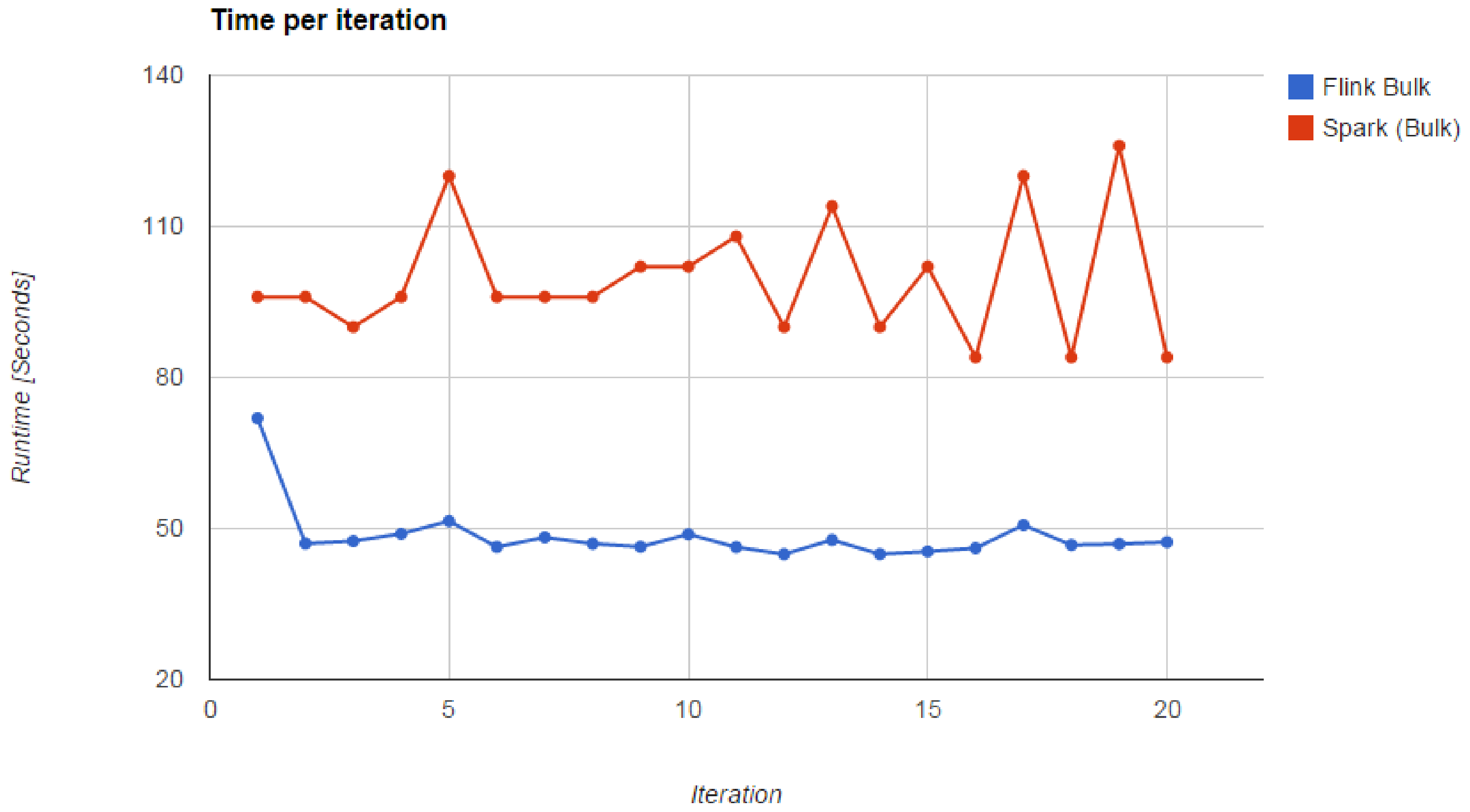
- Per job
  - Parallelism

- System config
  - Total JVM heap size (-Xmx)
  - % of total JVM size for Flink runtime
  - Memory for network buffers (soon not needed)

- That's all you need. System will not throw an OOM exception to you.

Benefits of managed memory

- More reliable and stable performance (less GC effects, easy to go to disk)

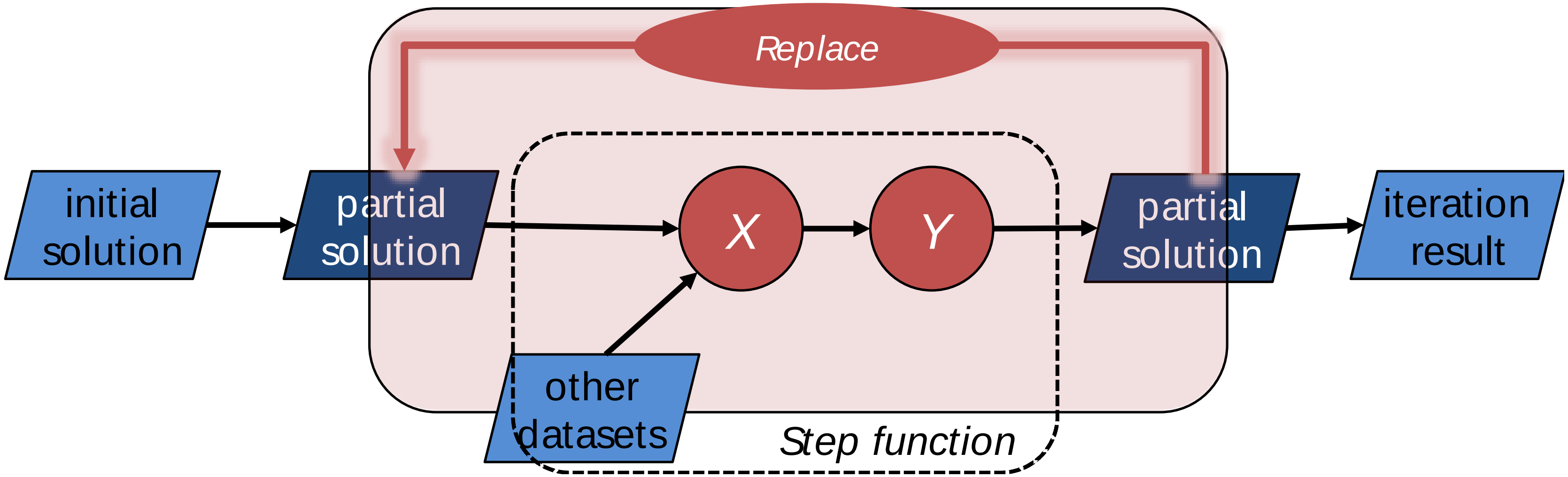


Native iterative processing

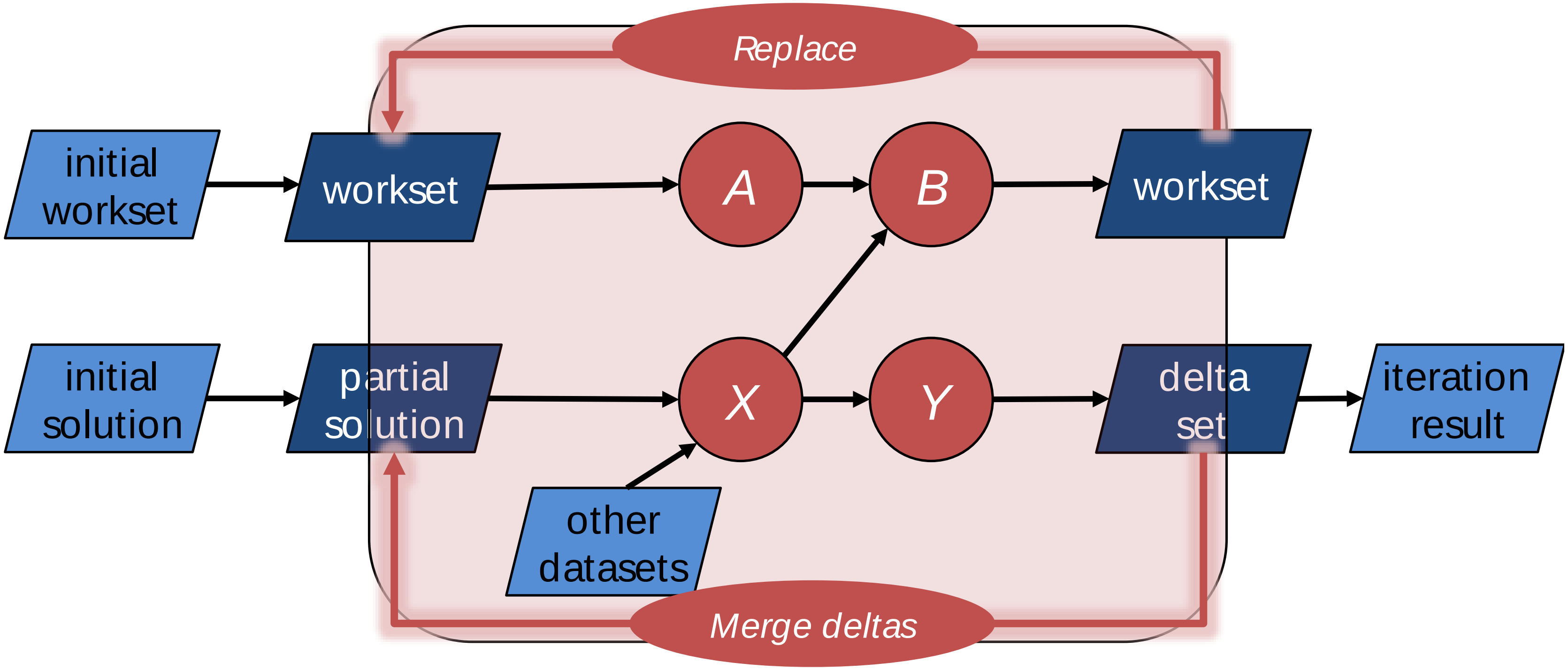
Example: Transitive Closure

```
case class Path (from: Long, to: Long)

val env =
  ExecutionEnvironment.getExecutionEnvironment

val edges = ...

val tc = edges.iterate (10) { paths: DataSet[Path] =>
  val next = paths
    .join(edges).where("to").equalTo("from") {
      (path, edge) => Path(path.from, edge.to)
    }
    .union(paths).distinct()
  next
}

tc.print()
env.execute()
```

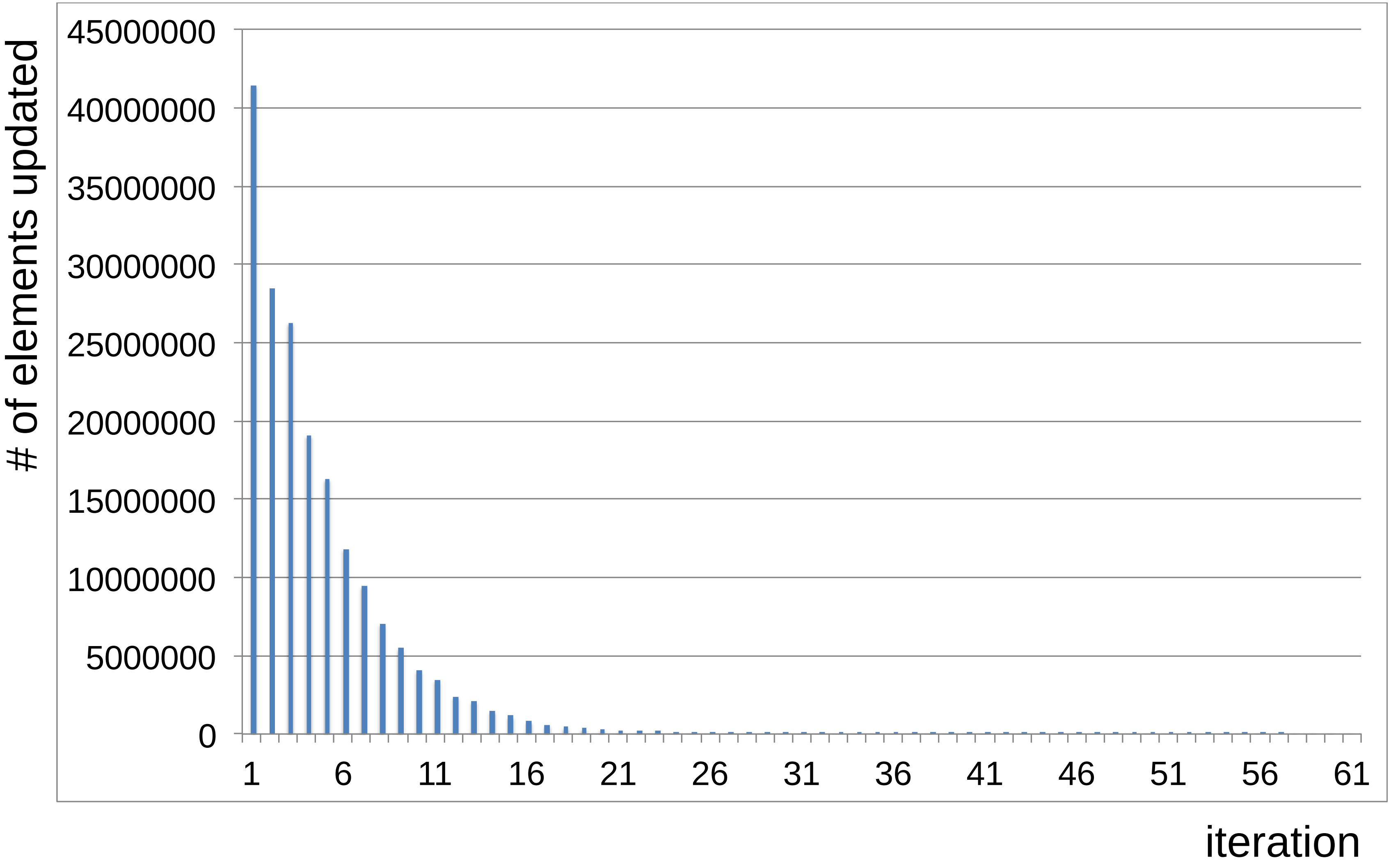
Iterate natively



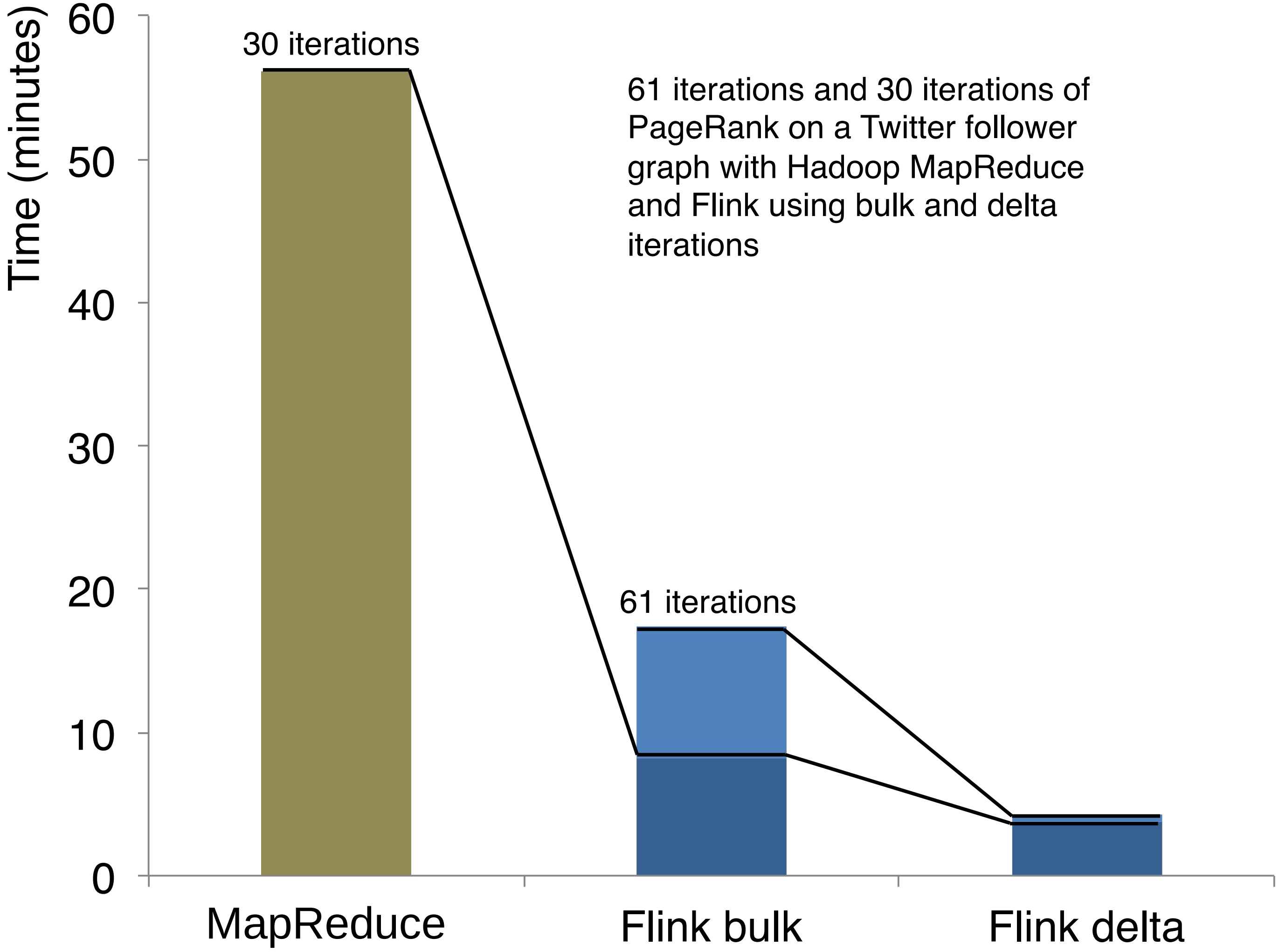
Iterate natively with deltas



Effect of delta iterations



Iteration performance



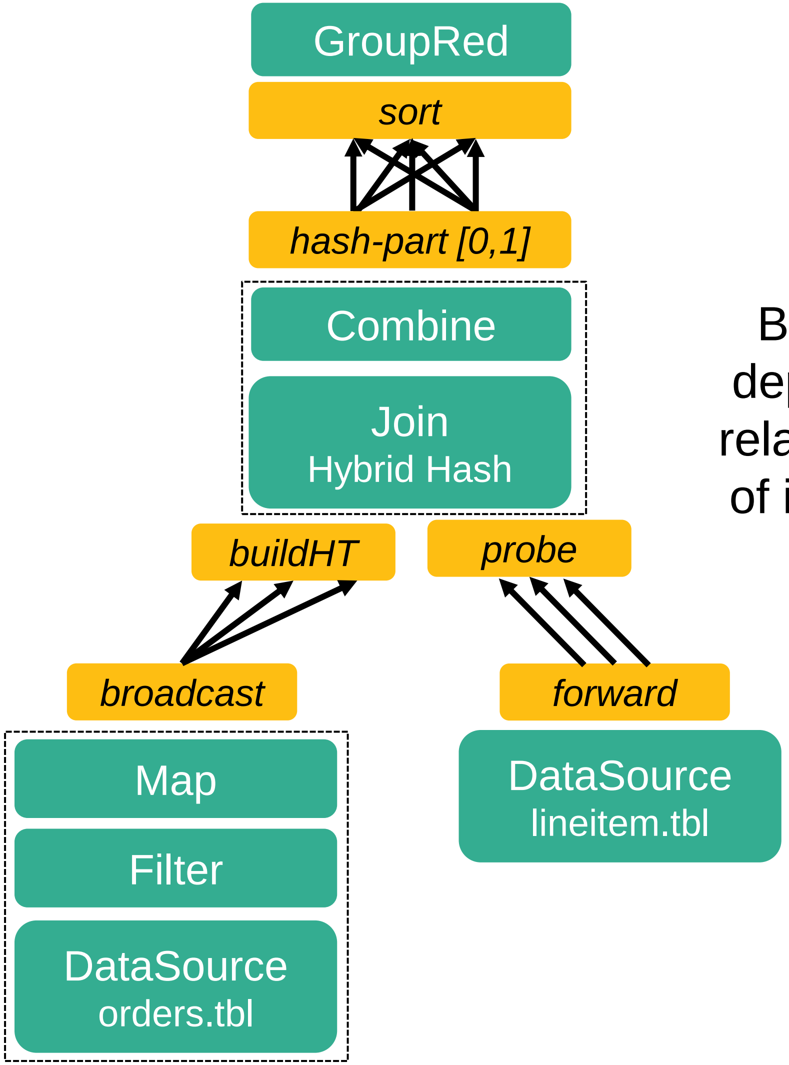
61 iterations and 30 iterations of PageRank on a Twitter follower graph with Hadoop MapReduce and Flink using bulk and delta iterations

Program optimization

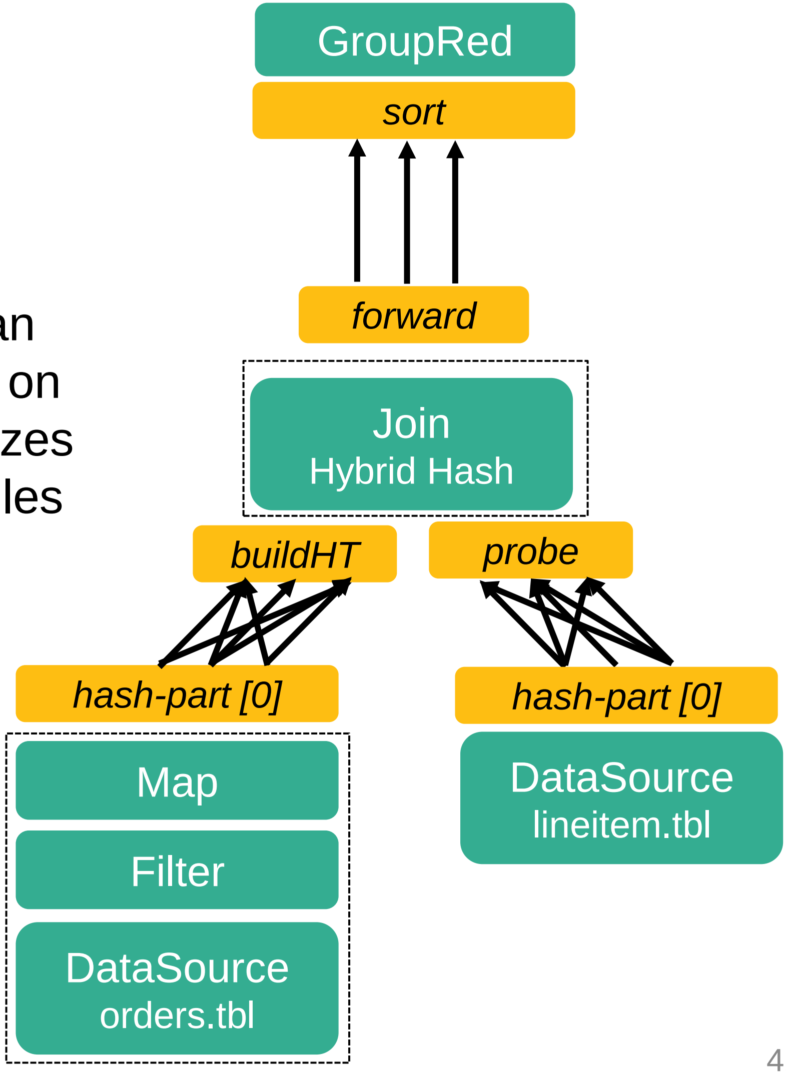
A simple program

```
val orders = …
val lineitems = …

val filteredOrders = orders
  .filter(o => dataFormat.parse(l.shipDate).after(date))
  .filter(o => o.shipPrio > 2)

val lineitemsOfOrders = filteredOrders
  .join(lineitems)
  .where(“orderId”).equalTo(“orderId”)
  .apply((o,l) => new SelectedItem(o.orderDate, l.extdPrice))

val priceSums = lineitemsOfOrders
  .groupBy(“orderDate”).sum(“l.extdPrice”);
```

Two execution plans



Best plan depends on relative sizes of input files



Examples of optimization

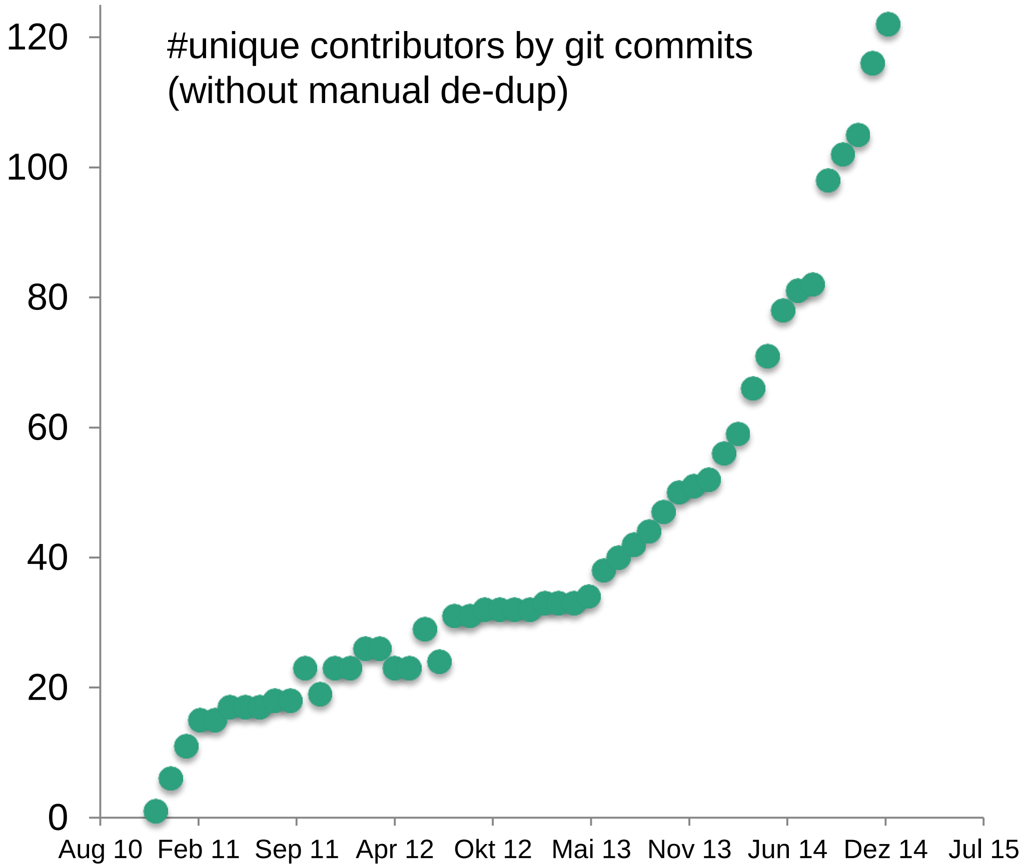
- Task chaining
  - Coalesce map/filter/etc tasks
- Join optimizations
  - Broadcast/partition, build/probe side, hash or sort-merge
- Interesting properties
  - Re-use partitioning and sorting for later operations
- Automatic caching
  - E.g., for iterations

Closing

Flink Roadmap for 2015

- Exactly-once streaming with flexible state
- Support for Google Dataflow
- Batch Machine Learning library
- Streaming Machine Learning with SAMOA
- Graph library additions (more algorithms)
- Interactive programs and Zeppelin
- SQL on top of expression language
- and more…

Flink community



flink.apache.org
@ApacheFlink