Iterative Functional Programming Models: Spark and Flink

Kaustubh Beedkar
kaustubh.beedkar@iitd.ac.in

CSE
IIT Delhi

Problems with MapReduce

- MapReduce provides
  - Automatic parallelization and failure handling
  - Simple programming interface
- Real-world applications often need multiple MR jobs, but **MapReduce does not compose well**
- Major limitation: **programmability**
  - Many MR steps, boilerplate code, spaghetti code
  - Custom code even for common operations
- Major limitation: **performance**
  - MapReduce is heavy on disk
  - No optimization across multiple (related) MR jobs

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)
[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here with our views on MapReduce. This is a good time to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of "jelly beans" rather than utilizing a much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to teach students how to program such clusters using a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce framework.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools DBMS users have come to depend on

Source

Improving MapReduce (1)

Idea 1: Higher-level languages

- Write programs in a suitable for higher-level language
- Programs are "compiled into" MapReduce job(s)
- Addresses (mainly) programmability

Examples

Hive (Facebook)


```
SELECT count(*)
FROM users
```

Pig (Yahoo)

```
A = load 'users';
B = group A all;
C = foreach B generate COUNT(A);
```

Improving MapReduce (2)

Idea 2: From MapReduce to **dataflow engines**

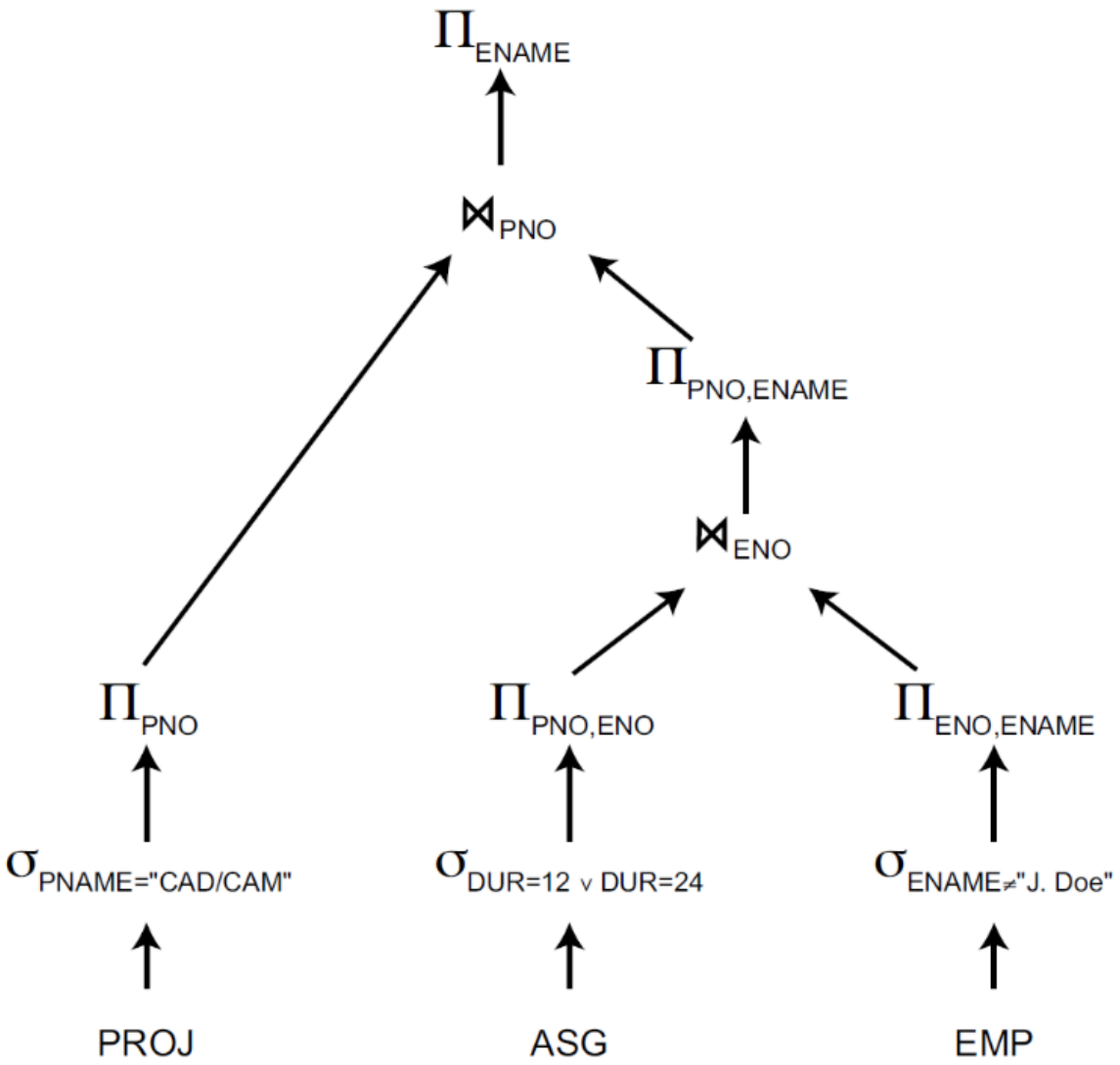
- Instead of compiling to MapReduce, use a more suitable execution environment
- Additionally addresses performance

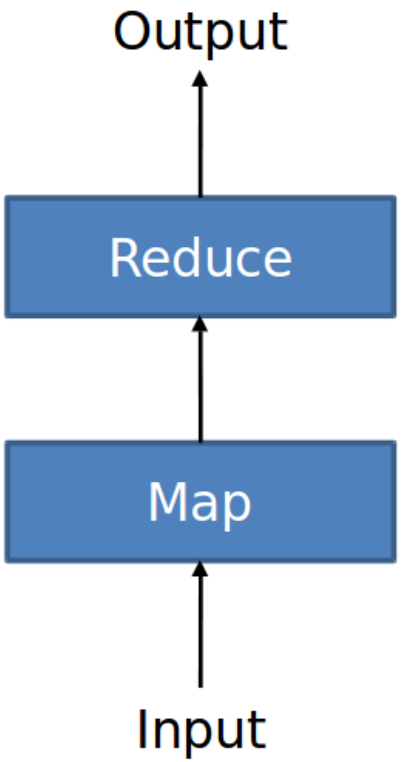
Examples

Dataflow programming

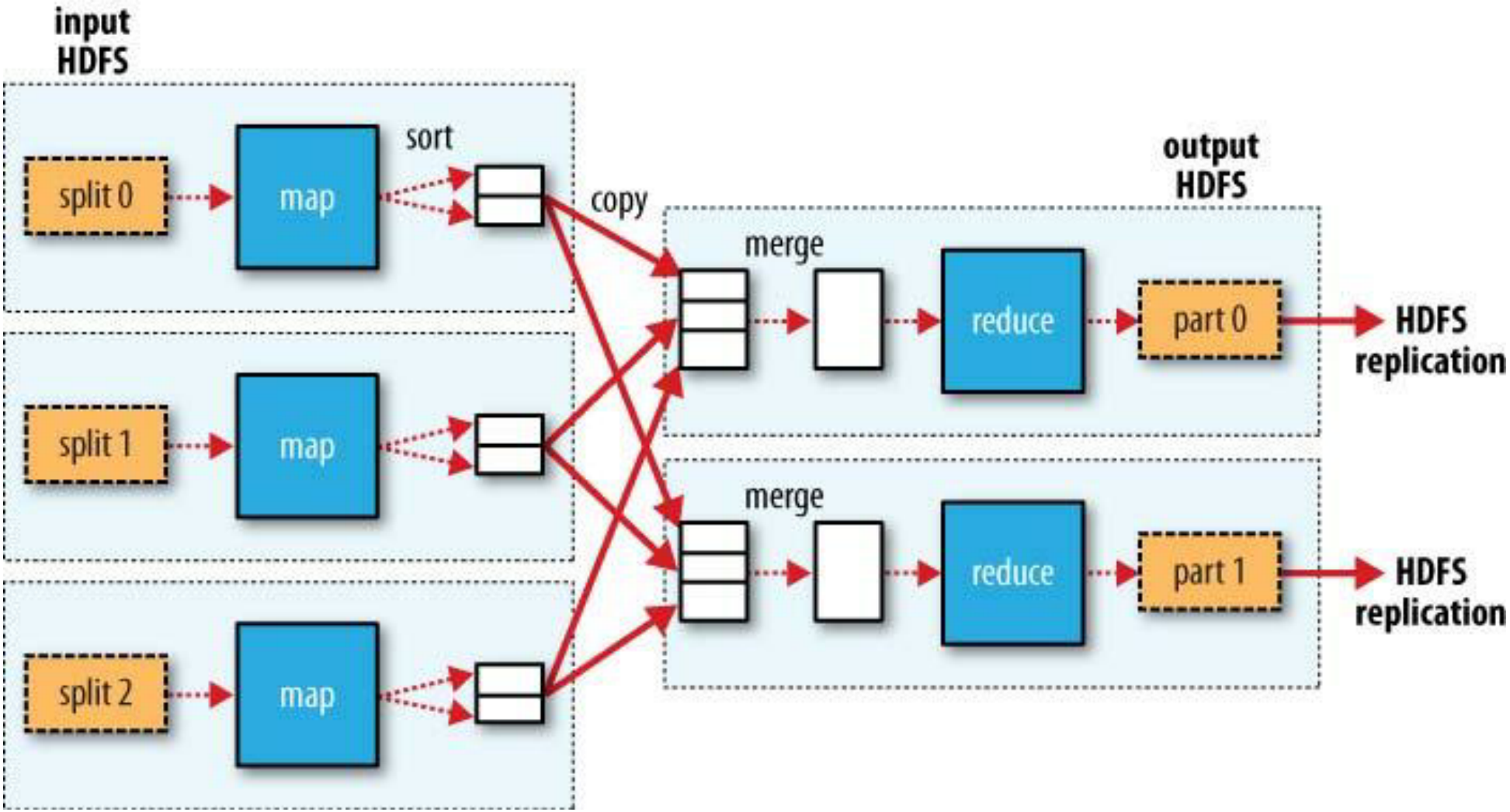
- **Imperative program**
  - Focus on control flow, data at rest
  - "Which command next?"
- **Dataflow program**
  - Focus on data flow, program at rest
  - "Where to put data items next?"
  - Modelled as a directed acyclic graph (DAG)
    - Vertices = operators
    - Edges = inputs and outputs
    - Operators are "black boxes"

Example (logical) dataflows



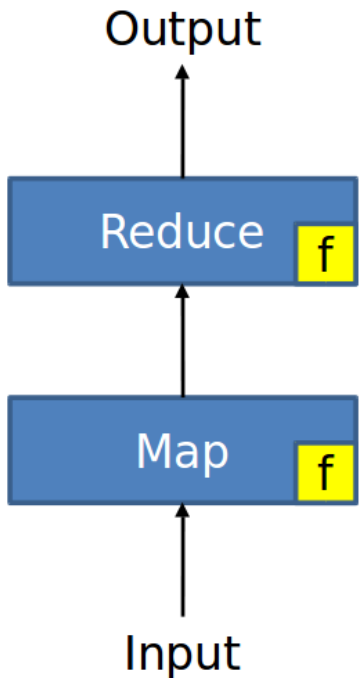


Example (physical) dataflow



MapReduce as dataflow

- Two operators, fixed logical dataflow
  - Key: operators are **parameterized by user defined functions (UDF)**
- Automatic parallelization at runtime
  - Picks and executes one of the multiple possible physical dataflows (#maps, #reduces,..)
- **Dataflow engines** push this idea further
  - Keep UDFs
  - Add more operators
  - Improve implementation
  - Add logical/physical optimizations



Let's summarize

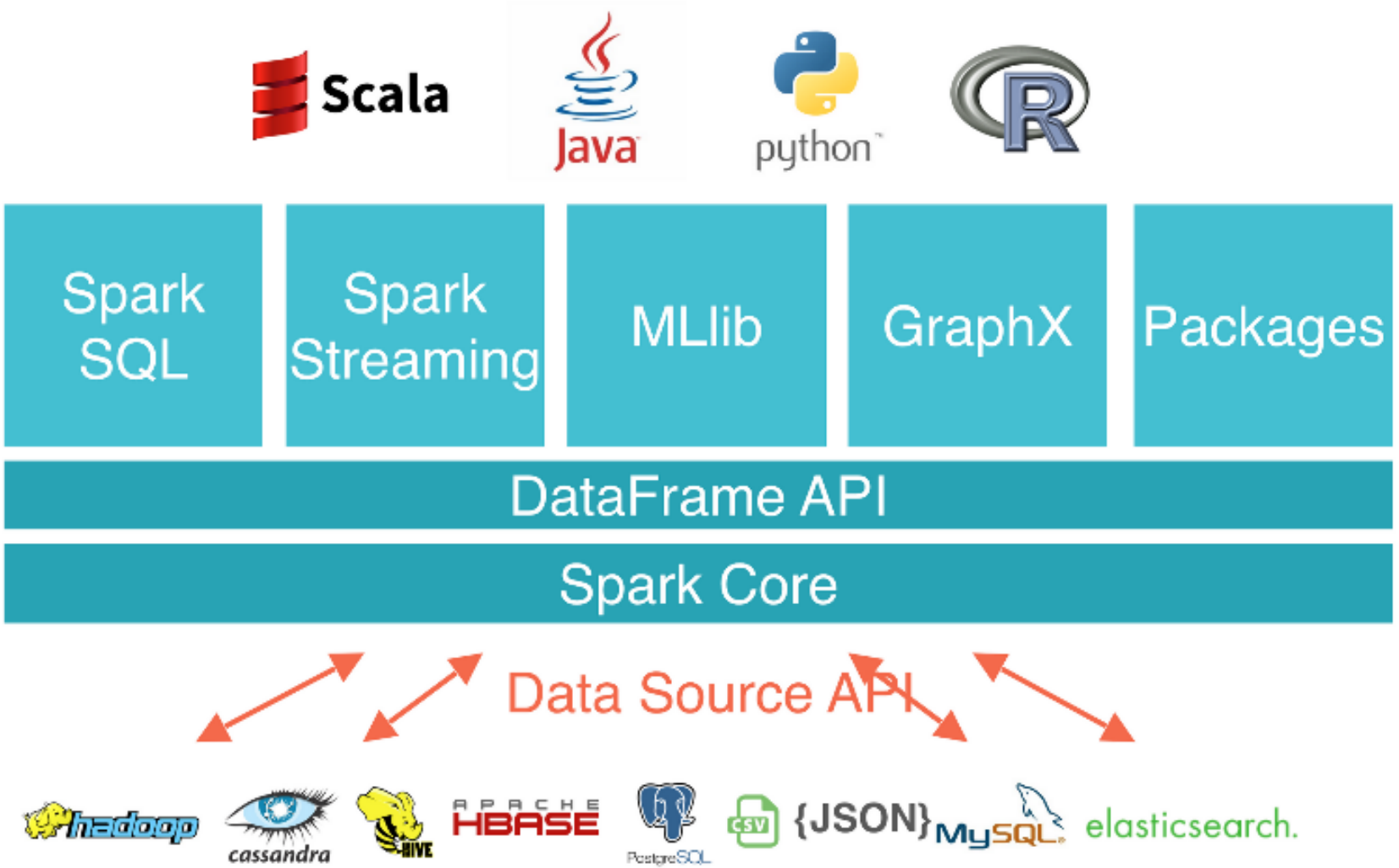
- Programmers provide **logical dataflow**
  - Operators, function parameters, connections
  - Using a suitable higher-level language
  - Note: this is different from SQL
- Dataflow engine creates a **physical dataflow**
  - Includes logical optimizations
  - Includes physical optimizations
- Add **executes on a cluster**
  - Automatic parallelization
  - Automatic failure handling
- In this lecture: Apache Spark & Apache Flink

Outline

Outline

Apache Spark

- A "unified analytics engine for large-scale data processing"
- An open-source Apache project (http://spark.apache.org)



Programmability (WordCount)

```java
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text

    public void reduce(Text key, Iterable<IntWritable> values, Cont
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

>50 lines of MapReduce java code

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

A few lines in Spark's Scala API

Performance (Sorting 100TB)

2013 Record: Hadoop

2100 machines

72 minutes

2014 Record: Spark

207 machines

23 minutes

Also sorted 1PB in 4 hours

sortbenchmark.org, Xin 2015

Core abstraction: RDD

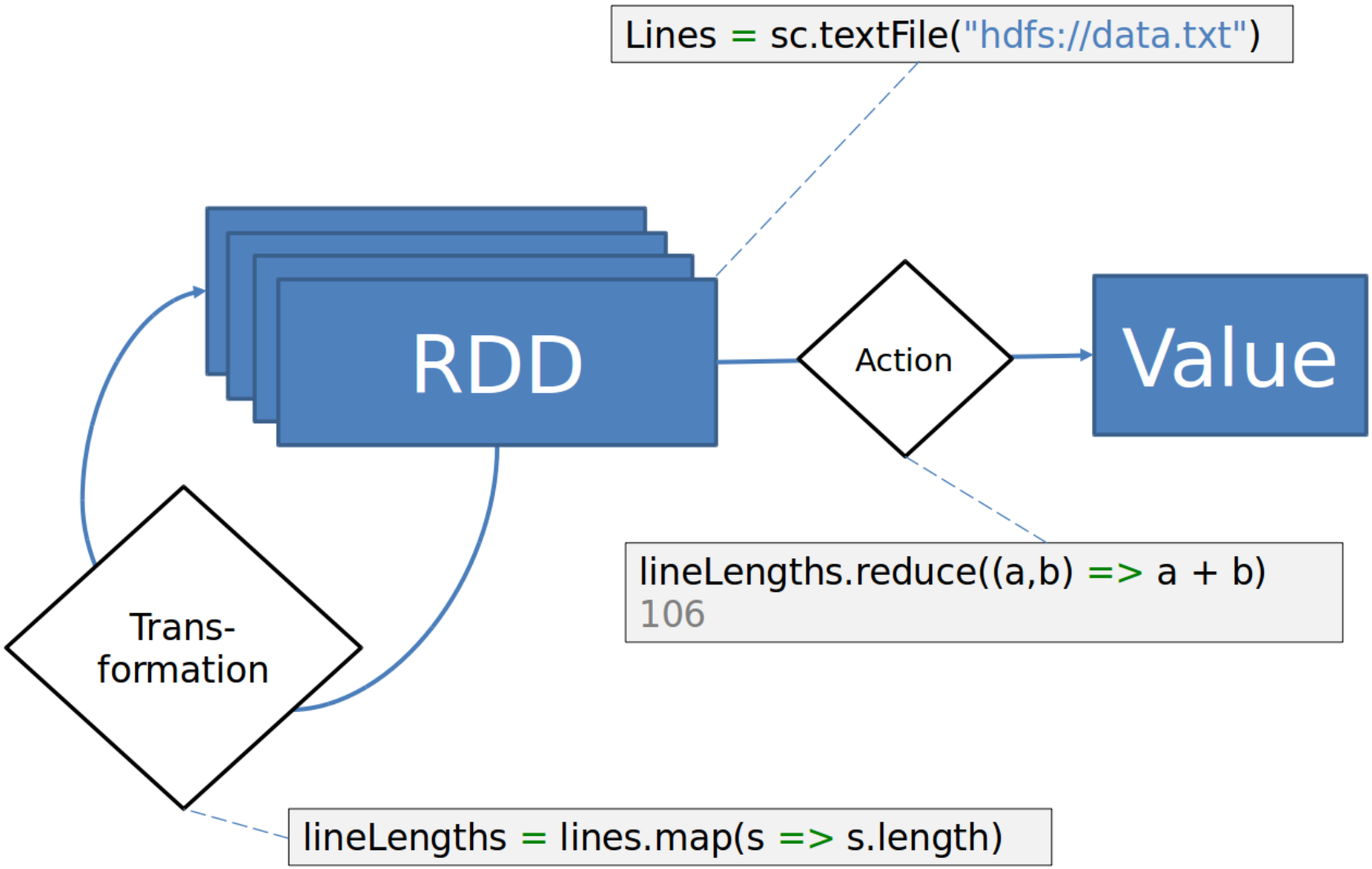
Resilient distributed dataset (RDD)

- Collections of objects (conceptually)
- Spread across a cluster, stored in RAM, on disk, ...
- Built through parallel transformations
- Automatically rebuilt on failure

- Write programs in terms of distributed datasets and operations on them

- Operations
  - **Transformations** (map, filter, groupBy, ...)
  - **Actions** (count, collect, save, ...)

Working with RDDs (1)



Working with RDDs (2)

- **RDDs** are collections of objects (of some type)
  - Immutable, partitioned, fault-tolerant
  - Can be operated on in parallel

- Creating RDDs
  1. **Parallelize** an existing single-node collection

```
val data = Array(1, 2, 3, 4, 5)
val numbers = sc.parallelize(data)
// numbers: RDD[Int] =
// ParallelCollectionRDD [...]
```

  2. **Reference** an external collection (e.g., HDFS file)

```
val lines = sc.textFile("hdfs://data.txt")
// lines: RDD[String] = MapPartitionsRDD [...]
```

  3. **From RDDs**

```
val lineLengths = lines.map(s => s.length)
// lineLengths: RDD[Int] = MapPartitionsRDD[...]
```

Example

- Contents of "data.txt" (2 lines)
  One ring to rule them all, one ring to find them.
  One ring to bring them all and in the darkness bind them.

- ```
  val lines = sc.textFile("hdfs://data.txt")
  // lines: RDD[String]=MapPartitionsRDD[...]
  ```

- ```
  lines.collect
  //res:Array[String] = Array(One ring to rule them all, one ring to find them
  ., One ring to bring them all and in the darkness bind them.)
  ```

Working with RDDs (3)

- **Transformations**
  - Takes an RDD and produces and RDD
  - Computed **lazily** (i.e. not immediately)
  - Example map()

```
val lines = sc.textFile("hdfs://data.txt")
val lineLengths = lines.map(s => s.length)
// lineLengths: RDD[Int] = MapPartitionsRDD[...]
```

  - Discussion of Spark's **map transformation**
    - Parameterized by a UDF $f : T \rightarrow U$
    - Input UDF is a single value (of type T), not a k/v pair
    - UDF outputs exactly one value (of type U), not a collection of k/v pairs (like in functional programming languages)
    - Cf. **flatMap transformation**: one input value; 0, 1, or more output values (which are flattened, like in MapReduce)

Example (ctd...)

- lines . collect

  // res: Array[String] = Array(One ring to rule them all, one ring to find them., One ring to bring them all and in the darkness bind them.)

- val lineLengths = line.map(s =>s.length)

  // lineLengths: RDD[Int] = MapPartitionsRDD [...]

- lineLengths . collect

  // res: Array[Int] = Array(49,57)

Working with RDDs (4)

- **Actions**
  - Takes an RDD and returns a value (to driver program, to external storage, ...)
  - Triggers computation
  - Examples: reduce(), collect()

```
val lines = sc.textFile("hdfs://data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLength.reduce((a,b) => a+b)
// totalLength:Int = 106
```

- Discussion on Spark's **reduce action**
  - Parameterized by UDF $f : (T, T) \rightarrow T$
  - Semantics as in aggregation functions
  - Different from Reduce in MapReduce (no keys, no grouping of values,...)

Simulating MapReduce in Spark

- Simulating Map: use **flatMap** (or **map**) transformation
- Simulating Shuffle
  - Use **groupByKey** transformation $\equiv$ Shuffle
  - Defined on RDDs of k/v pairs (e.g., RDD[String, Int])
  - Collects all values for each key and produces RDD of (key, collection of values)-pairs (e.g., RDD[String, Seq[Int]])
- Simulating Reduce: use **flatMap** (or **map**) transformation
- Often more efficient: **reduceByKey** transformation (similar MR combiner)

WordCount using groupByKey

```
lines.flatMap(_.split(""))
.map((_, 1))
.groupByKey
.map(p => (p._1, p._2.sum))
```

WordCount using reduceByKey

```
lines.flatMap(_.split(""))
  .map((_, 1))
  .reduceByKey(_ + _)
```

Example (1)

- lines . collect

  res: Array[String] = Array(One ring to rule them all, one ring to find them., One ring to bring them all and in the darkness bind them.)

- lines .flatMap(_. split (" ")).map((_, 1)). collect

  res40: Array[(String, Int)] = Array((One,1), (ring,1), (to,1), (rule,1), (them,1), (all,,1), (one,1), (ring,1), (to,1), (find,1), (them.,1), (One,1), (ring,1), (to,1), (bring,1), (them,1), (all,1), (and,1), (in,1), (the,1), (darkness,1), (bind,1), (them.,1))

Example (2)

- lines .flatMap(_. split (" ")) .map((_, 1)) .groupByKey.collect

  res: Array[(String, Iterable[Int])] = Array((rule,CompactBuffer(1)), (bring,CompactBuffer(1)), (One,CompactBuffer(1, 1)), (one,CompactBuffer(1)), (them.,CompactBuffer(1, 1)), (ring,CompactBuffer(1, 1, 1)), (them,CompactBuffer(1, 1)), (find,CompactBuffer(1)), (bind,CompactBuffer(1)), (all,,CompactBuffer(1)), (all,CompactBuffer(1)), (to,CompactBuffer(1, 1, 1)), (in,CompactBuffer(1)), (darkness,CompactBuffer(1)), (and,CompactBuffer(1)), (the,CompactBuffer(1))

Example (3)

- ```
  lines.flatMap(_.split(" ")).map((_, 1))
    .groupByKey
    .map(p => (p._1, p._2.sum)).collect
  ```

  res: Array[(String, Int)] = Array((rule,1), (bring,1), (One,2), (one,1), (them.,2), (ring,3), (them,2), (find,1), (bind,1), (all,,1), (all,1), (to,3), (in,1), (darkness,1), (and,1), (the,1))

- ```
  lines.flatMap(_.split(" ")).map((_, 1))
    .reduceByKey(_ + _).collect
  ```

  [same result]

Spark's API

Transformations

map
flatMap
filter
sample
union
intersection
substract
distinct
groupBy
groupByKey
sortByKey

treeReduce
join
leftOuterJoin
rightOuterJoin
cogroup
Cartesian
zip
pipe
repartition
foreach
cache
persist
unpersist

Actions

reduce
fold
aggregate
foreach
count
countByValue
collect
first
take
takeOrdered
saveAs[...]
...

Language support

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains("error");
  }
}).count();
```

Standalone Programs

Python, Scala, & Java

Interactive Shells

Python & Scala

Performance

Java & Scala are faster due to static typing

…but Python is often fine

And more ...

- **DataFrames**
  - Think: RDD with schema
  - Common, important case
  - Special DSL; e.g., data.groupBy("dept").avg("age")
  - Can push some operations to data sources (e.g., DBMS)
- **SparkSQL** for running SQL queries
- **Streaming** for processing real-time data
- **SparkX** for graph processing
- **MLlib** for machine learning
- Components can be composed with each other

Outline

Spark as a dataflow engine

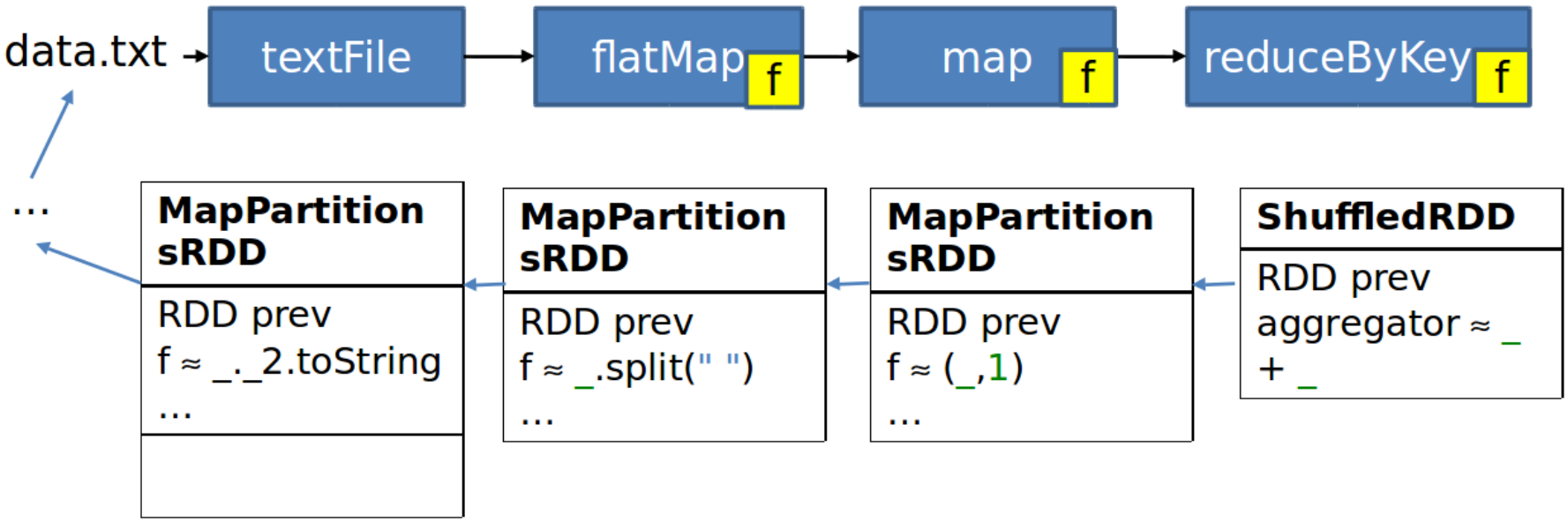
- Spark has RDDs and operators

- We can view it as a dataflow engine
  - RDDs correspond to logical dataflows
  - Transformations combine/add operators to dataflows
  - Actions compile and execute a dataflow

- An RDD is not really a collection of values
  - Instead: A dataflow that produces a collection of values when executed

Example (1)

- `sc.textFile("hdfs://data.txt")`
  `.flatMap(_.split(" ")).map((_,1))`
  `.reduceByKey(_+_)`
- Recall: transformations are computed lazily
  - Spark builds internal representation of dataflow



Example (2)

- Spark also allows you to retrieve the logical dataflow
- ```
  sc.textFile("hdfs://data.txt")
    .flatMap(_.split(" "))).map((_,1))
    .reduceByKey(_+_)
    .toDebugString
  ```

  res: String = (2) ShuffledRDD[91] at reduceByKey at <console>:28 [] +-(2) MapPartitionsRDD[90] at map at <console>:28 [] — MapPartitionsRDD[89] at flatMap at <console>:28 [] — hdfs:///data.txt MapPartitionsRDD[88] at textFile at <console>:28 [] — hdfs:///data.txt HadoopRDD[87] at textFile at <console>:28 []
  ```

An RDD is an interface

- Abstract class in Scala (RDD[T])

- Consists of
  1. List of **partitions** (= splits in Hadoop)
  2. List of **dependencies** on parent RDDs
  3. Function to compute a partition from its parents

  } lineage

  4. (Optional) a **partitioner** (hash, range)
  5. (Optional) **preferred locations** for each partition

  } optimized execution

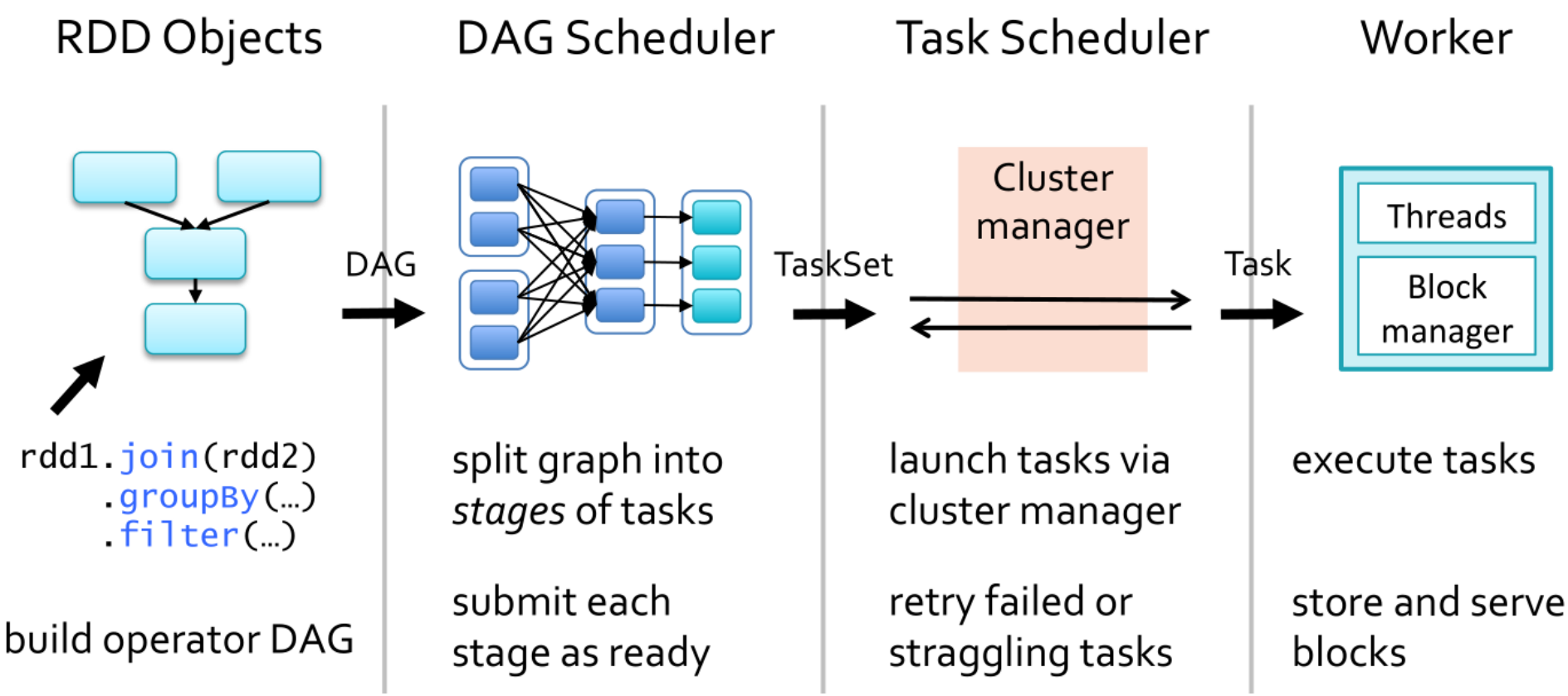
- This information allows Spark to build a suitable parallel execution plan

Example: FilteredRDD

- Corresponds to filter transformations
  - Takes an RDD and filters it using a predicate

1. Partitions: same as parent RDD
2. Dependencies: "one-to-one" on parent
3. Compute(partition): compute parent and filter it
4. PreferredLocations(partition): none (ask parent)
5. Partitioner = none

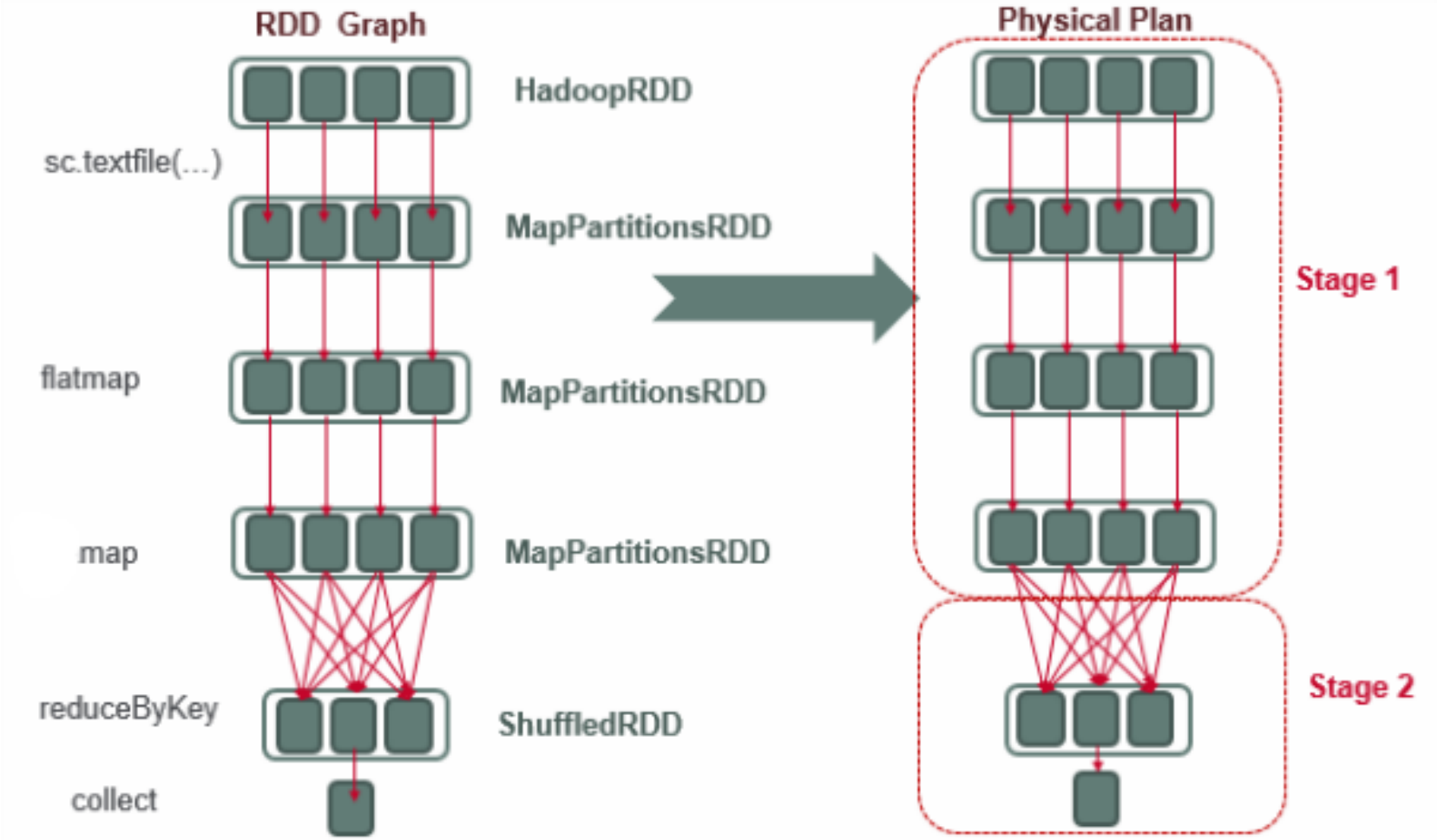
Execution process



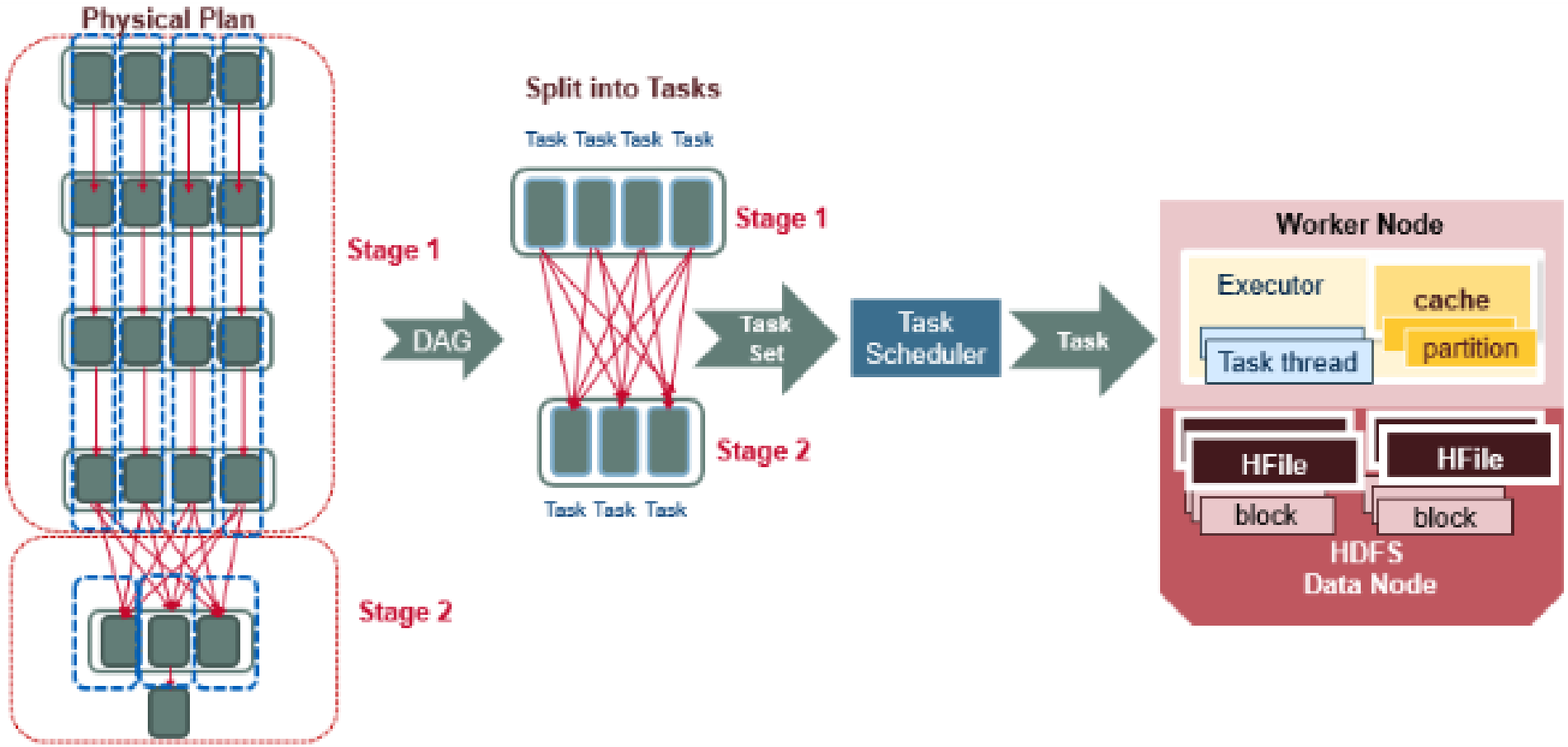
Execution plans

- When using an action, Spark translates logical representation into a physical execution plan
- Multiple operations can me merged into **tasks**
  - Each tasks fetches its input performs some function, and produces output
- Tasks are grouped into **stages**
  - Tasks within a stage can be computed without data movement
  - Between stages, data movement is (often) necessary
- Communication between tasks: pipelining
- Communication between stages: network
  - Data generally buffered in memory
  - If too big, disk also used
  - Cf. Hadoop MapReduce: always use disk

Execution plans (example)

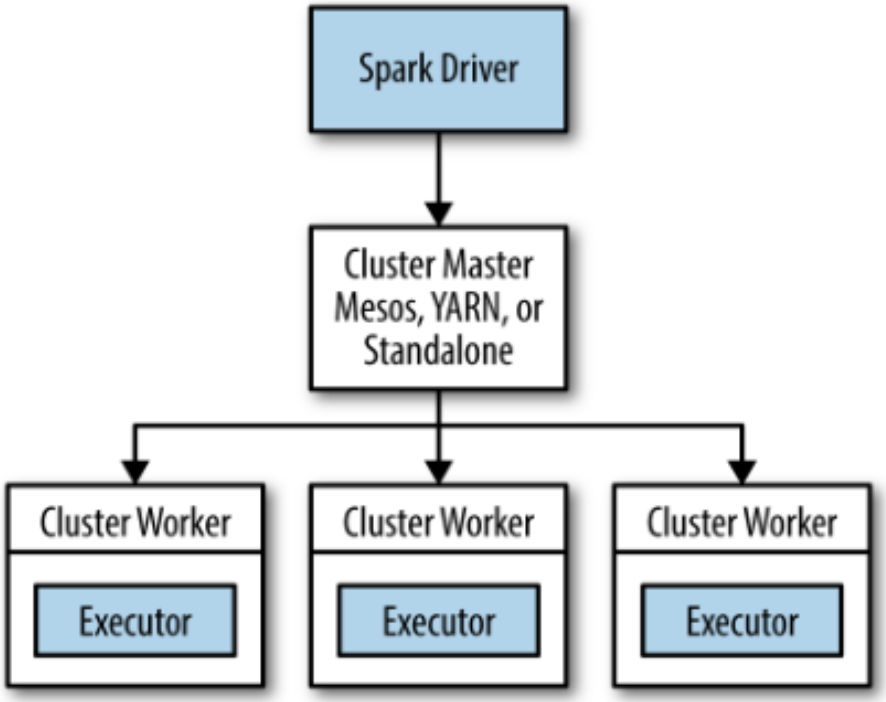


Execution plans (example cond...)

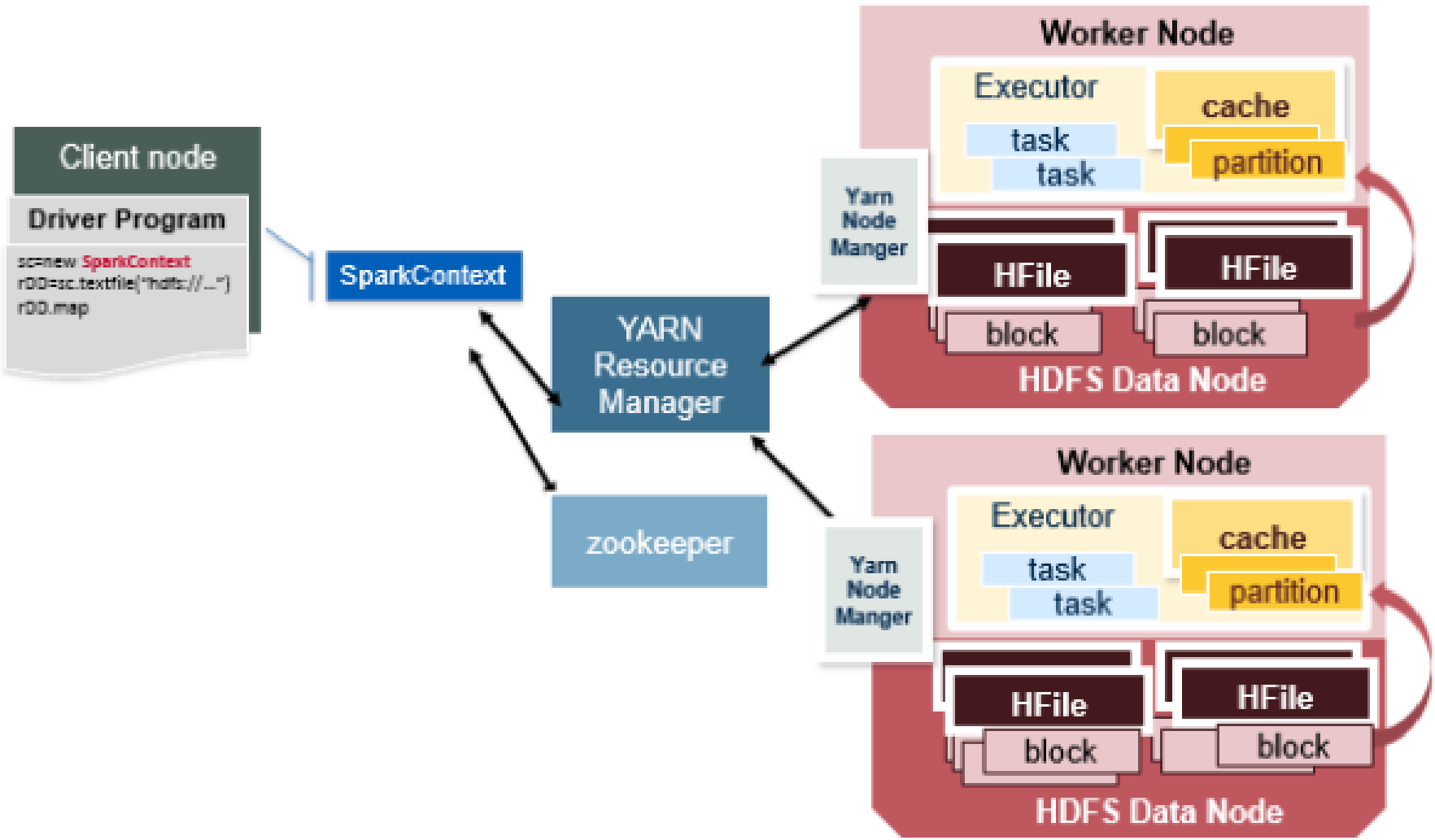


Spark's runtime architecture

- **Driver** submits job
- **Cluster master** manages cluster workers and launches executors
- **Cluster workers** are machines that do the work
- **Executors** are containers that run tasks
- A task can run in parallel or on multiple executors (one per partition)



Spark Application on Hadoop Cluster



Spark's parallel processing

- Sorts using *parallel range partitioning sort*
- By default, join transformation performs a *parallel hash join*
- groupByKey *repartitions* data based on grouping key (default: hash; can override)
- reduceByKey additionally uses *pre-aggregation*
- Spark understands how data is partitioned
  - Can do joins locally if data *co-partitioned* on input
  - Can avoid shuffling data if data is already partitioned on key

Reusing computation (1)

- What does the following code do?

```
val counts = sc.textFile("hdfs://data.txt")
  .flatMap(_.split(" "))).map((_, 1))
  .reduceByKey(_ + _)

counts.collect

counts.collect
```

Reusing computation (2)

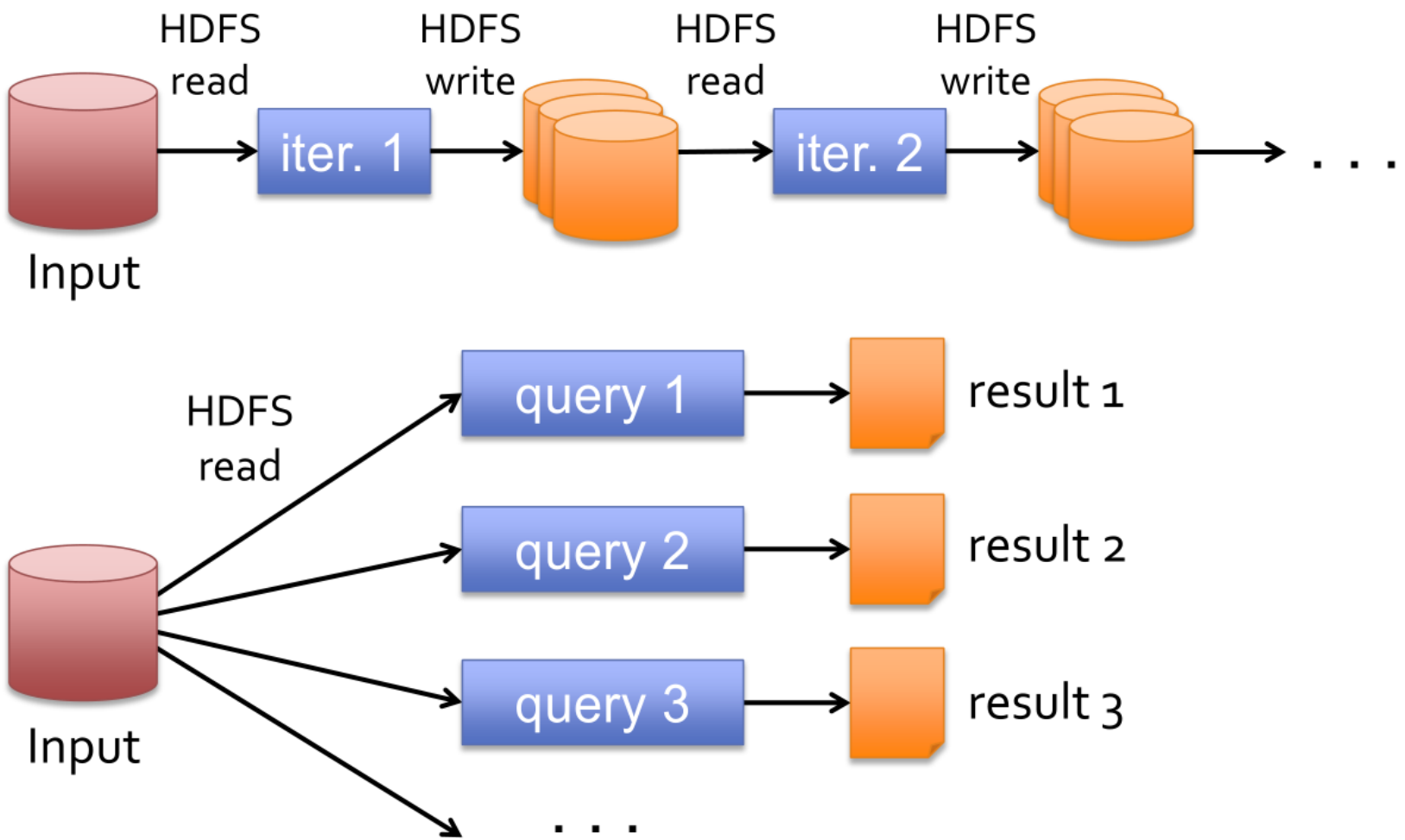
- Intermediate results can be cached
  - As objects in memory, as binary representations in memory, on disk, in memory and on disk,...
- Simple case: cache() keeps objects in memory

```
val counts = sc.textFile("hdfs://data.txt")
 .flatMap(_.split(" "))).map((_, 1))
 .reduceByKey(_ + _).cache

counts.collect // runs jobs, caches results

counts.collect // reuses cached results
```

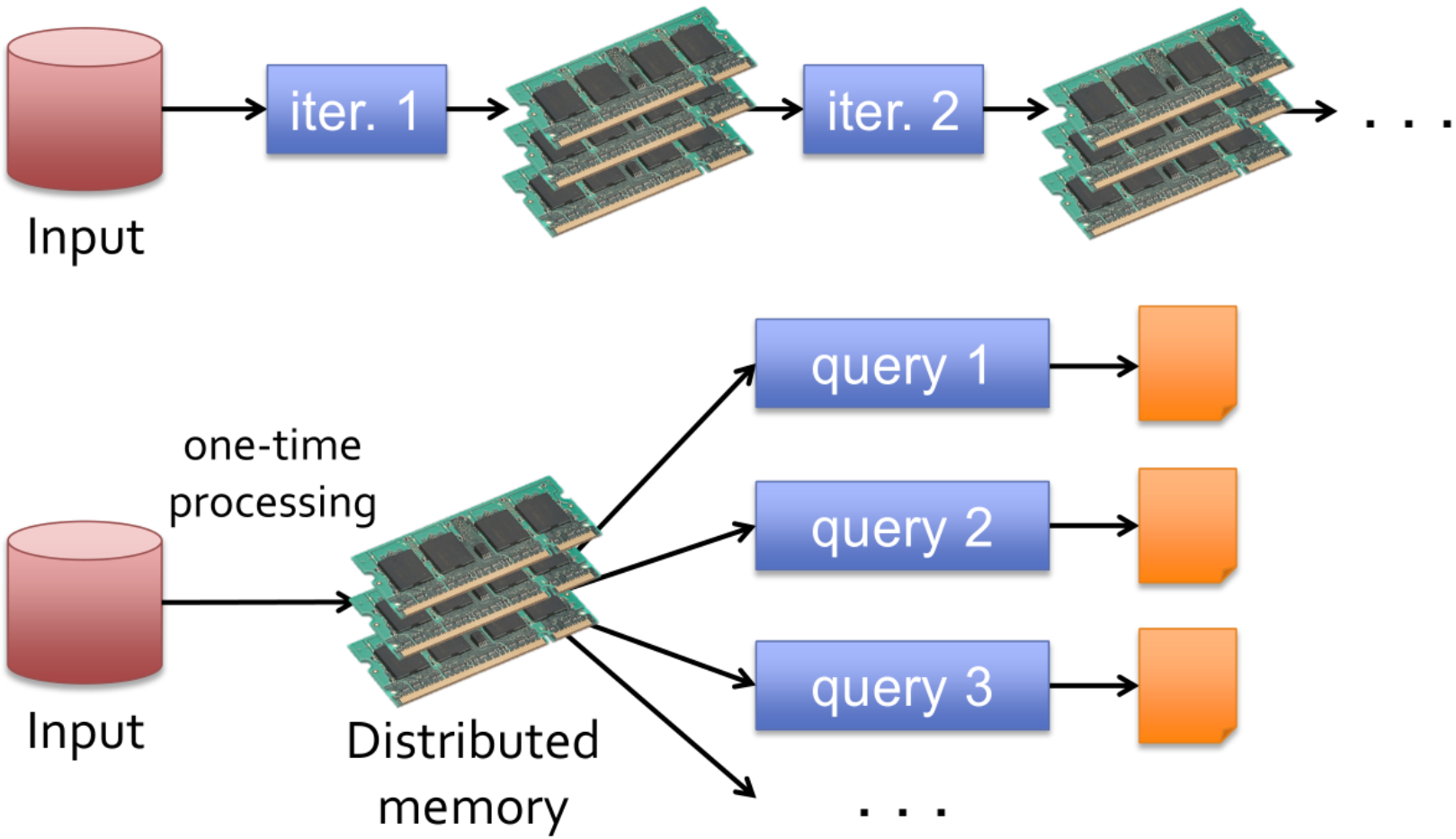
Data sharing in MapReduce



Slow due to replication, serialization, and disk IO

event.cwi.nl/lsde

Data sharing in Spark



~10× faster than network and disk

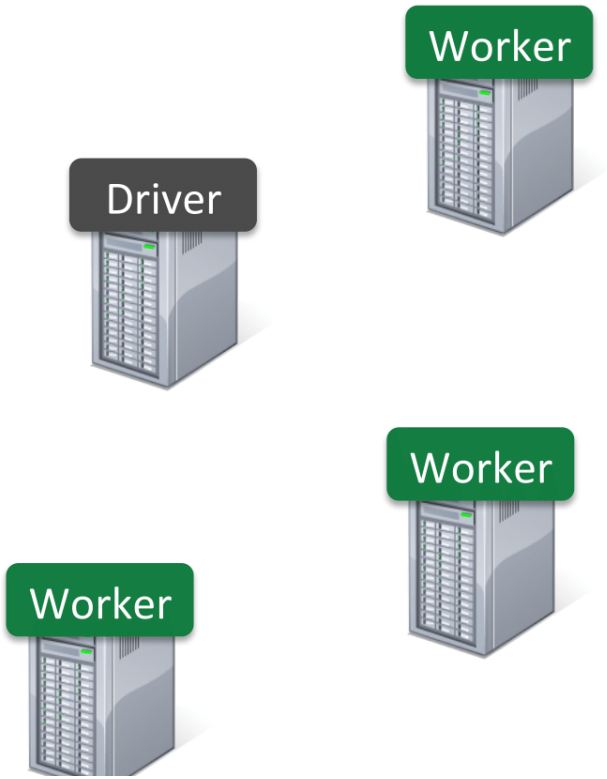
event.cwi.nl/lsde

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Example: Log Mining

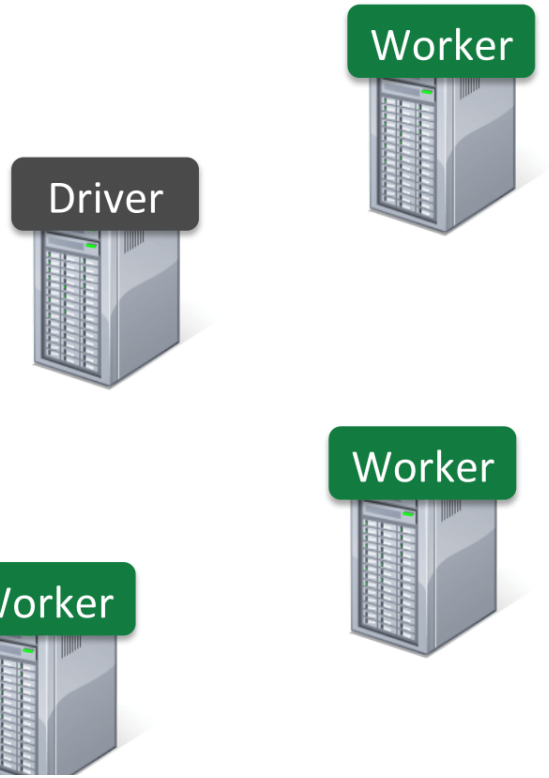
Load error messages from a log into memory, then interactively search for various patterns



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
```

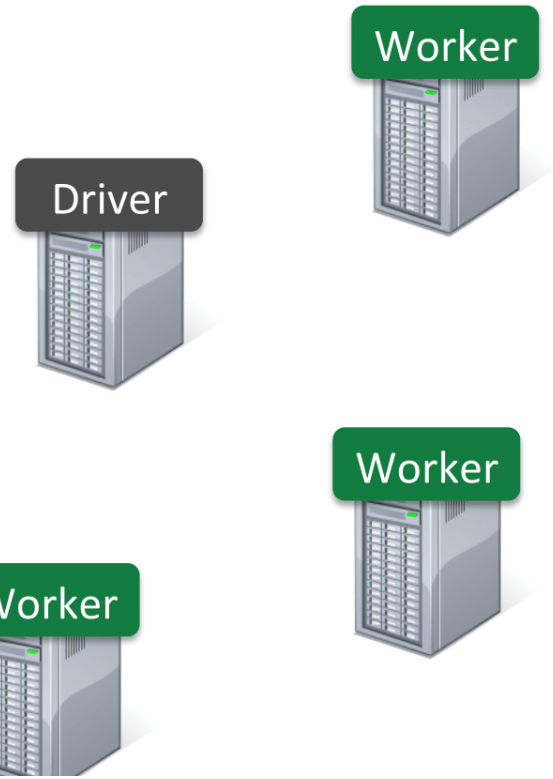


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Base RDD

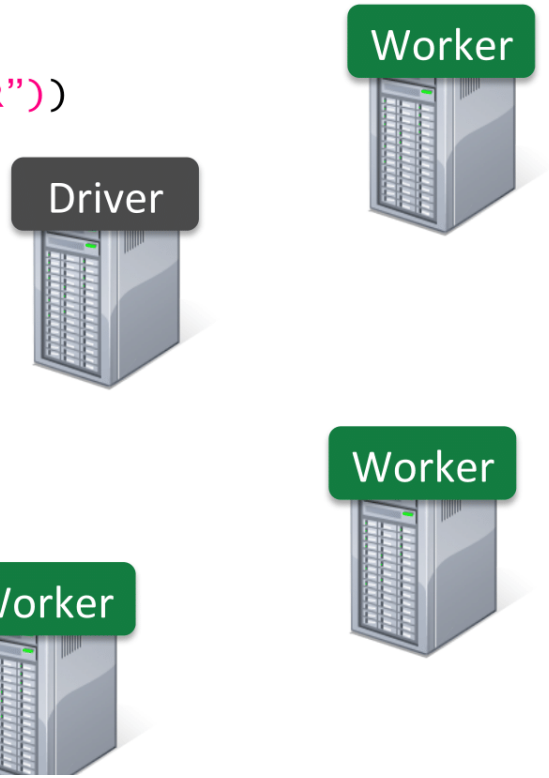
```
lines = spark.textFile("hdfs://...")
```



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```

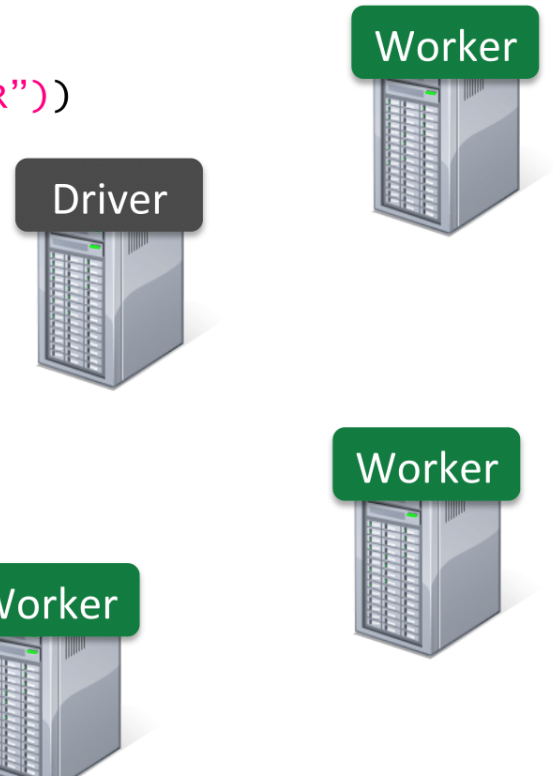


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Transformed RDD

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```

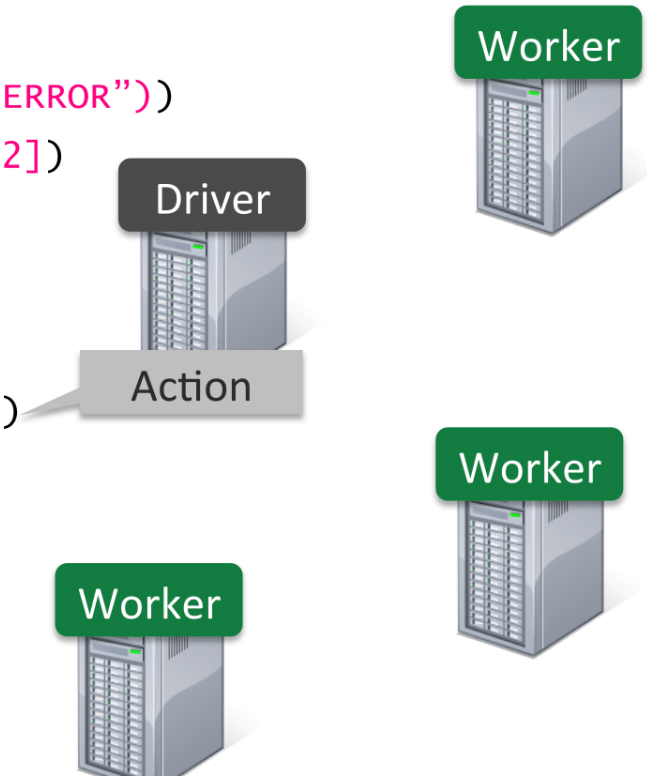


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```

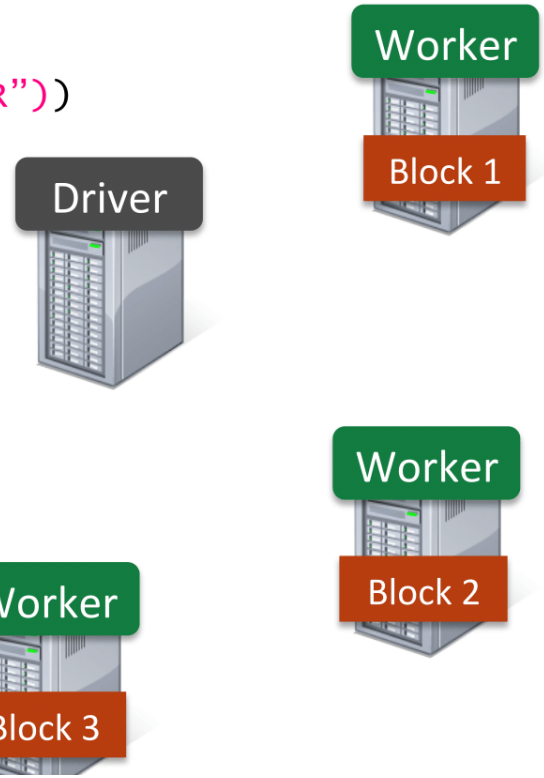


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
```

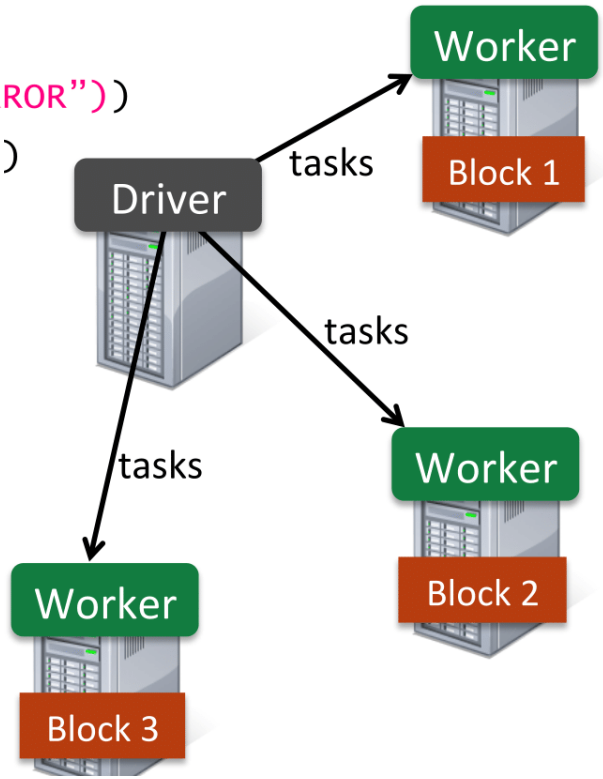


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
```

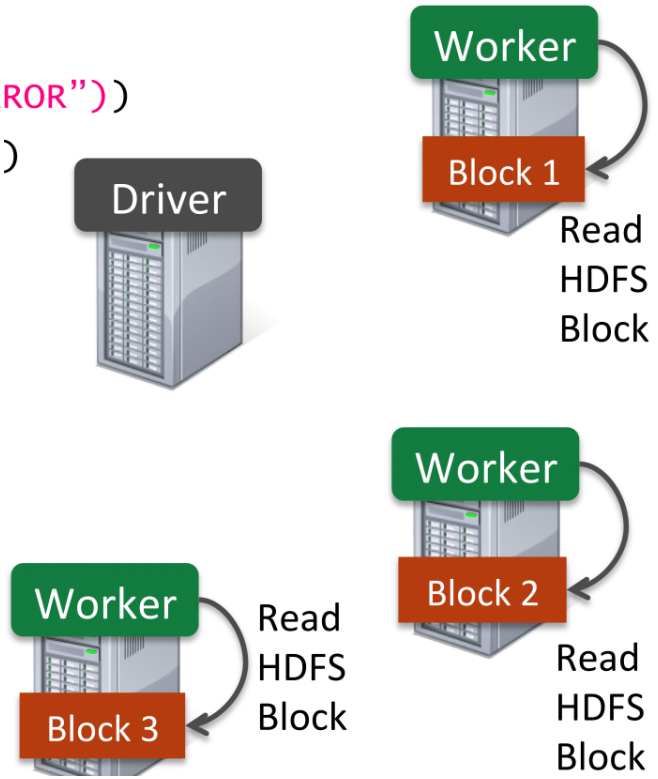


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
```

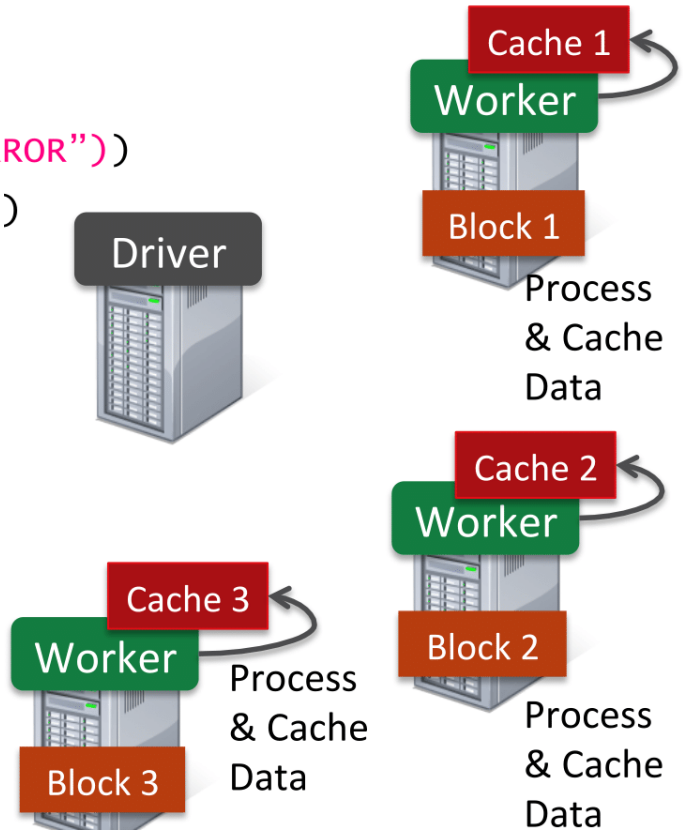


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
```

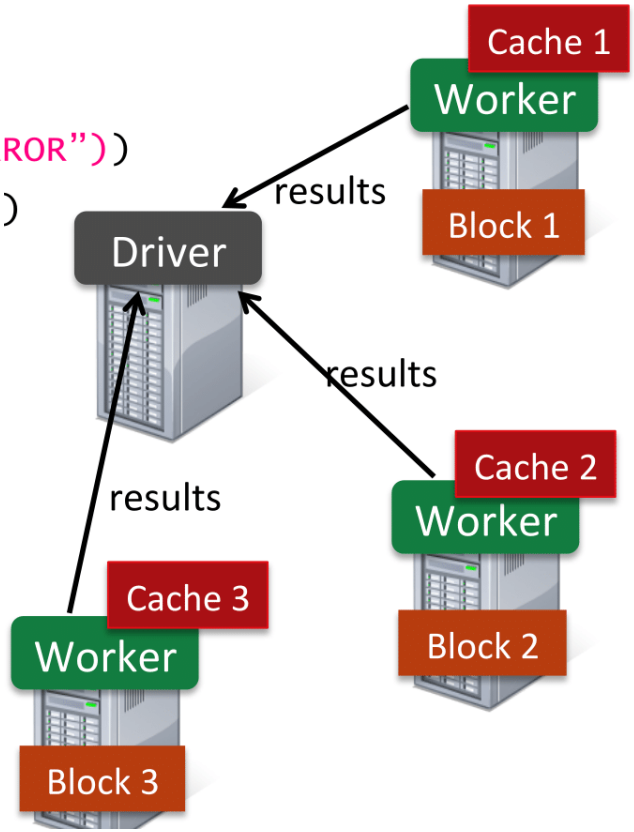


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
```

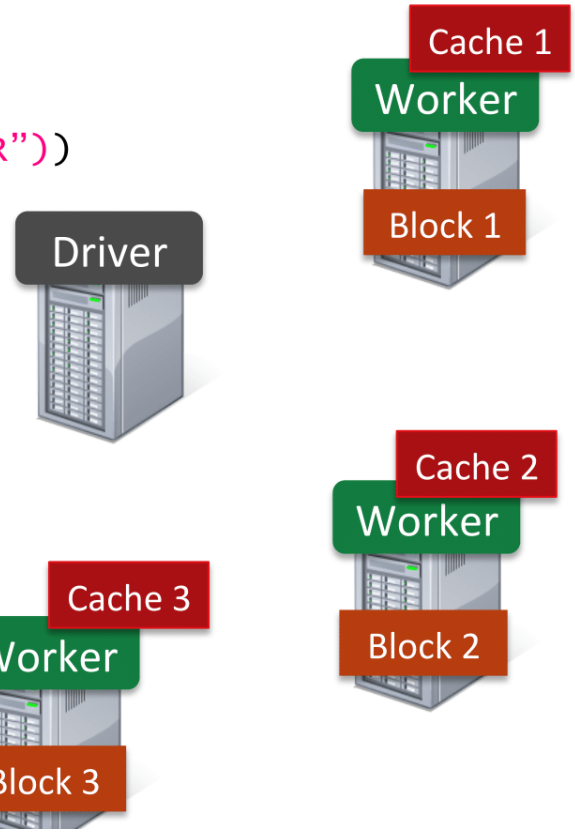


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

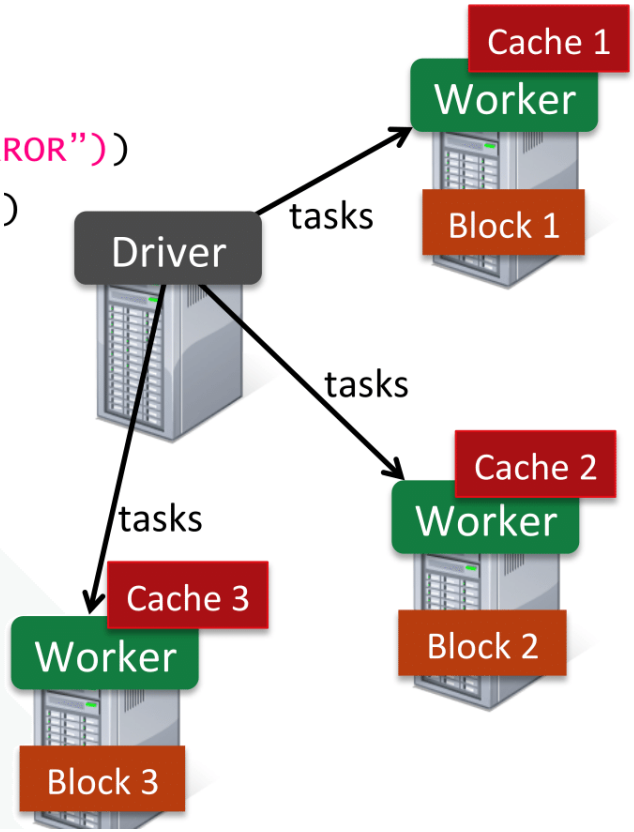


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

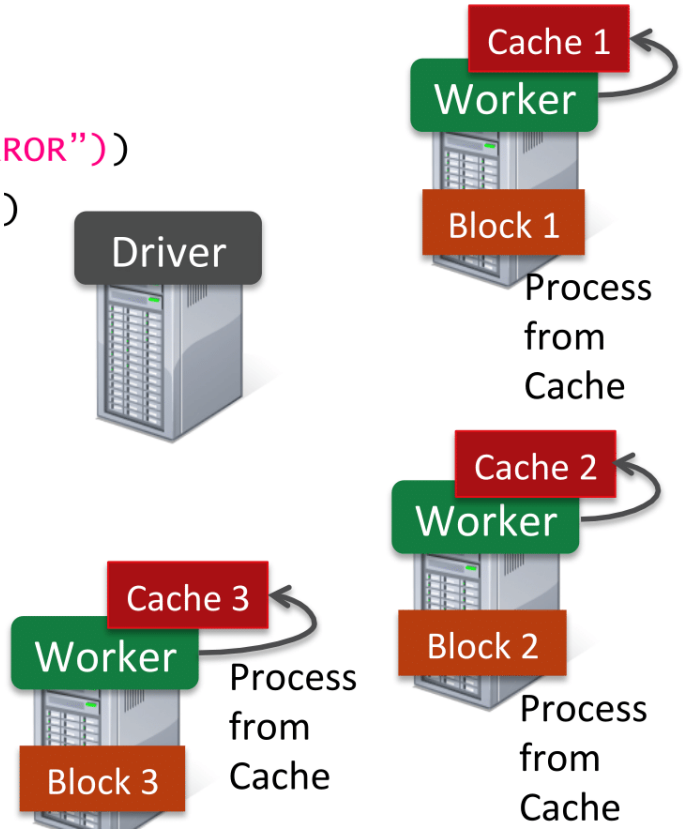


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

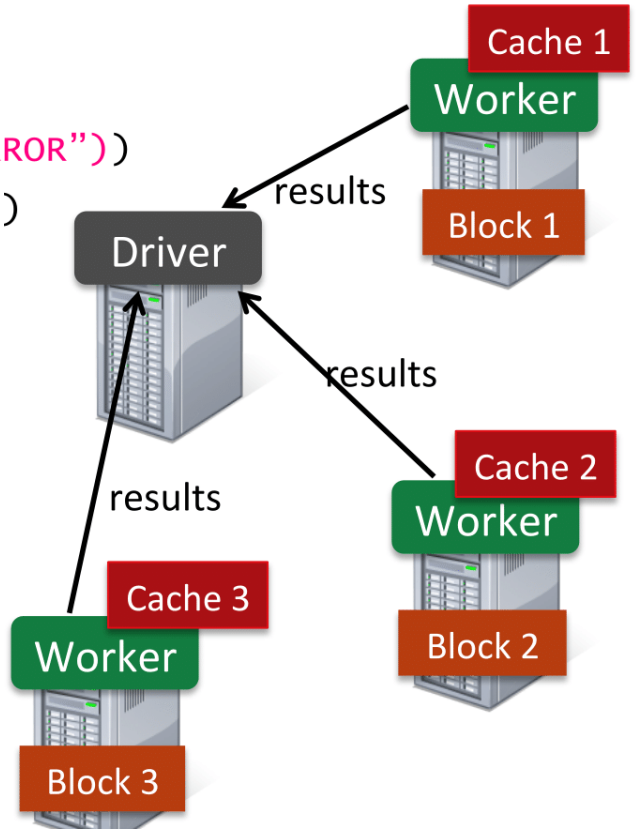


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

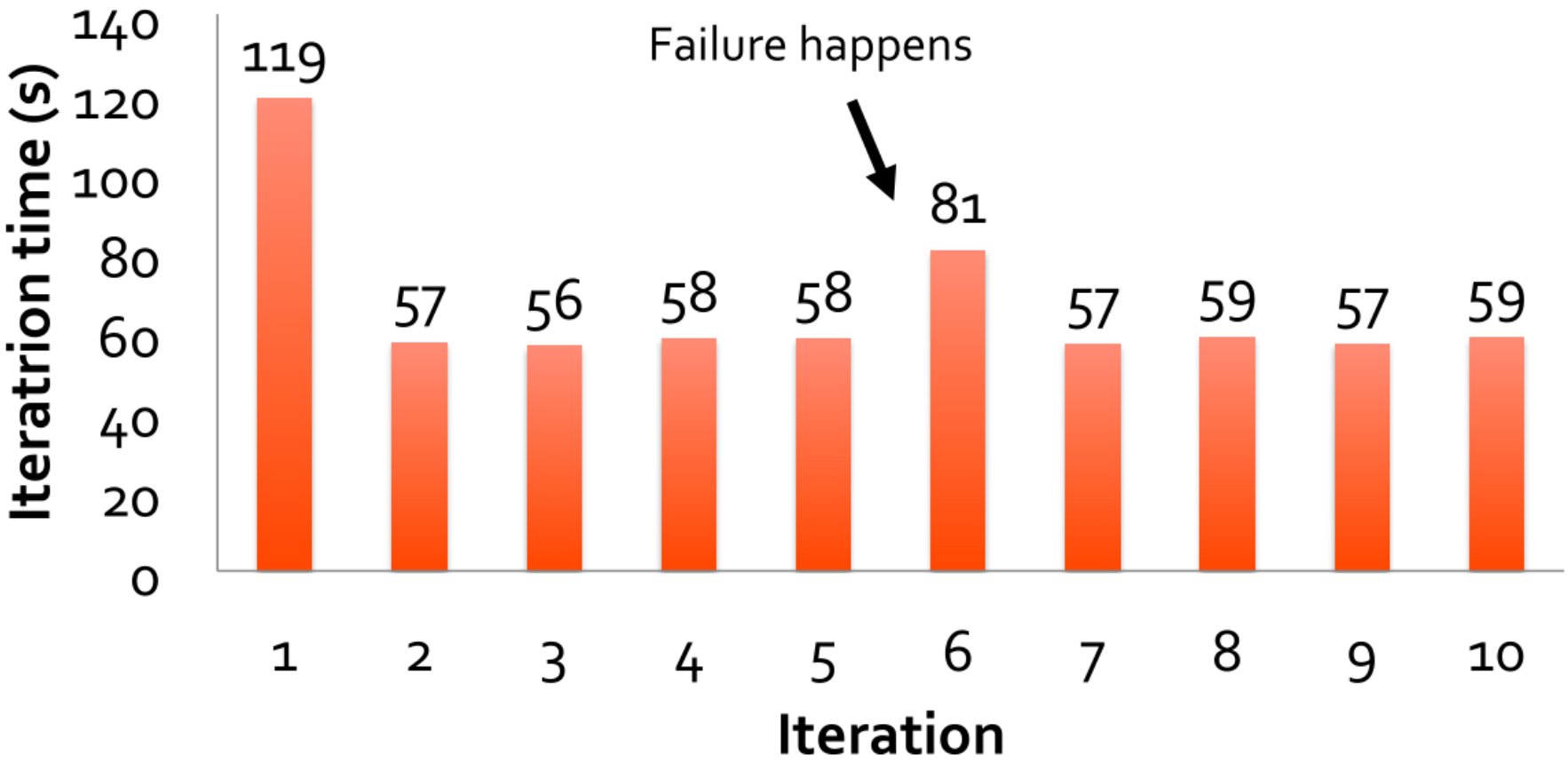
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



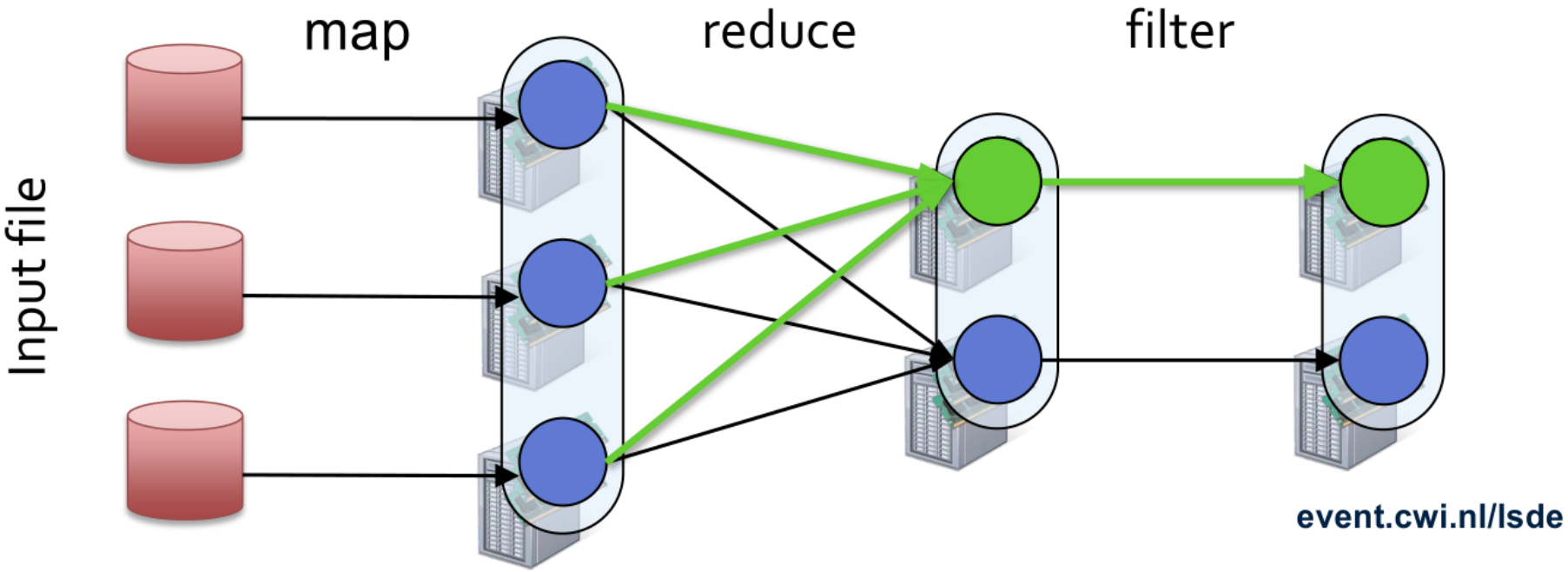
Fault tolerance (1)

- Lineage information in RDDs can be used to reconstruct lost partitions
- This is done transparently



Fault tolerance (2)

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Outline

Summary (Spark)

- Spark improves efficiency
  - In-memory computing
  - General dataflows
- Spark improves usability
  - Rich APIs in multiple languages
  - Interactive shell
- Spark unifies various big data analytics tasks
  - SQL, streams, machine learning, graphs, ...
- Fault tolerance and parallel processing retained

Literature

- Karau, Konwinski, Wendell, Zaharia
  Learning Spark: Lightning-fast data analysisO'Reilly, 2015

- Apache Spark
  http://spark.apache.org/

- Zaharia et al.
  Spark: Cluster Computing with Working Sets
  HotCloud 2010

- Michel Schinz, Philipp Haller
  A Scala tutorial for Java programmers
  2015

Outline

Outline

What is Apache Flink?

- Open source framework for distributed Big Data Analytics
  - Exploits data streaming
  - In-memory processing
  - Supports iteration operators
  - More later ...
- Originally **Stratosphere** project at TU-Berlin conceived by Prof. Volker Markl
- Now community driven like Apache Hadoop & Apache Spark
  - https://flink.apache.org/

Use case highlights

@ UBER

- Apache Flink deployed as a streaming platform service
- Billions messages / petabytes of data per day
- Incrementally realizing more and more services
  - Growth: "how much did we earn in SF in the last 5 mins"
  - Intelligent alerting: Ban driver/rider if suspicious activity
  - Intelligent forecasting: Increase accuracy of ETA models

12

@ Alibaba Group

Blink in Alibaba Production

✓In production for almost one year

✓Run on thousands of nodes

- hundreds of jobs
- The biggest cluster is more than 1000 nodes
- the biggest job has 10s TB states and thousands of subtasks

✓Supported key production services on last Nov 11th, China Single's Day

- China Single's Day is by far the biggest shopping holiday in China, similar to Black Friday in US
- Last year it recorded $17.8 billion worth of gross merchandise volumes in one day
- Blink is used to do real time machine learning and increased conversion by around 30%

13

@ NETFLIX

- Various use cases
  - Example: Stream ingestion, routing
  - Example: Model user interaction sessions
- Mix of stateless / moderate state / large state
- Stream Processing as a Service
  - Launching, monitoring, scaling, updating

4000+ Kafka brokers, 50+ clusters

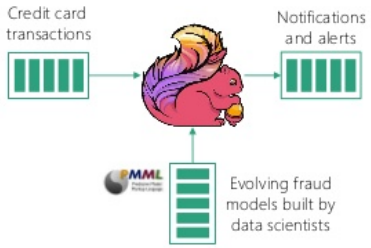
100's of Data Streams (Flink Jobs)

3700+ Docker containers running

1400+ nodes with 22K+ cpu cores

14

@ ING



Detecting fraud in real time

As fraudsters get better, need to update models without downtime

Live 24/7 service

15

Stephan Ewen@Flink Forward 2017

Powered by Flink

Alibaba, the world's largest retailer, uses a fork of Flink called Blink to optimize search rankings in real time.

Read more about Flink's role at Alibaba

BetterCloud, a multi-SaaS management platform, uses Flink to surface near real-time intelligence from SaaS application activity.

See BetterCloud at Flink Forward SF 2017

Bouygues Telecom is running 30 production applications powered by Flink and is processing 10 billion raw events per day.

See Bouygues Telcom at Flink Forward 2016

Capital One, a Fortune 500 financial services company, uses Flink for real-time activity monitoring and alerting.

See Capital One's case study slides

Drivetribe, a digital community founded by the former hosts of "Top Gear", uses Flink for metrics and content recommendations.

Read about Flink in the Drivetribe stack

Ericsson used Flink to build a real-time anomaly detector with machine learning over large infrastructures.

Read a detailed overview on O'Reilly Ideas

King, the creators of Candy Crush Saga, uses Flink to provide data science teams a real-time analytics dashboard.

Read about King's Flink implementation

MediaMath, a programmatic marketing company, uses Flink to power its real-time reporting infrastructure.

See MediaMath at Flink Forward SF 2017

Mux, an analytics company for streaming video providers, uses Flink for real-time anomaly detection and alerting.

Read more about how Mux is using Flink

Otto Group, the world's second-largest online retailer, uses Flink for business intelligence stream processing.

See Otto at Flink Forward 2016

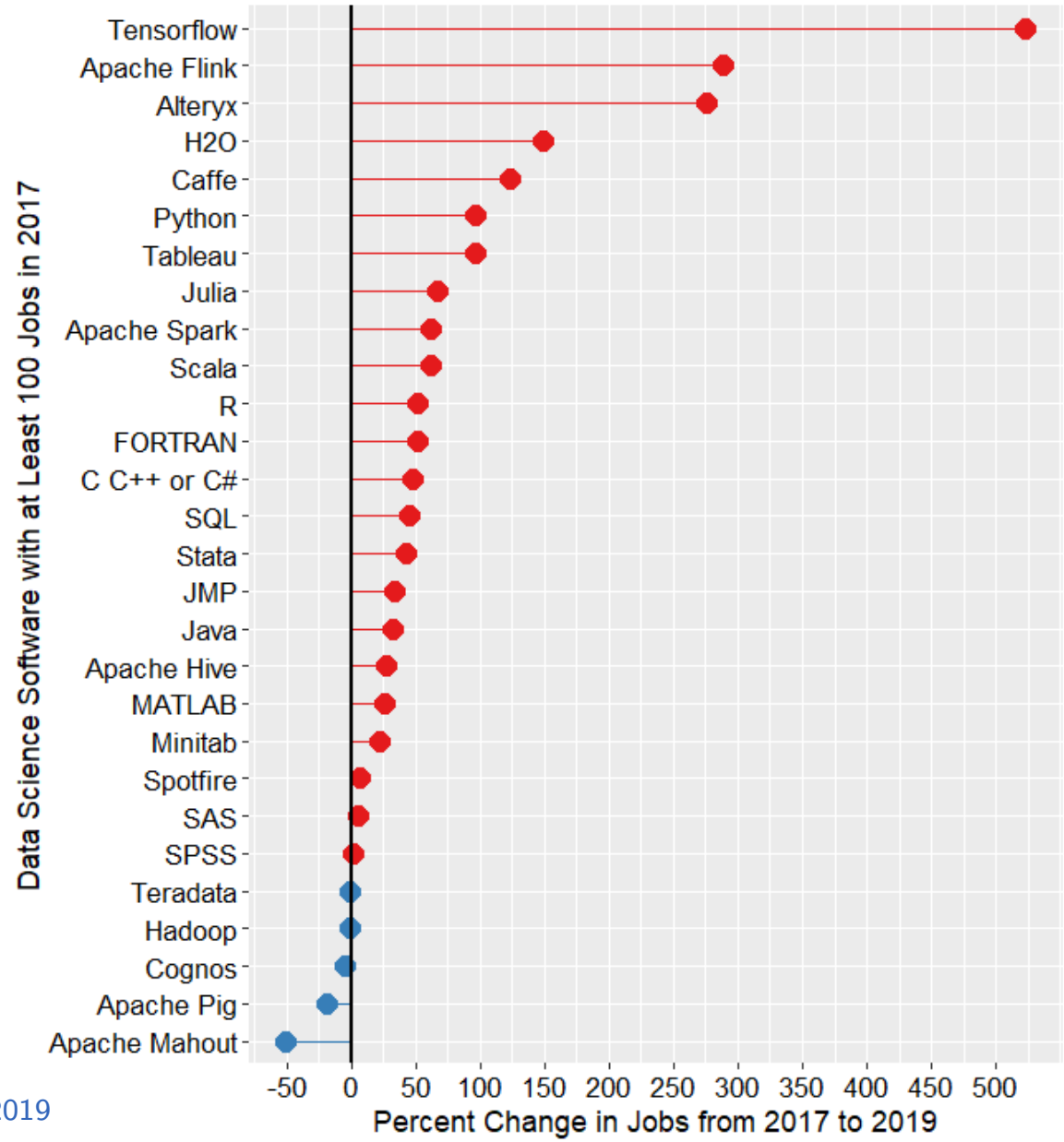
ResearchGate, a social network for scientists, uses Flink for network analysis and near-duplicate detection.

See ResearchGate at Flink Forward 2016

Zalando, one of the largest ecommerce companies in Europe, uses Flink for real-time process monitoring and ETL.

Read more on the Zalando Tech Blog

Flink as Top Data Science Software



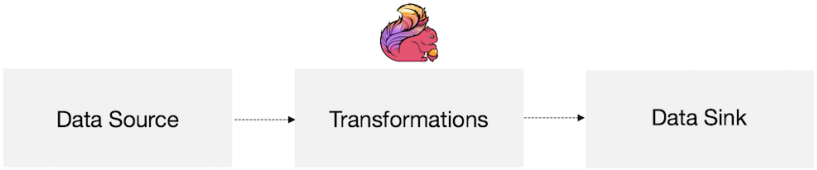
Data Science Jobs Report 2019

Apache Flink & Stream Processing: Overview

- Two types of datasets
  1. Unbounded
     - e.g., sensor data, financial markets, IoT,...
  2. Bounded (or batch data)
- Two types of execution models
  1. Streaming execution model
     - continuous processing on data that is continuously produced
  2. Batch execution model
     - Processing is "finite"
- Flink provides
  - A framework for distributed stream processing
    - Note: Flink treats batch as a special case of streaming processing
  - Accurate results
  - Fault tolerance
  - Scalability (high throughput, low latency)

Outline

Basic Flink program



- Flink program is made of
  - **Data source**: Incoming data that Flink processes
  - **Transformations**: The processing step, when Flink modifies incoming data
  - **Data sink**: Where Flink sends data after processing
- Example

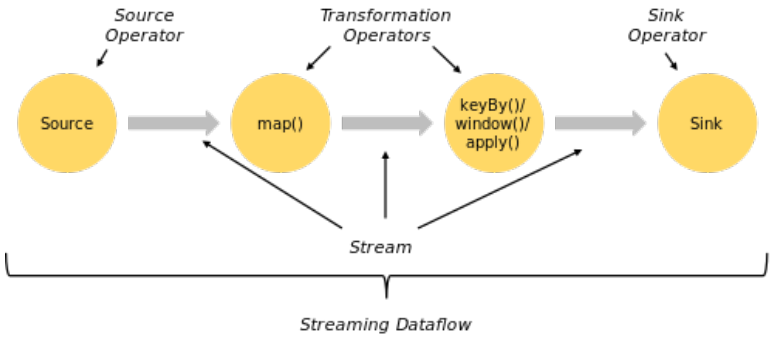
```
DataStream<String> lines = env.addSource(
                            new FlinkKafkaConsumer<>(…));

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

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

stats.addSink(new RollingSink(path));
```

Source
Transformation
Transformation
Sink

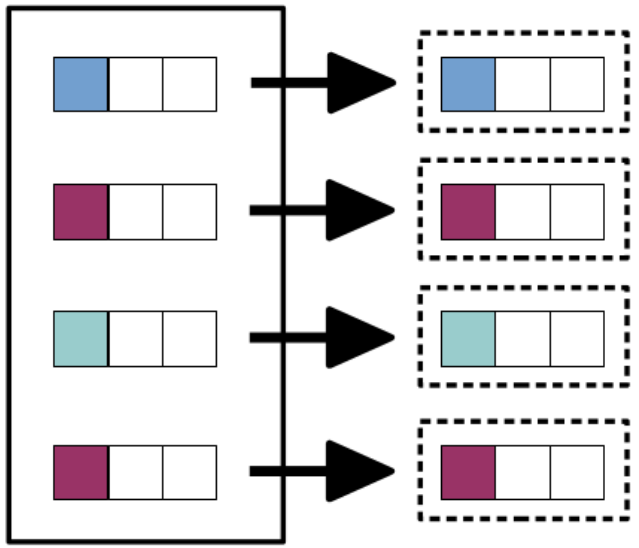


Transformations

- Based on Parallelization Contracts (PACTs): Core abstraction
  - Second order functions (generalization of MapReduce)
  - Core operators include
    - Map
    - Reduce
    - Cross
    - Join
    - CoGroup

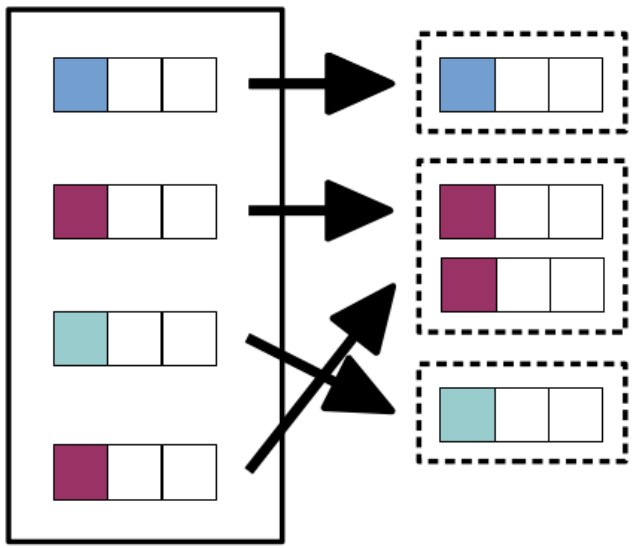
Transformations

- Based on Parallelization Contracts (PACTs): Core abstraction
  - Second order functions (generalization of MapReduce)
  - Core operators include
    - **Map**
    - Reduce
    - Cross
    - Join
    - CoGroup



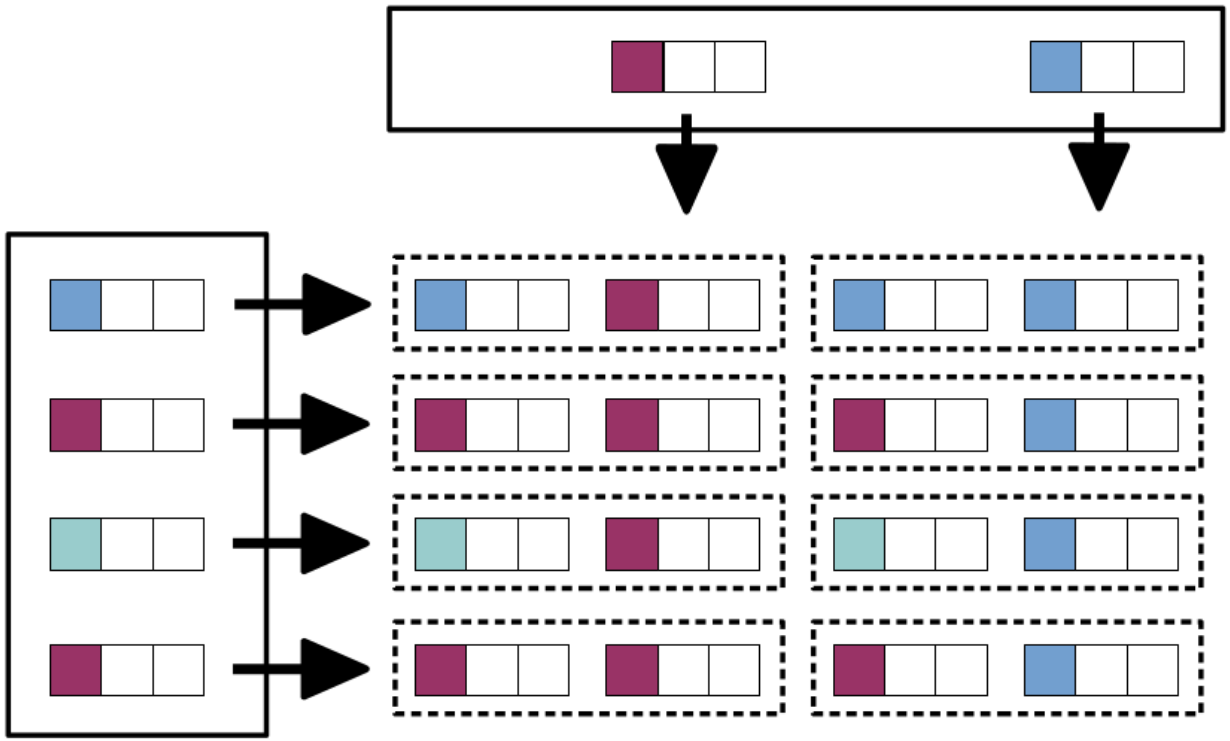
Transformations

- Based on Parallelization Contracts (PACTs): Core abstraction
  - Second order functions (generalization of MapReduce)
  - Core operators include
    - Map
    - **Reduce**
    - Cross
    - Join
    - CoGroup



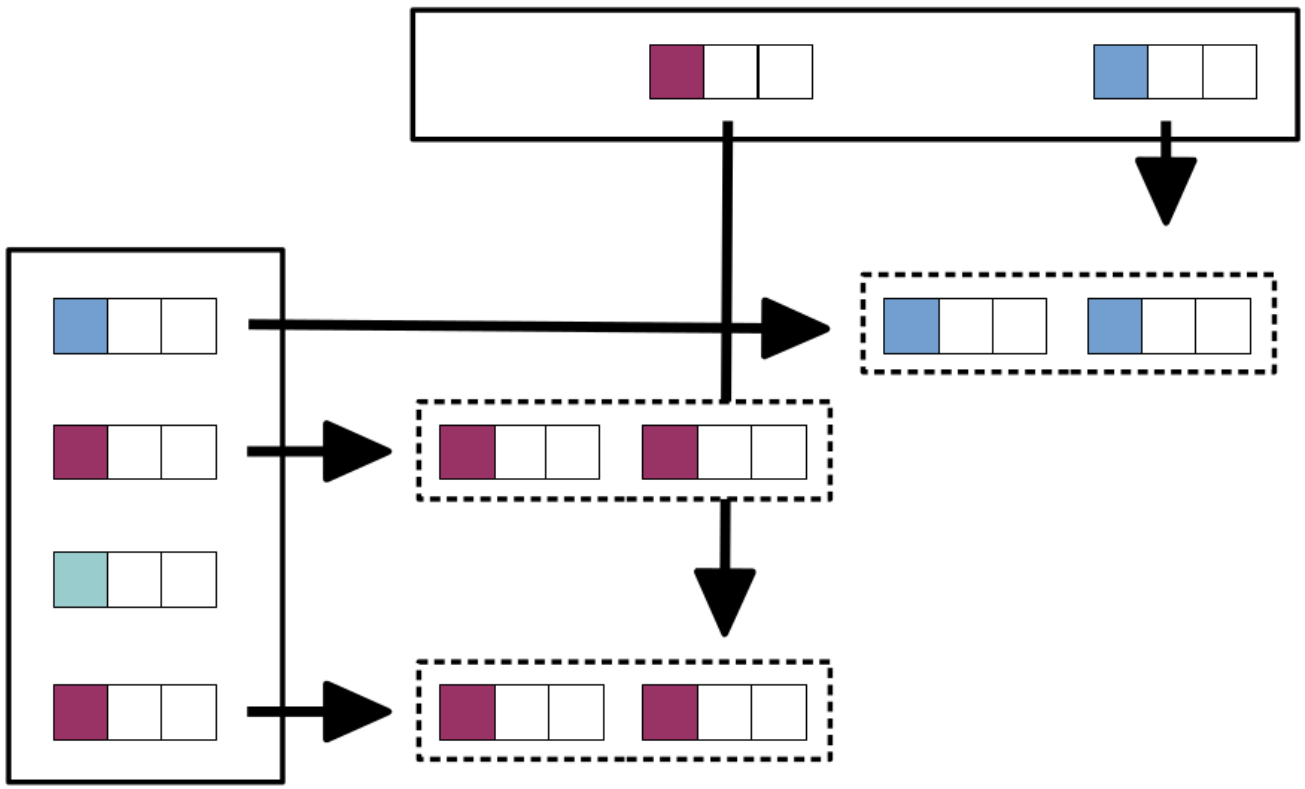
Transformations

- Based on Parallelization Contracts (PACTs): Core abstraction
  - Second order functions (generalization of MapReduce)
  - Core operators include
    - Map
    - Reduce
    - **Cross**
    - Join
    - CoGroup



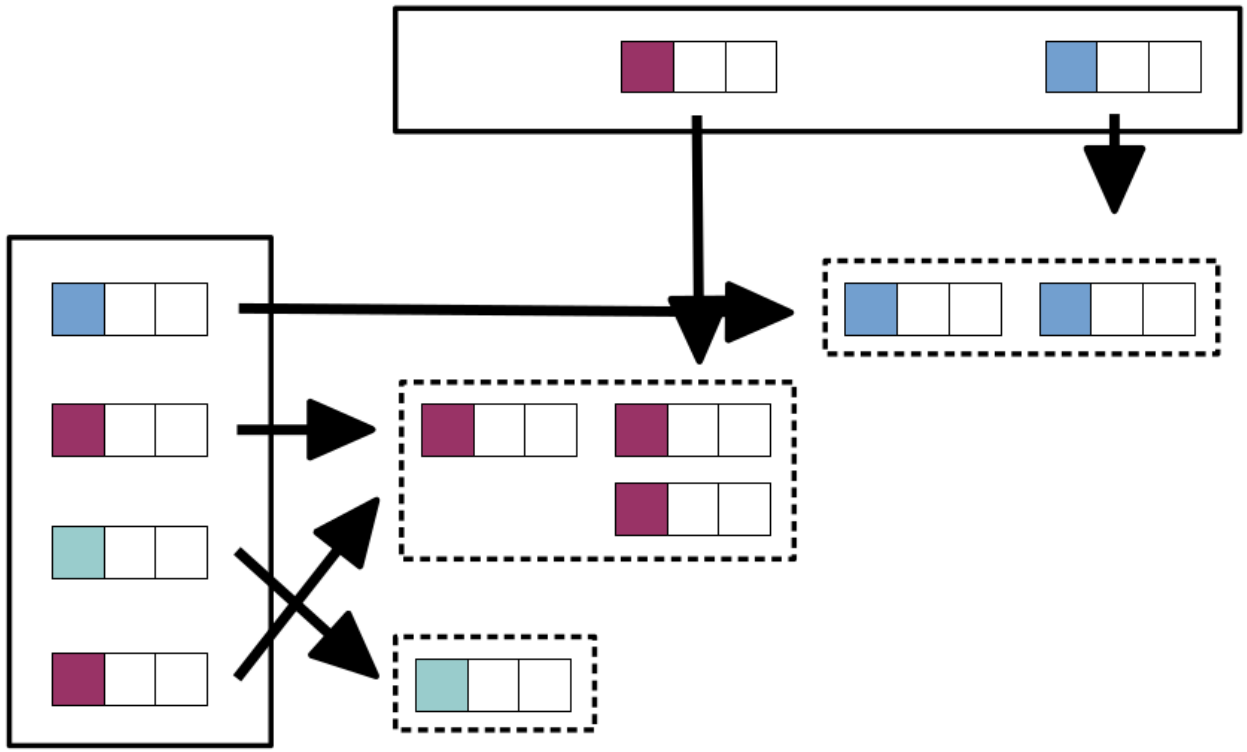
Transformations

- Based on Parallelization Contracts (PACTs): Core abstraction
  - Second order functions (generalization of MapReduce)
  - Core operators include
    - Map
    - Reduce
    - Cross
    - **Join**
    - CoGroup



Transformations

- Based on Parallelization Contracts (PACTs): Core abstraction
  - Second order functions (generalization of MapReduce)
  - Core operators include
    - Map
    - Reduce
    - Cross
    - Join
    - **CoGroup**



Transformations

- Based on Parallelization Contracts (PACTs): Core abstraction
  - Second order functions (generalization of MapReduce)
  - Core operators include
    - Map
    - Reduce
    - Cross
    - Join
    - CoGroup
  - Additionally includes many rich set of operators

Union,	Iterate,	Delta Iterate,
Filter,	GroupReduce,	FlatMap,
Project,	Aggregate,	Distinct,
Vertex Update,	...	

Flink's core API

- Supports both stream & batch processing

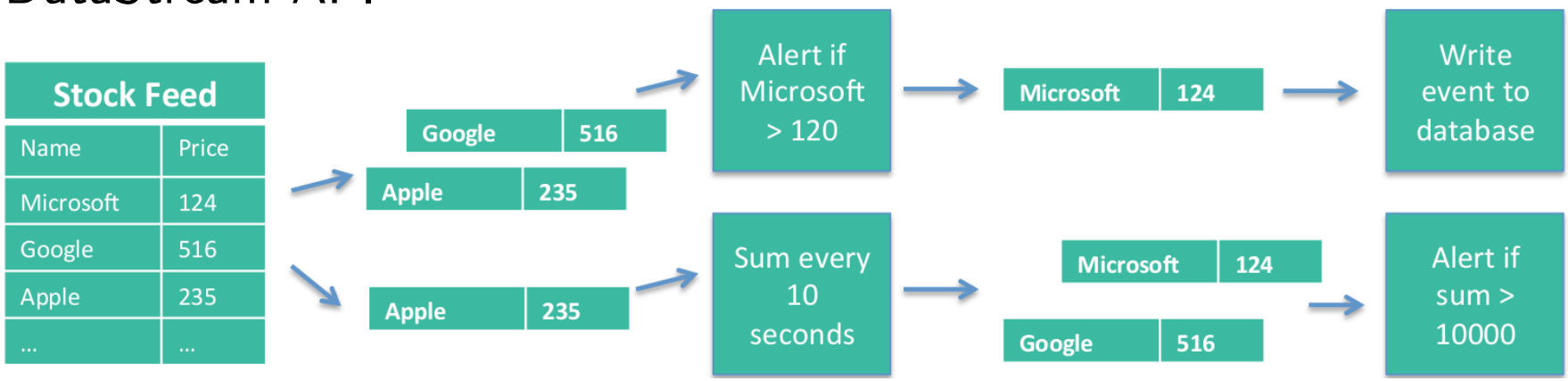
	Streaming	**Batch**
Input	infinite	finite
Data transfer	pipelined	blocking or pipelined
Latency	low	high

Flink's core API

- Supports both stream & batch processing

	Streaming	Batch
Input	infinite	finite
Data transfer	pipelined	blocking or pipelined
Latency	low	high

- DataStream API

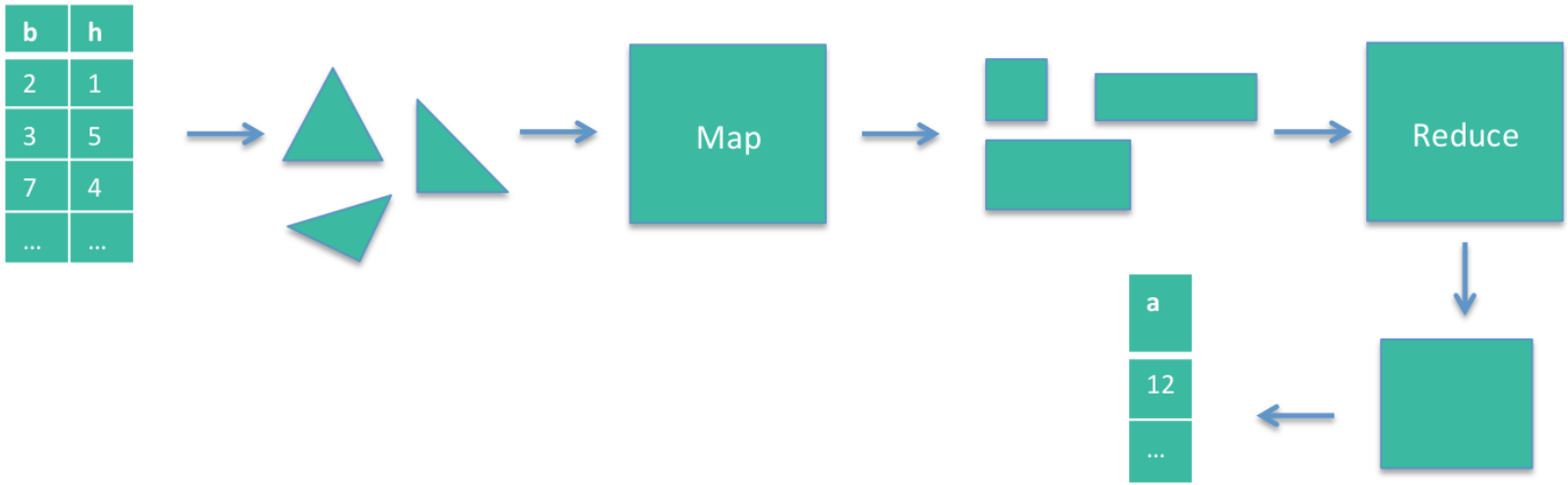


Flink's core API

- Supports both stream & batch processing

	Streaming	**Batch**
Input	infinite	finite
Data transfer	pipelined	blocking or pipelined
Latency	low	high

- DataSet API



Example (Word Count)

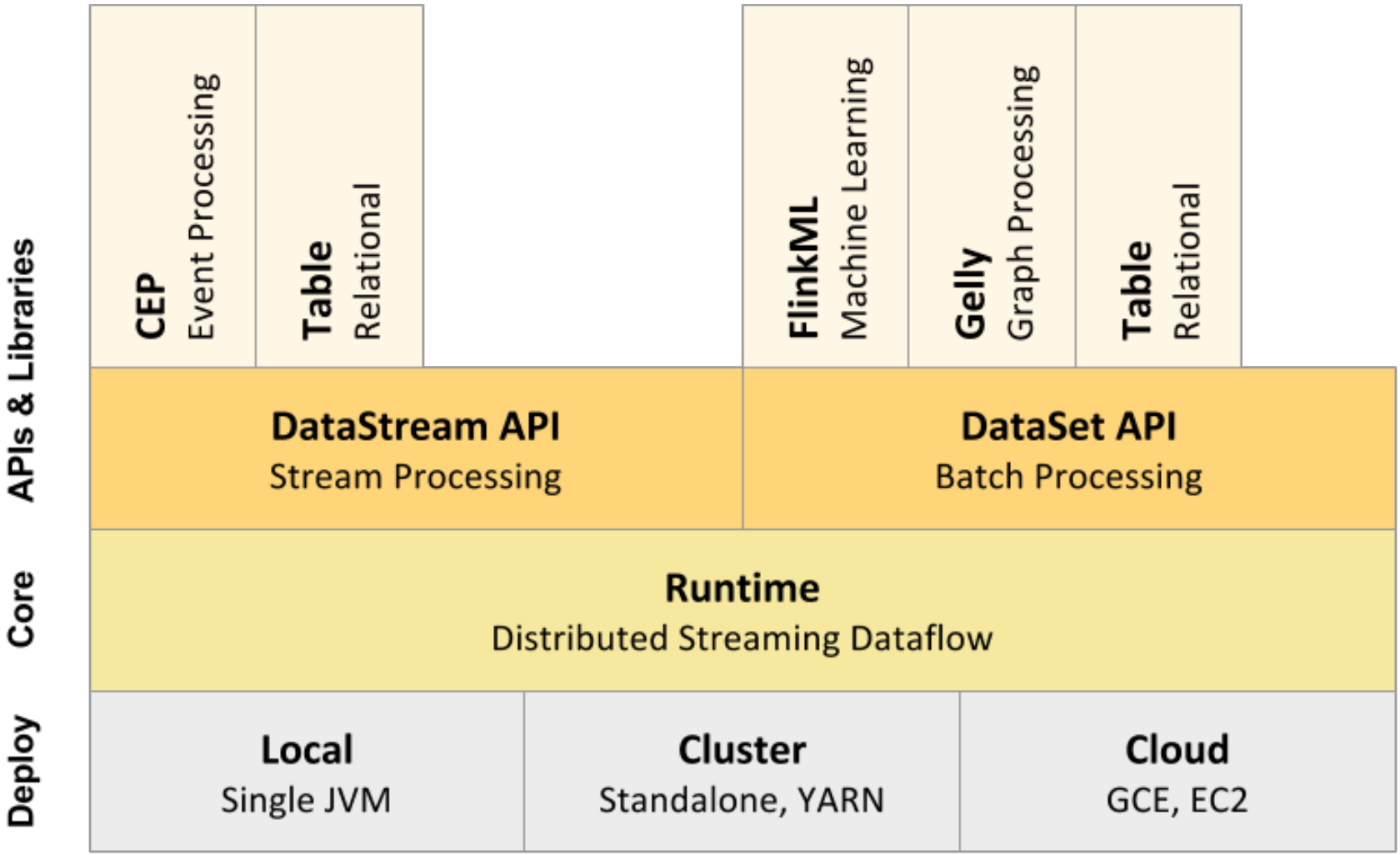
- DataSet API

```
val lines = env.readTextFile(...)
lines.flatMap {line => line.split(" ")
                       .map(word => Word(word,1))}
     .groupBy(0).sum(1)
     .print()
```

- DataStream API

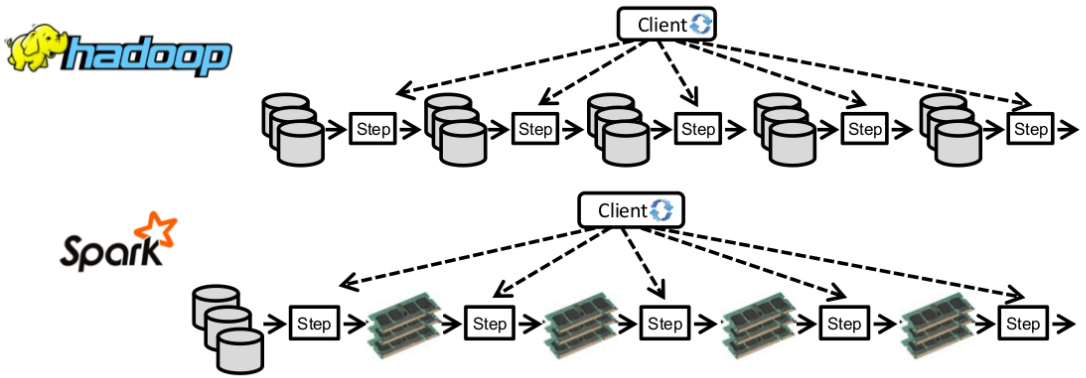
```
val lines = env.socketTextStream(...)
lines.flatMap {line => line.split(" ")
                       .map(word => Word(word,1))}
     .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS)
         )
     .groupBy(0).sum(1)
     .print()
```

Flink Stack


APIs & Libraries
CEP
Event Processing
Table
Relational
FlinkML
Machine Learning
Gelly
Graph Processing
Table
Relational
DataStream API
Stream Processing
DataSet API
Batch Processing
Core
Runtime
Distributed Streaming Dataflow
Deploy
Local
Single JVM
Cluster
Standalone, YARN
Cloud
GCE, EC2

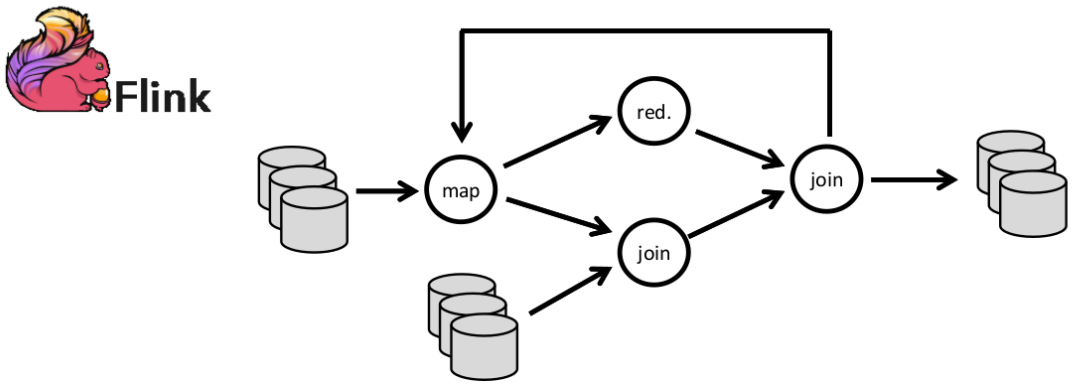
Iterations in Flink

- Flink offers **built-in** iterations and **delta iterations**
  - Key for efficiently executing machine learning and graph algorithms
- Built-in vs driver-based iterations



Loop outside the system, in driver program

Iterative program looks like many independent jobs

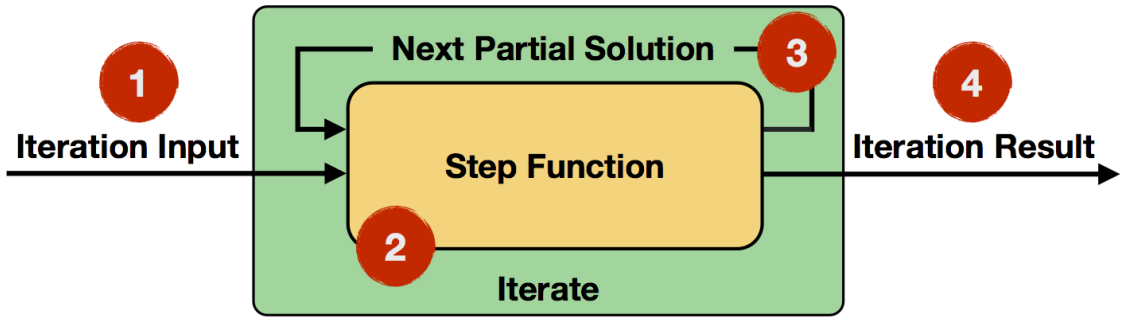


Dataflows with feedback edges

System is iteration-aware, can optimize the job

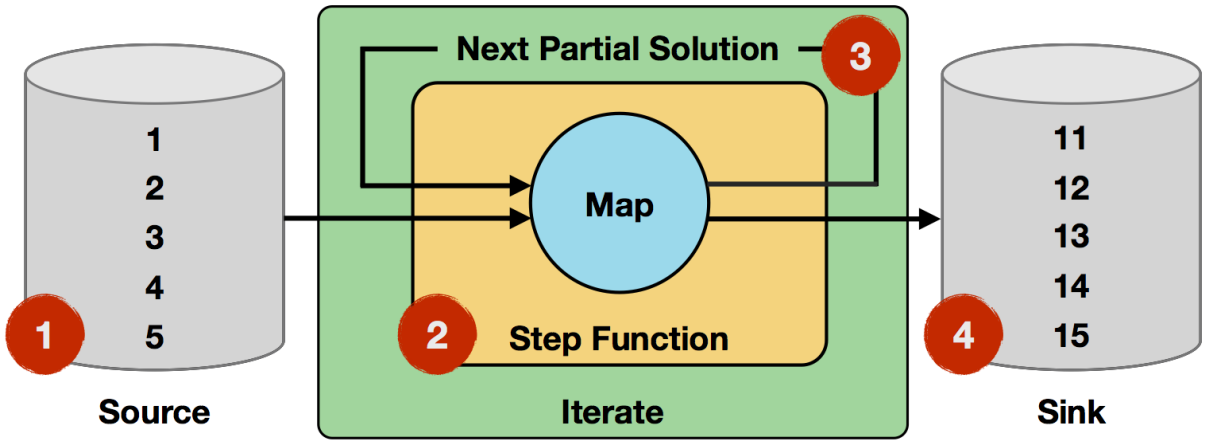
Iterate operator

- Built-in operator to support looping over data
- Applies step function to partial solution until convergence
- Step function can be arbitrary Flink program
- Convergence via fixed number of iterations or custom convergence criterion



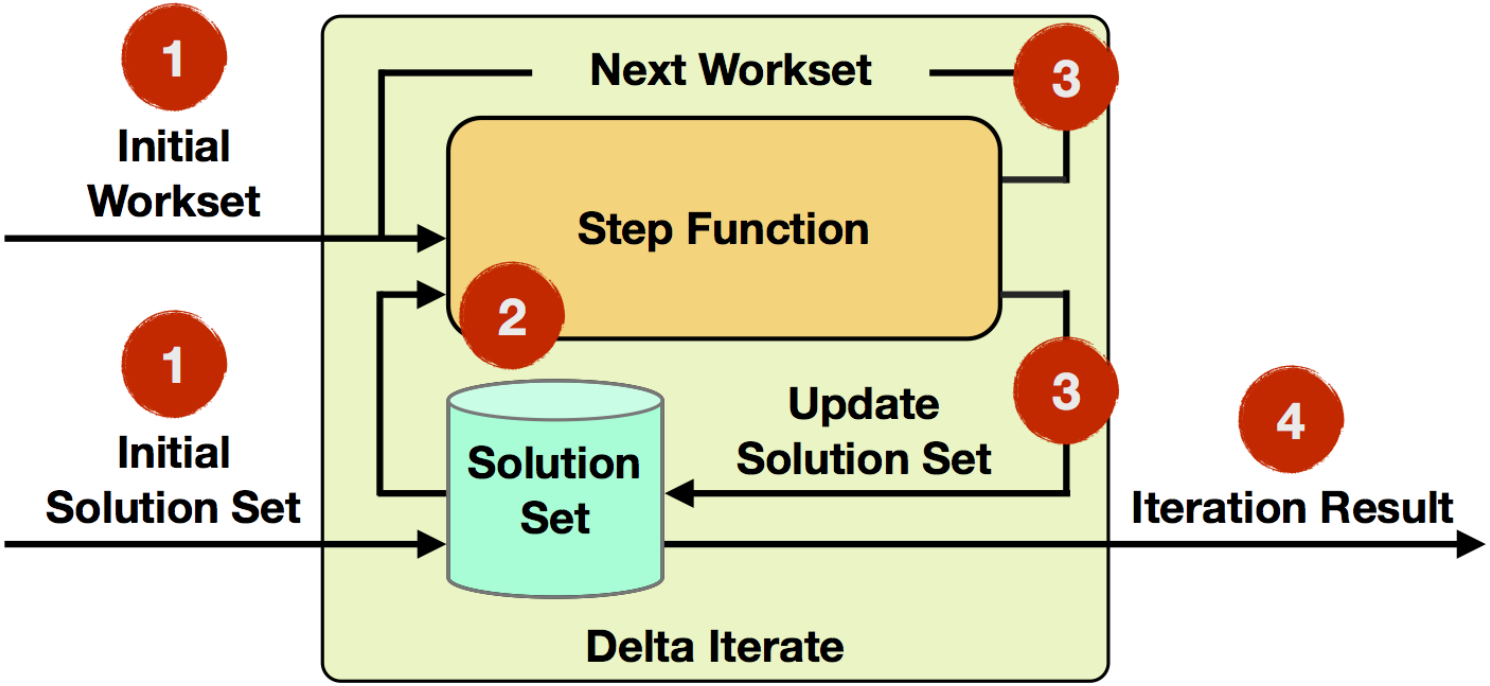
Example

- Iteratively increment a set of numbers
  1. Input: five single-field records(ints 1 to 5)
  2. Step function: map($x \Rightarrow x + 1$)
  3. Next partial solution: output of map operator
  4. Result: after 10 iterations 10



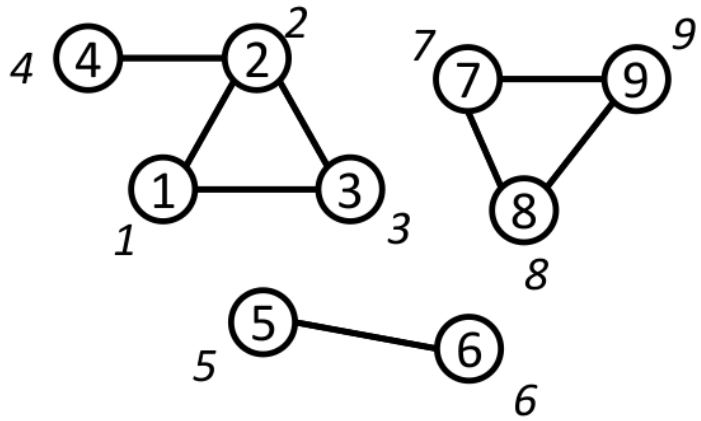
Delta Iterate operator

- Compute next workset and changes to the partial solution until workset is empty
- generalizes vertex-centric computing of Pregel and GraphLab



Example

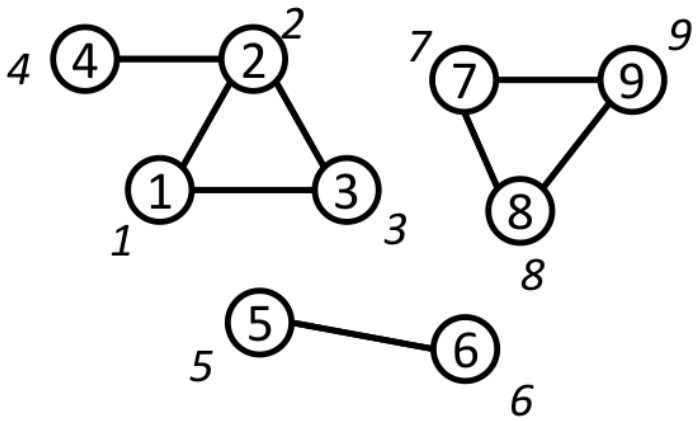
- Find connected connected components of a graph
- Example



- Start
  - Vertices have IDs that represent the component they belong to
  - Initially, every vertex has its own id (its own component)
- Step
  - Each vertex tells its neighbors its component id
  - Vertices take the min-ID of all candidates from its neighbors
  - A vertex that did not adopt a new ID need not participate in the next step

Example: connected components

Solution Set



Workset

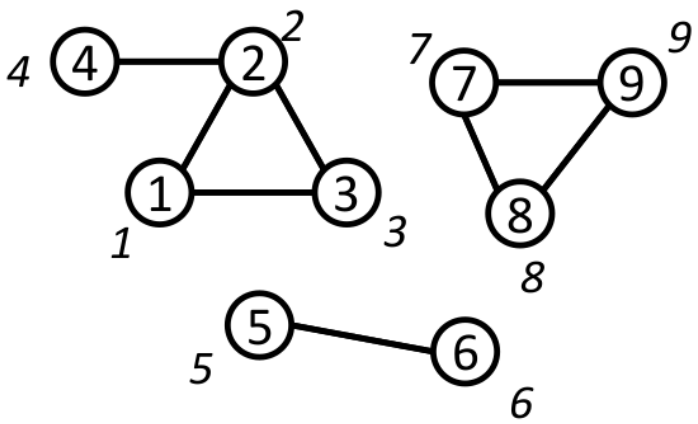
1 (2,2)	3 (1,1)	8 (7,7)
(3,3)	(2,2)	(9,9)
2 (1,1)	4 (2,2)	9 (7,7)
(3,3)		(8,8)
(4,4)	5 (6,6)	
	6 (5,5)	

Solution Set Delta

Messages sent to neighbors:

1 (4, 3) means that vertex 1 receives a candidate id of 3 from vertex 4

Example: connected components



Workset

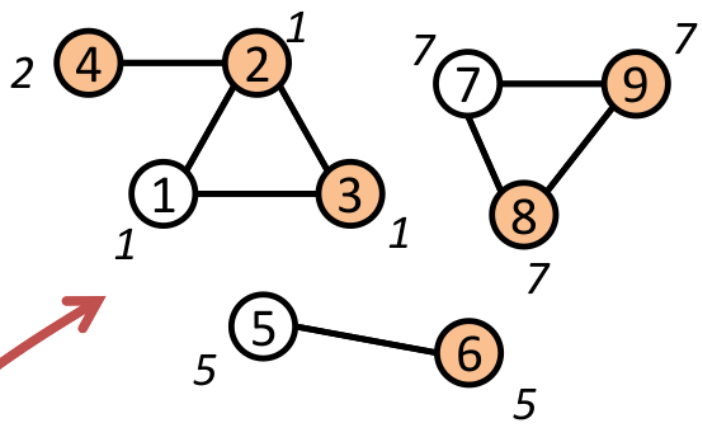
1 (2,2)	3 (1,1)	8 (7,7)
(3,3)	(2,2)	(9,9)
2 (1,1)	4 (2,2)	9 (7,7)
(3,3)		(8,8)
(4,4)	5 (6,6)	
	6 (5,5)	

Solution Set Delta

(2,1)	(6, 5)
(3,1)	(8,7)
(4,2)	(9,7)

Example: connected components

Solution Set



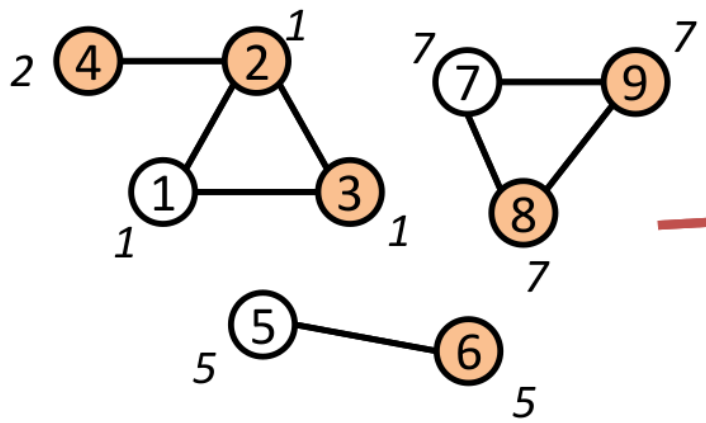
Workset

1 (2,2)	3 (1,1)	8 (7,7)
(3,3)	(2,2)	(9,9)
2 (1,1)	4 (2,2)	9 (7,7)
(3,3)		(8,8)
(4,4)	5 (6,6)	
	6 (5,5)	

Solution Set Delta

(2,1)	(6, 5)
(3,1)	(8,7)
(4,2)	(9,7)

Example: connected components



Workset

1 (2,1)
(3,1)

2 (3,1)
(4,2)

3 (2,1)

4 (2,1)

5 (6,5)

7 (8,7)
(9,7)

8 (9,7)

9 (8,7)

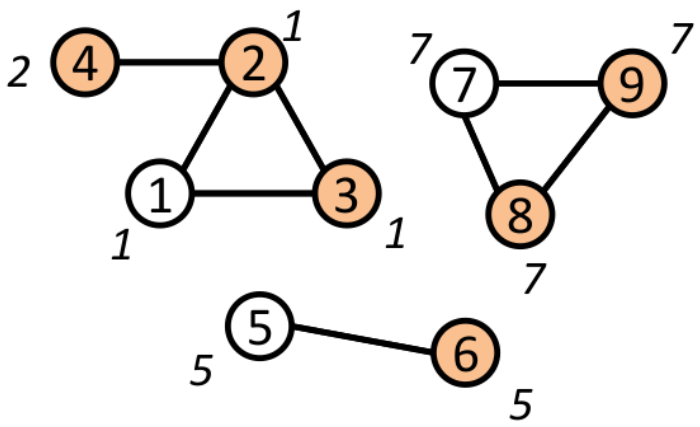
Solution Set Delta

(2,1)
(3,1)
(4,2)

(6, 5)
(8,7)
(9,7)

Example: connected components

Solution Set



Workset

1 (2,1)
(3,1)

2 (3,1)
(4,2)

3 (2,1)

4 (2,1)

5 (6,5)

7 (8,7)
(9,7)

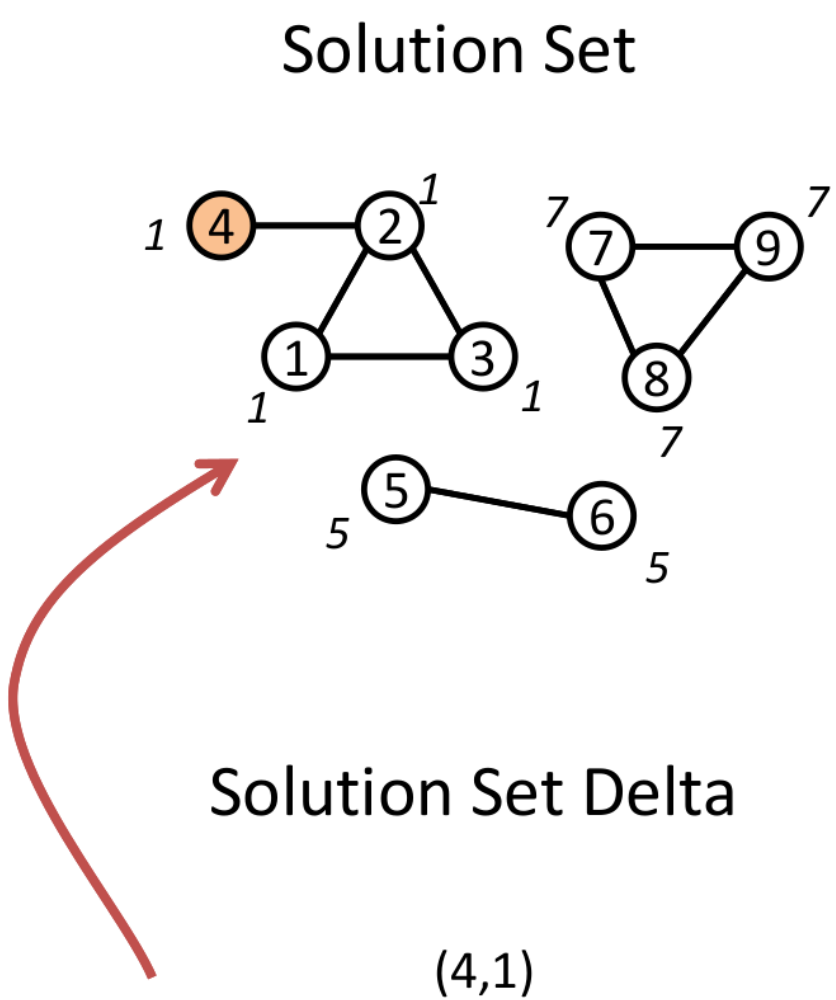
8 (9,7)

9 (8,7)

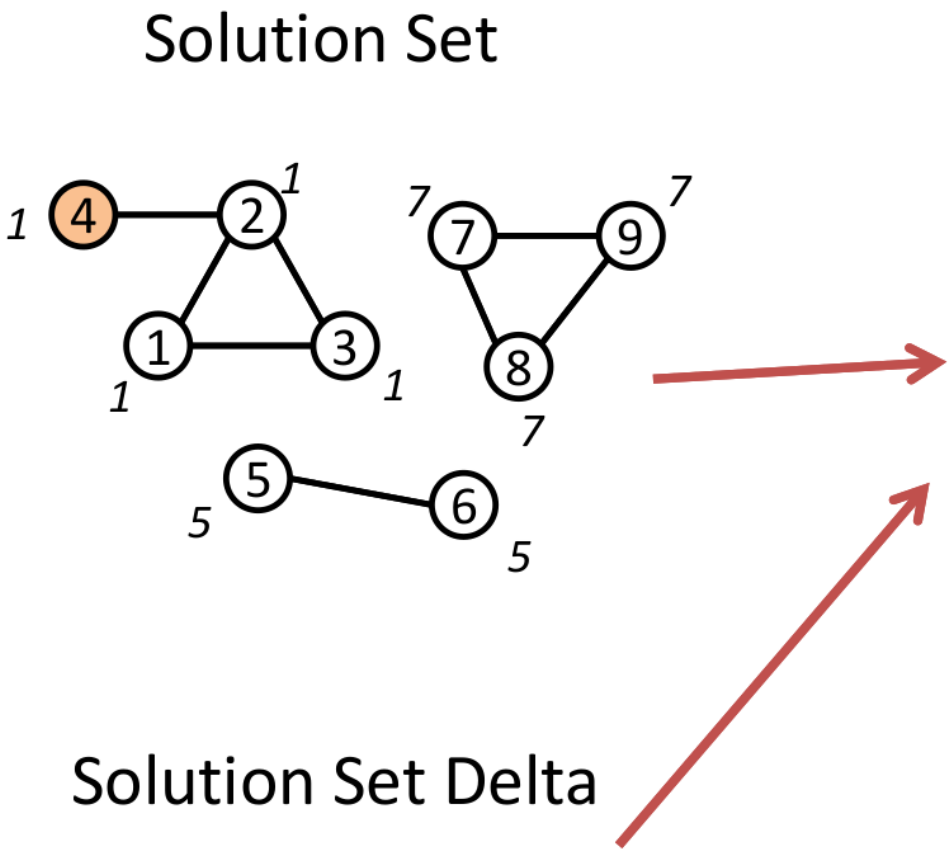
Solution Set Delta

(4,1)

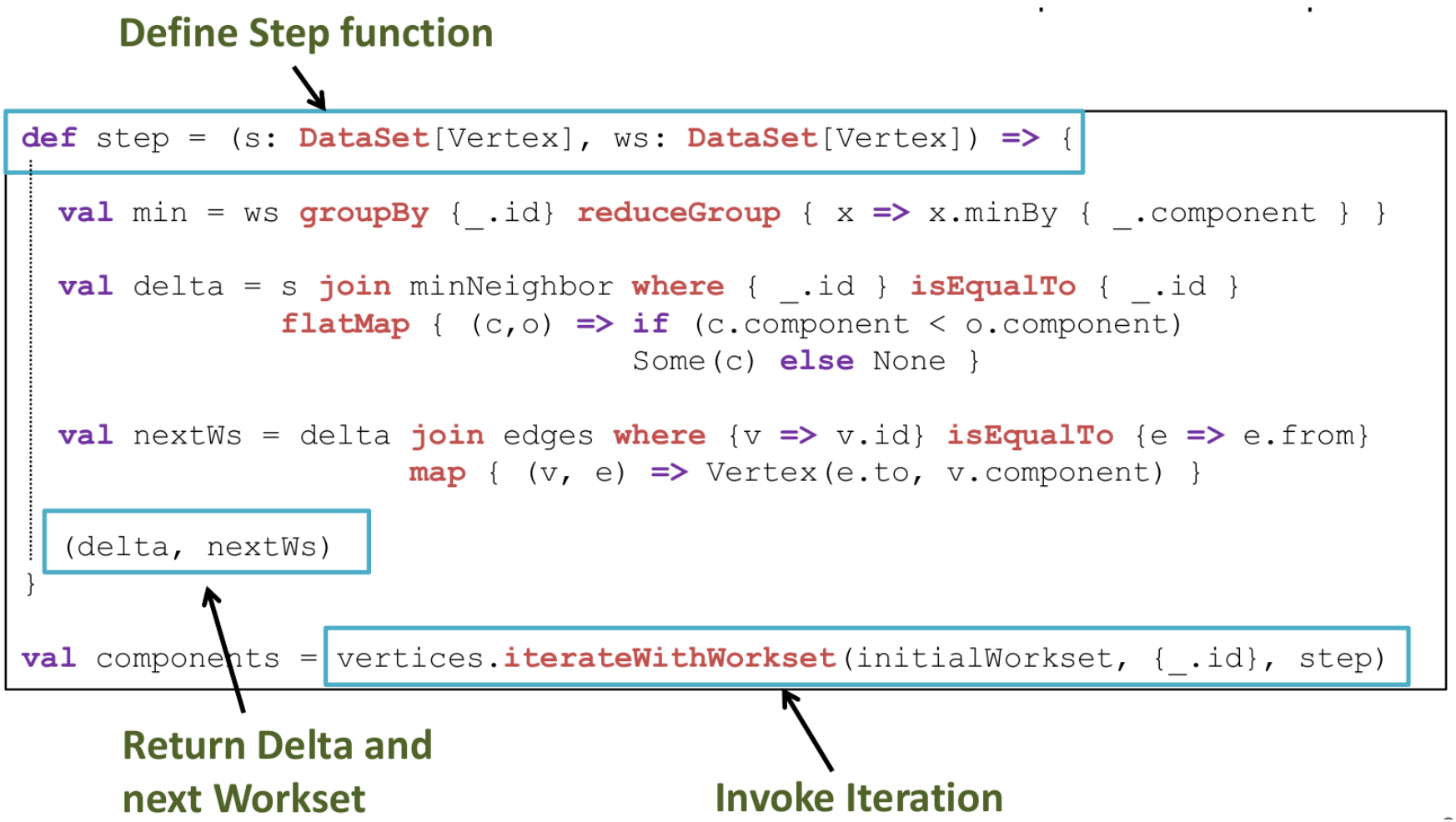
Example: connected components



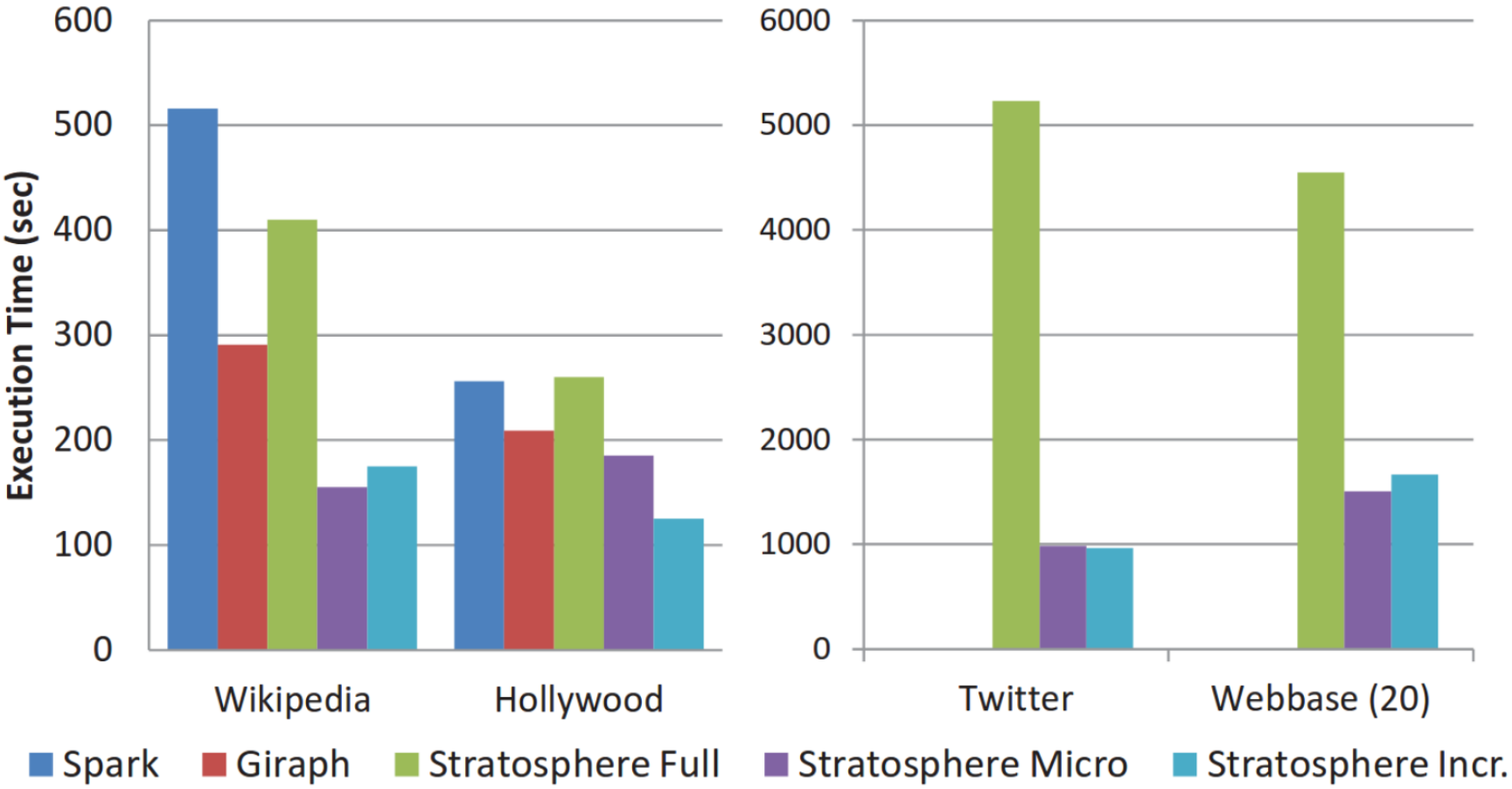
Example: connected components



Example: connected components



Example: connected components

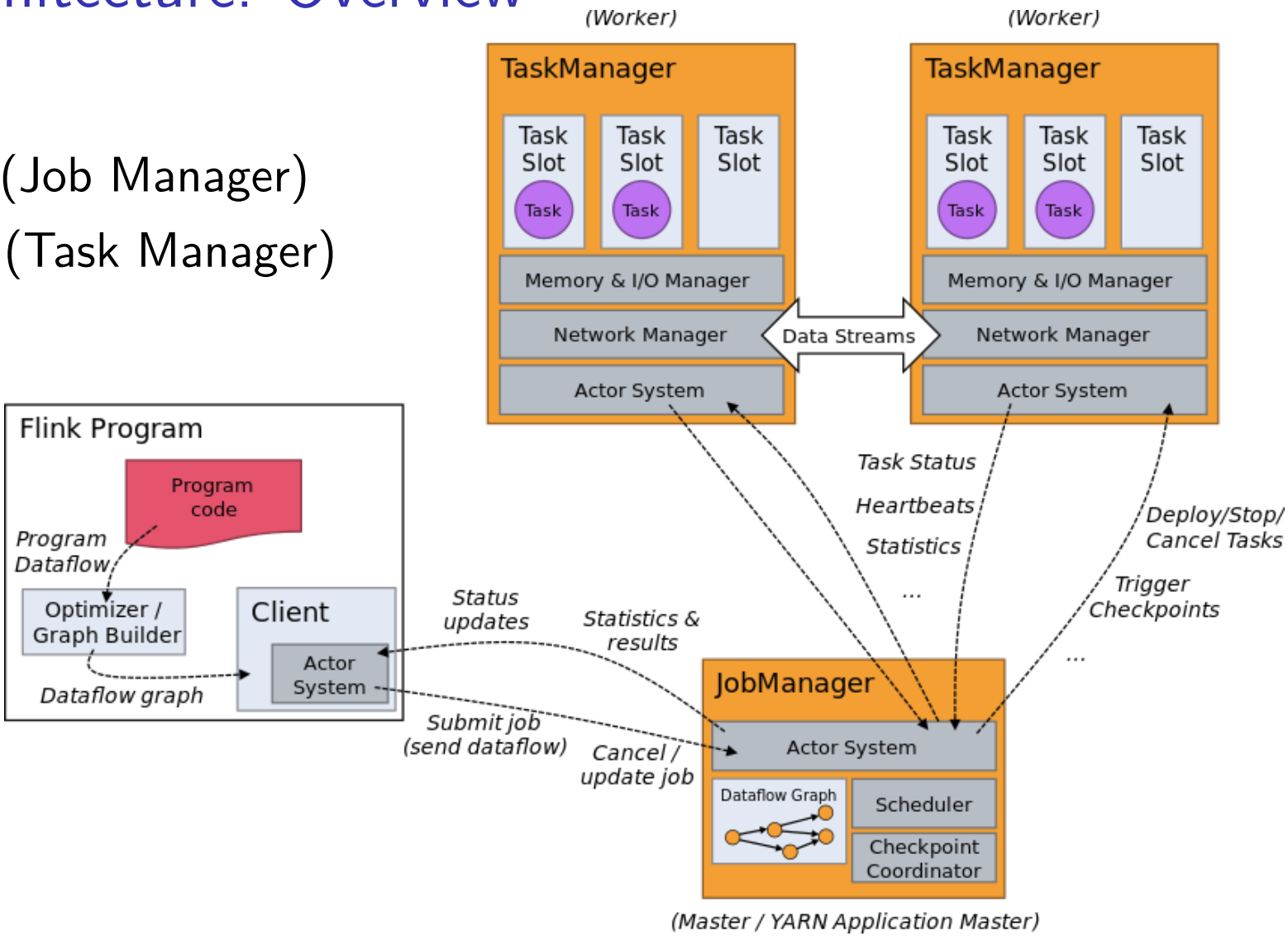


Spinning Fast Iterative Data Flows, VLDB 2012

Outline

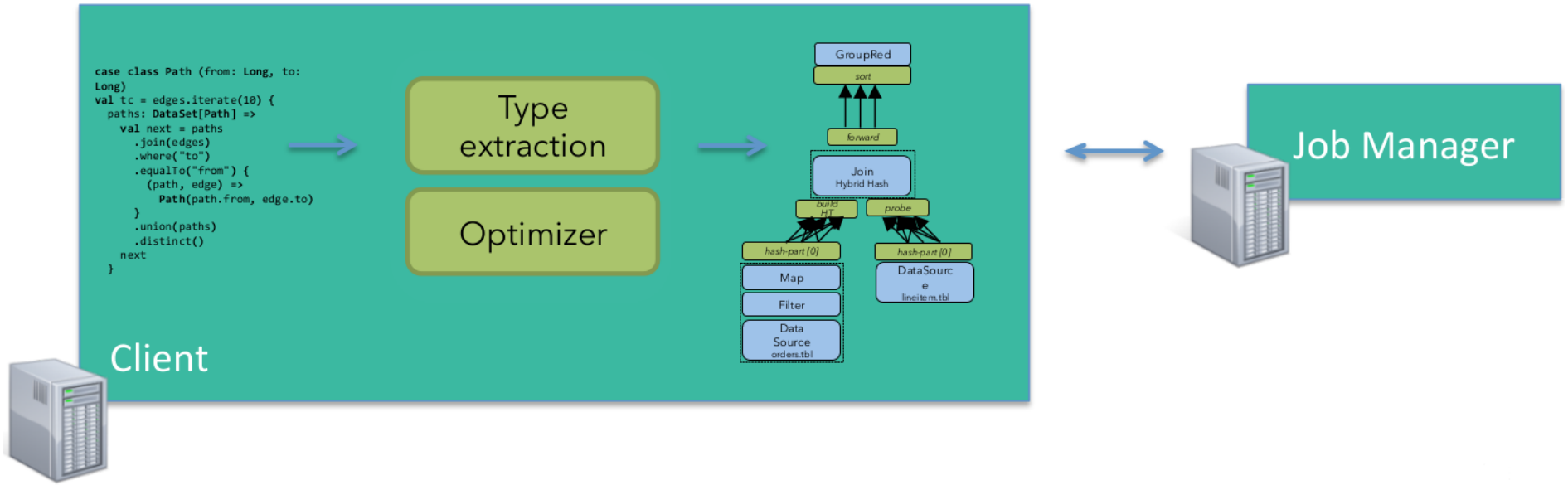
Flink's architecture: Overview

- Client
- Master (Job Manager)
- Worker (Task Manager)



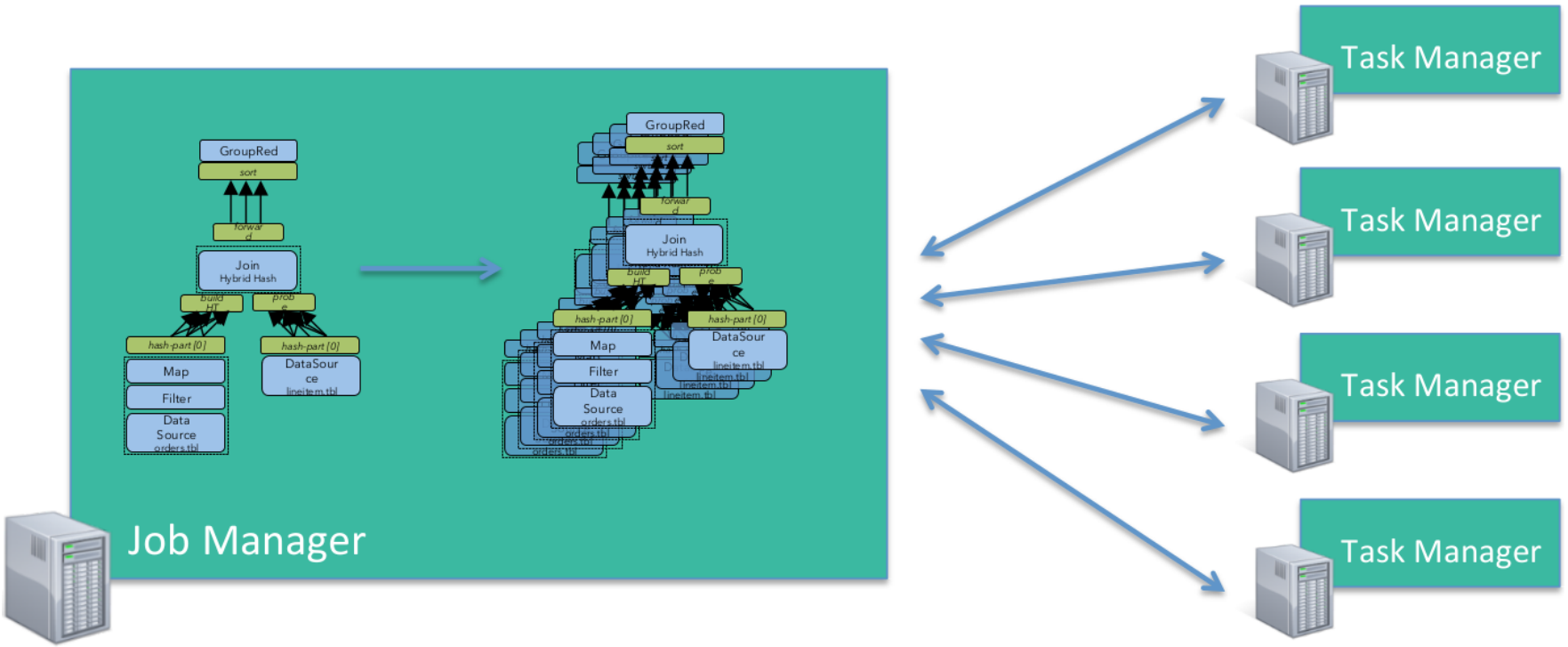
Flink's architecture: Client

- Optimize
- Construct job graph
- Pass job graph to job manager
- Retrieve job results



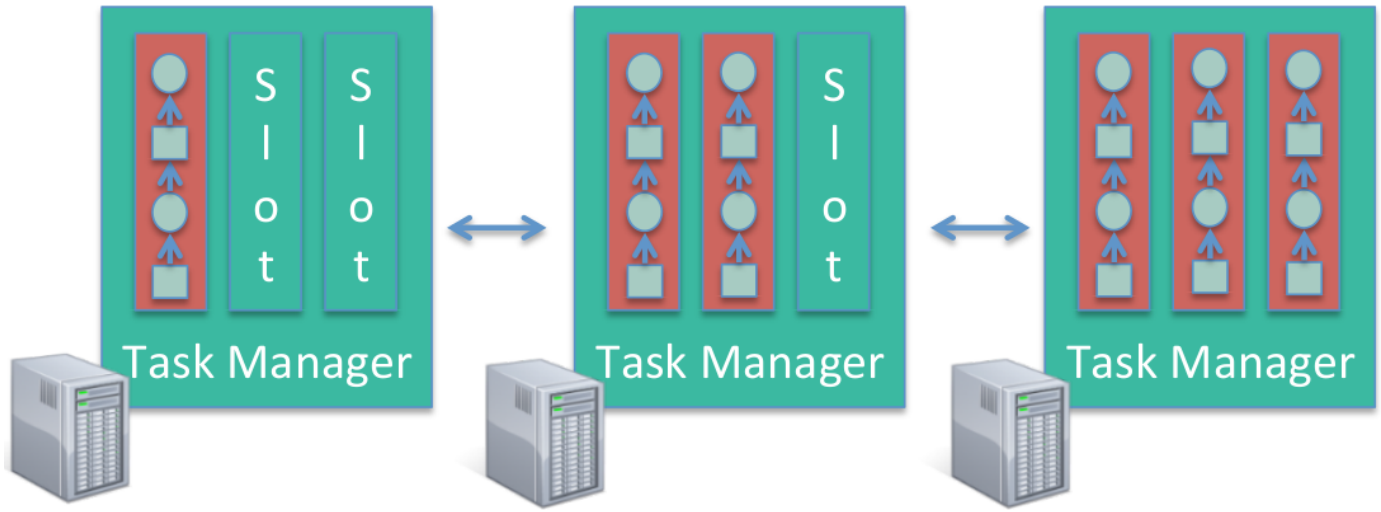
Flink's architecture: Job Manager

- **Parallelization**: Create Execution Graph
- **Scheduling**: Assign task to task managers
- **State tracking**: Supervise the execution



Flink's architecture: Task Manager

- Operations are split up into **tasks** depending on the specified parallelism
- Each parallel instance of an operation runs in a separate **task slot**
- The scheduler may run several tasks from different operators in on task slot



Flink's optimizer

- Inspired by optimizers of parallel database systems
  - Logical plan equivalences, cost models, interesting properties,...
- However, more difficult than in the relational case
  - No fully specified operator semantics due to UDFs
  - Unknown UDFs $\rightarrow$ difficult to estimate intermediate result size
  - No pre-defined schema present
- Two-phase optimization
  - Logical plan rewriting
    - Optimizer generates equivalent plans by reordering operators
  - Physical execution plan
    - Follows a cost-based approach
    - Pick strategies for data shipping and local operators
    - Keeps tracks of interesting properties (e.g., sorting, grouping and partitioning)

Example

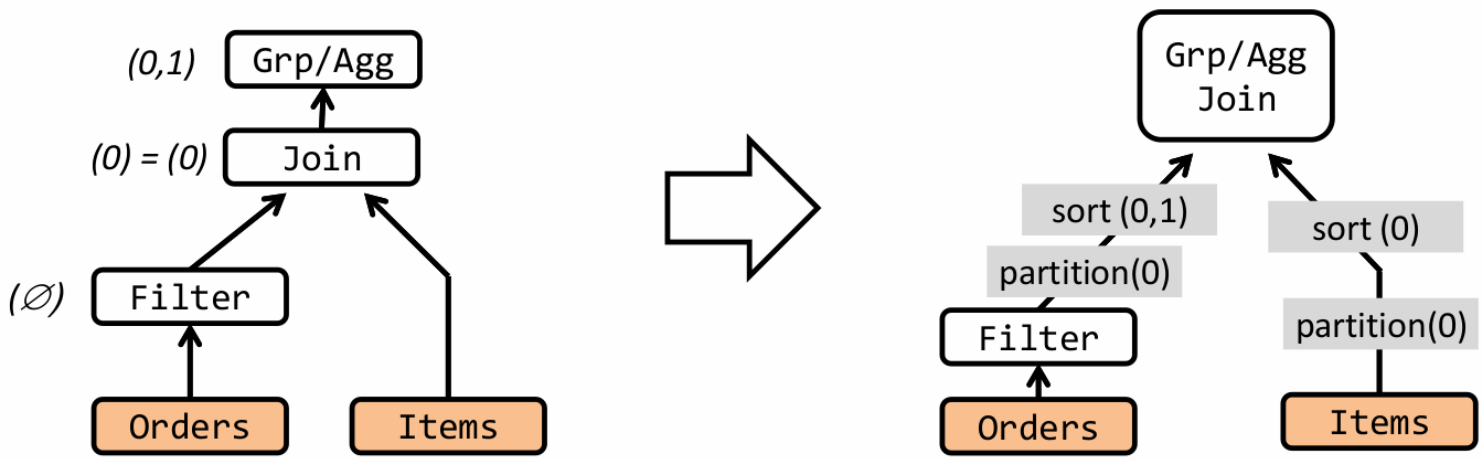
```
case class Order(id: Int, priority: Int, ...)
case class Item(id: Int, price: double, )
case class PricedOrder(id, priority, price)
```

```
val orders = DataSource(...)
val items = DataSource(...)

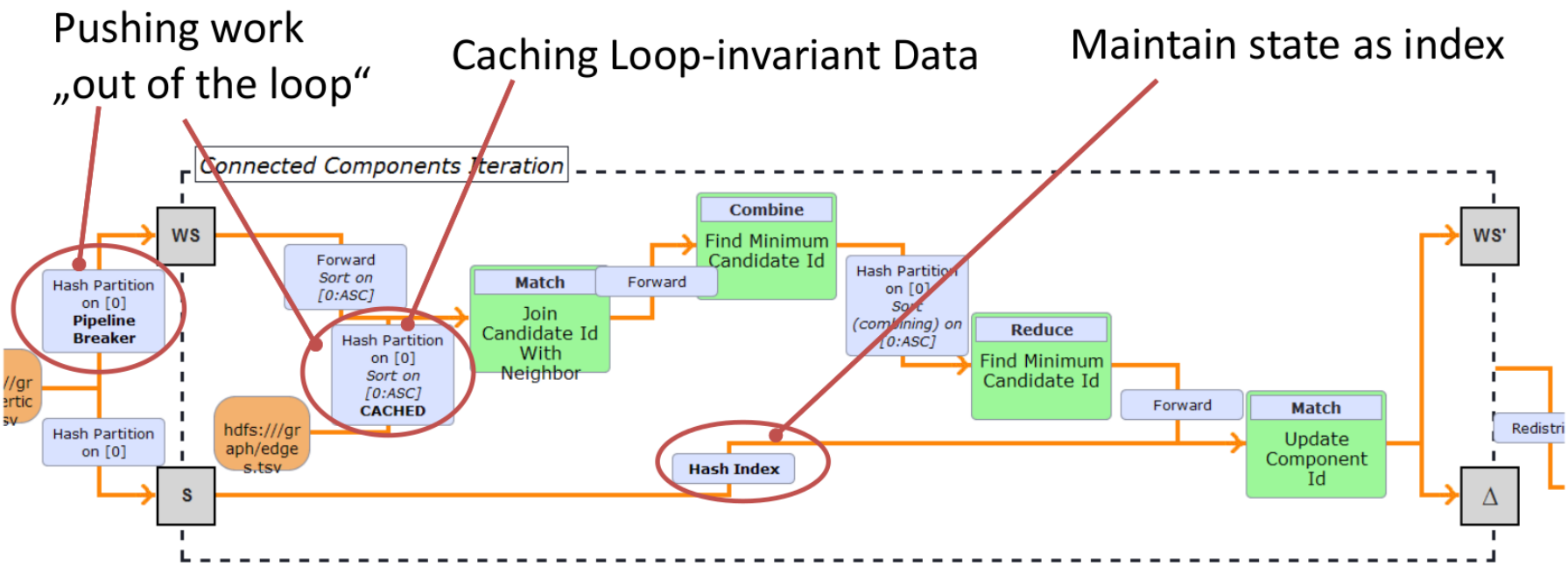
val filtered = orders filter { ... }

val prio = filtered join items where { _.id } isEqualTo { _.id }
                    map {(o,li) => PricedOrder(o.id, o.priority, li.price)}

val sales = prio groupBy {p => (p.id, p.priority)} aggregate ({_.price},SUM)
```



Optimizing iterative programs



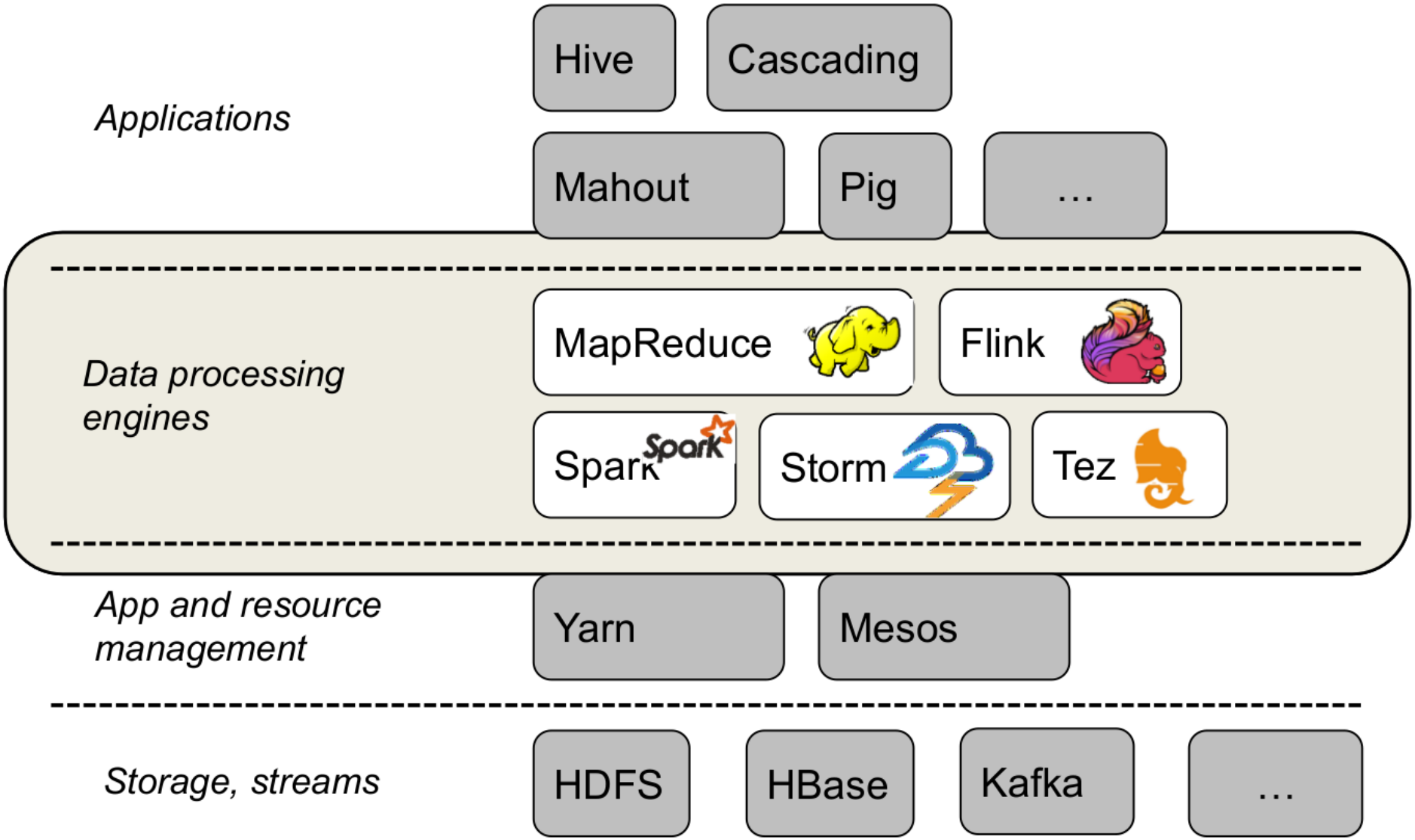
Outline

Summary (Flink)

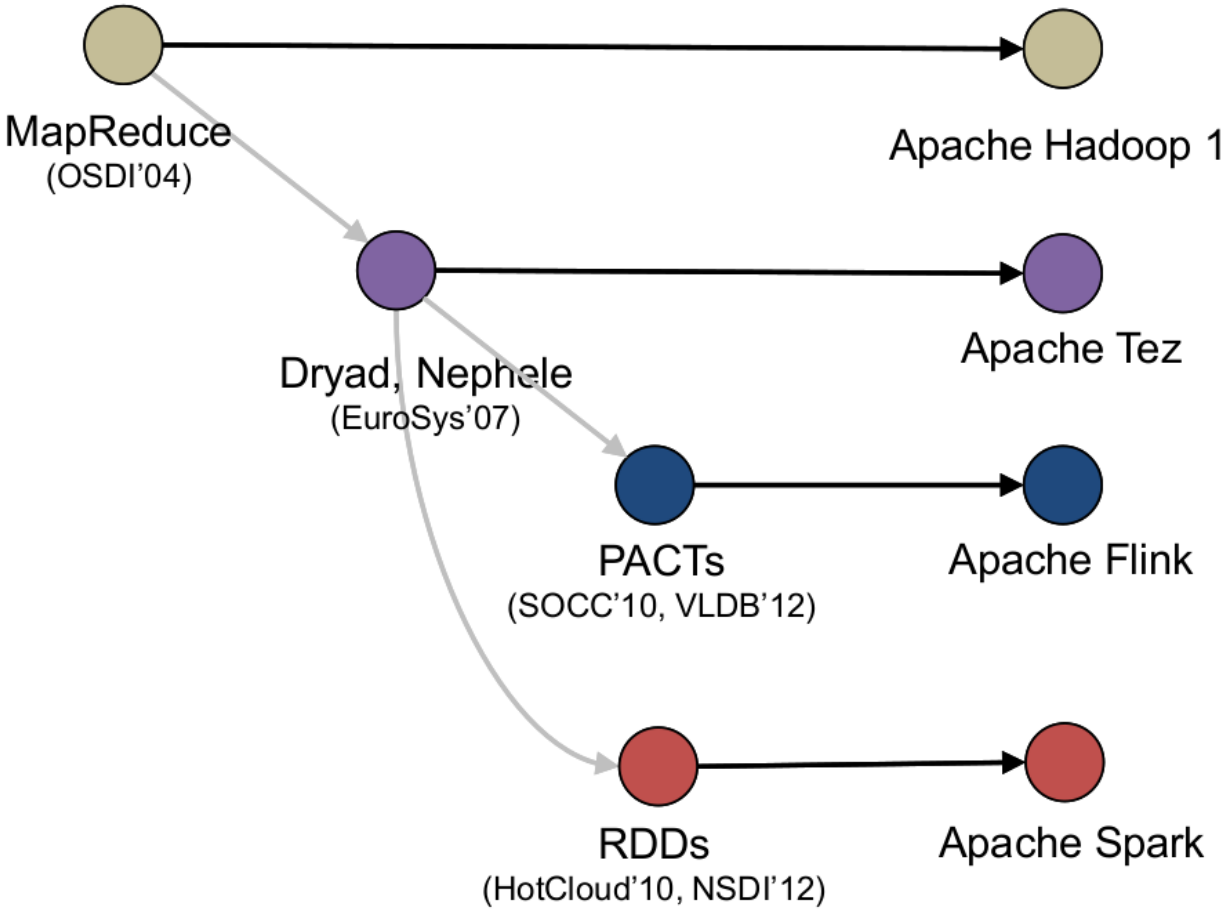
- Distributed stream processing engine
  - Can do both batch and stream processing
- Flink decouples API from execution
- Unique Flink internal features
  - Native iterations
  - Optimizer
  - Pipelined execution
- Very good performance
  - Capable of high throughput and low latency
  - Designed to run on large-scale clusters

Outline

Open source data infrastructure



Engine paradigms & systems (1)



Engine paradigms & systems (2)

- Small recoverable tasks
- Sequential code inside map & reduce functions

Dryad

- Extends map/reduce model to DAG model
- Backtracking-based recovery

- Functional implementation of Dryad recovery (RDDs)
- Restrict to coarse-grained transformations
- Direct execution of API

- Embed query processing runtime in DAG engine
- Extend DAG model to cyclic graphs
- Incremental construction of graphs

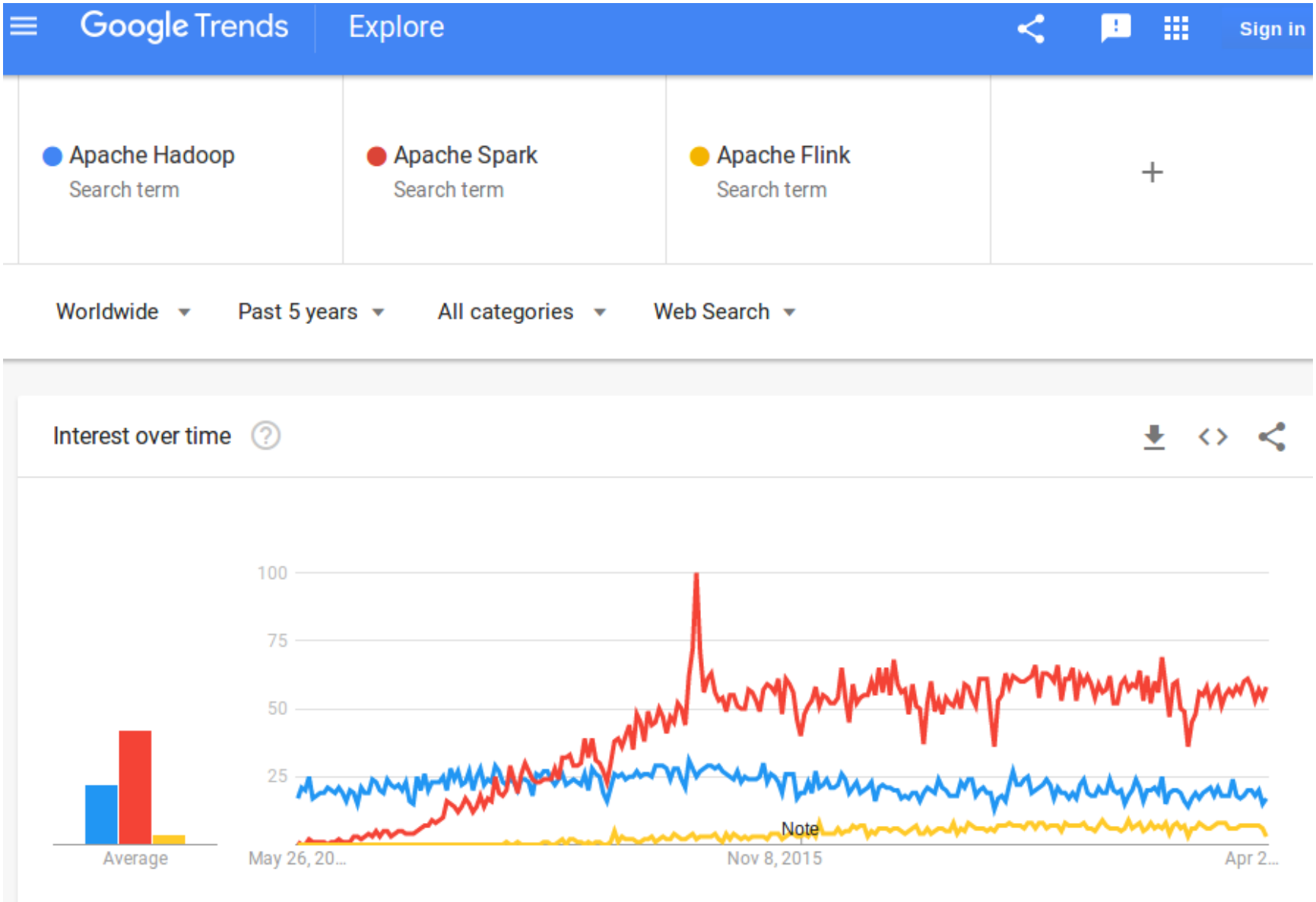
Engine comparison

			
API	MapReduce on k/v pairs	Transformations on k/v pairs collections	Iterative transformations on collections
Paradigm	MapReduce	RDD	Cyclic dataflows
Optimization	none	Optimization of SQL queries	Optimization in all APIs
Execution	Batch sorting	Batch with memory pinning	Stream with out-of-core algorithms

Apache Spark Streaming vs. Apache Flink

	Spark Streaming	Flink
Guarantee	Exactly once	
Throughput	High	
Overhead of fault tolerance	Low	
Computational model	Micro batches	Streaming
Window criteria	Time based	Time based Record based User defined
Memory management	Configured	Automatic

Google Trends (not a scientific comparison)



Acknowledgments

- Rainer Gemulla, Uni Mannheim
  CS 560, Large-Scale Data Management, 2016
- Reynold Xin, Stanford
  CS374 Guest Lecture: Apache Spark, 2015
- Peter Boncz, Hannes Muehleisen
  Large-Scale Data Engineering, 2016
- Maximilian Michels, Ufuk Celebi
  EIT ICT Summer School 2015