Introduction to dataflow engines and Apache Spark

Big Data Management

26-09-2025

photo: https://www.bmvg.de/de/service/open-data

Previously …

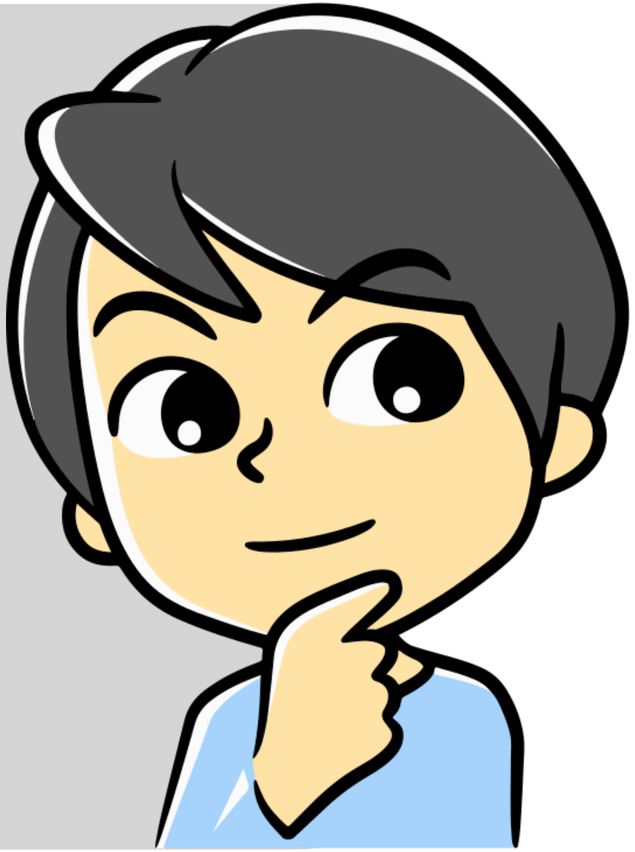
- HDFS
  - Partitioning data in fixed-size blocks
  - Replication (by default 3)
- MapReduce/Hadoop
  - Map and Reduce UDFs (user-defined functions)
  - Embarrassingly parallel programs
  - Fault-tolerant

Terminology:

- HDFS blocks
- User-defined functions (UDFs)
- Embarrassingly parallel
- Shuffling
- Data locality
- Apache foundation

Quiz Time

- Answer the questions found on LearnIT quiz under the Lecture 4 section



Today's Lecture

- ✦ **Dataflow engines paradigm**
- ✦ Introduction to Apache Spark

Problems with MapReduce

MapReduce benefits

- Automatic parallelization
- Failure handling
- Simple programming interface

MapReduce limitations

- Real-world applications often need **multiple** MR jobs
- **Programmability**:
  (i) Many MR steps, boilerplate code, spaghetti code
  (ii) Custom code even for common operations
- **Performance**:
  (i) Heavy on disk
  (ii) no optimization across multiple MR jobs

Problems with MapReduce

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

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.701.5795&rep=rep1&type=pdf

How to improve MapReduce?

- Direction 1: **Higher-level languages**
  - Write programs in a higher-level language
  - Programs are “transformed into” MapReduce job(s)
  - Addresses (mainly) **programmability**

Examples

Hive
(Facebook)


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

Pig Latin
(Yahoo)

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

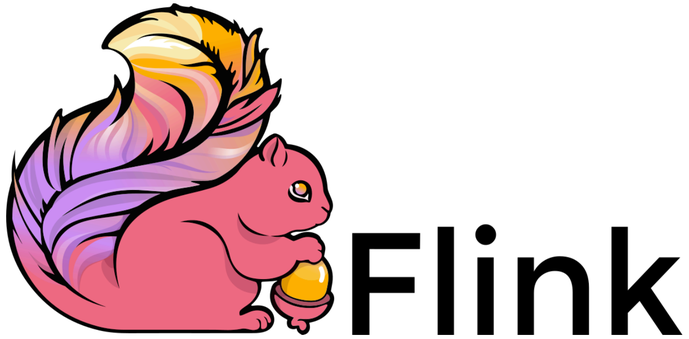
How to improve MapReduce?

- Direction 2: **Dataflow engines**
  - Build a more suitable execution environment
  - Addresses **performance**

Examples

Apache Spark
(Berkeley)

Apache Flink
(TU Berlin)



Dataflow programming

- **Imperative** program
  - Focus on control flow, data at rest
  - “Which command next?”

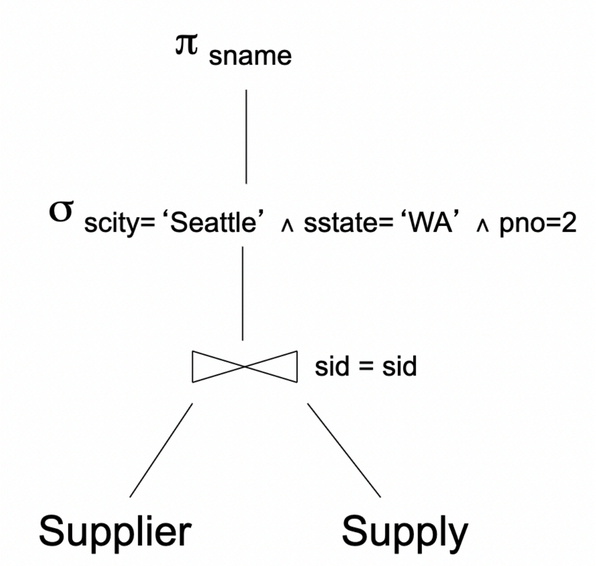
- **Dataflow** program
  - Focus on data flow, program at rest
  - “Where to put data items next?”
  - Modelled as a directed acyclic graph (DAG)
    - Vertices = operators
    - Edges = inputs and outputs
    - Operators are “black boxes”

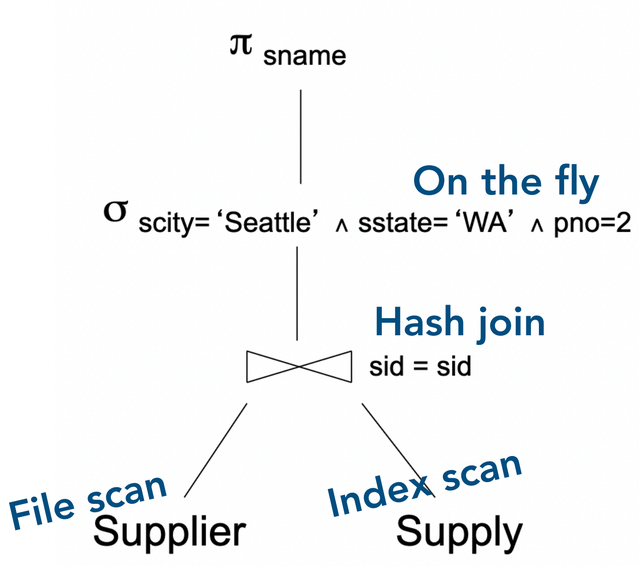
Typical optimization workflow in a database

SQL query → Logical plan → Physical plan

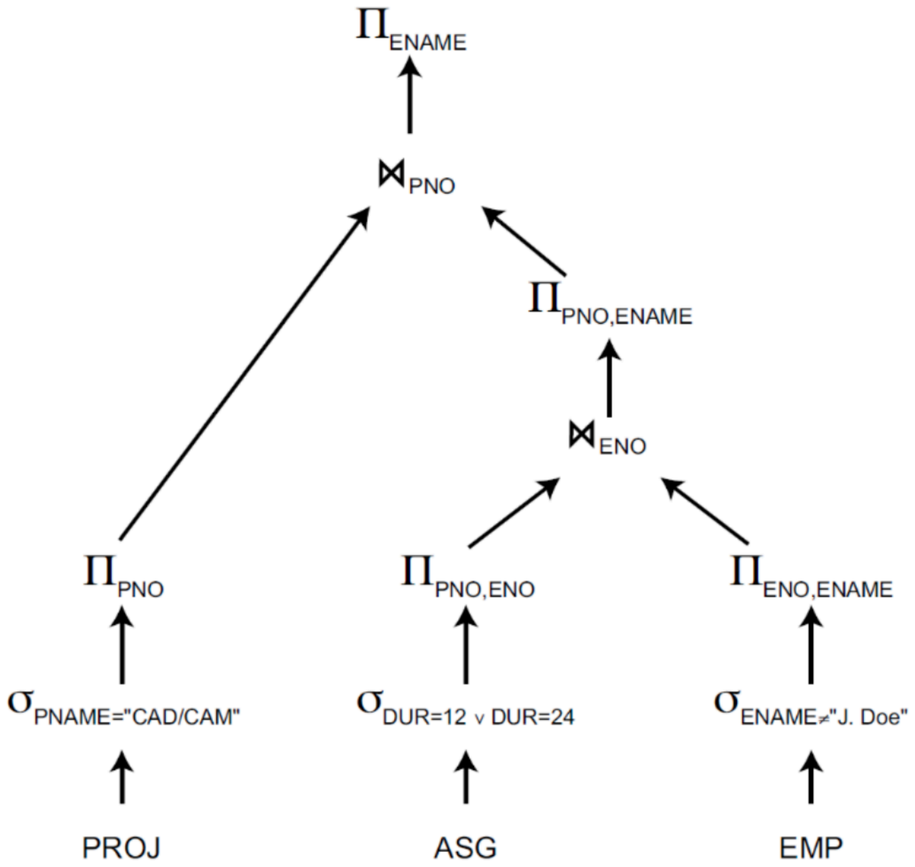
```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid and
y.pno = 2
and x.scity = 'Seattle'
and x.sstate = 'WA'
```

```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

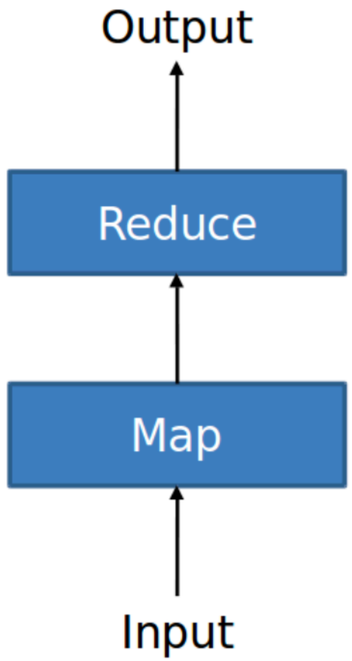




Example (logical) dataflow



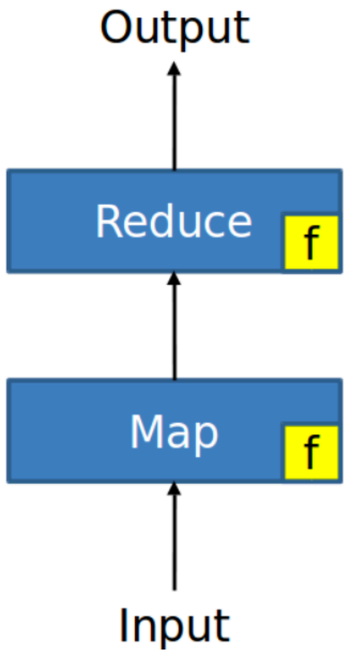
Relational logical plan



Mapreduce dataflow logical plan

MapReduce as a dataflow

- Two operators, fixed logical dataflow
  - Key: operators are **parameterized by User Defined Functions** (UDF)
- Automatic parallelization at runtime
  - Executes one of the possible **physical** dataflows (#maps, #reduces,..)



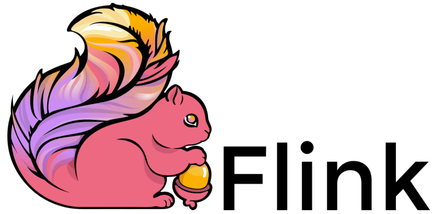
Beyond MapReduce

- **Dataflow** engines push this idea further
  - Keep UDFs
  - Add more operators
  - Improve implementation
  - Add logical/physical optimizations

Dataflow engines in a nutshell

- Programmers provide **logical dataflow**
  - Operators, function parameters, connections
  - Using a suitable higher-level language
  - Note: this is different from SQL

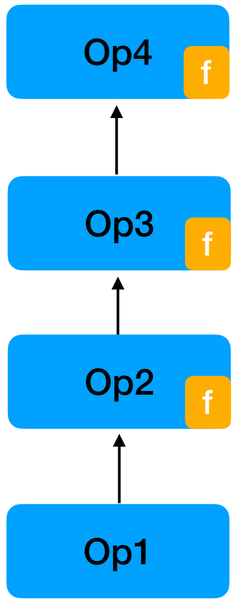
- A dataflow engine creates a **physical dataflow**
  - Includes logical optimizations
  - Includes physical optimizations

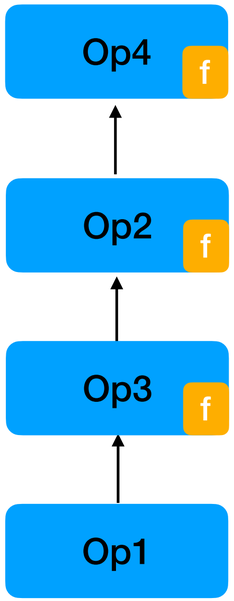


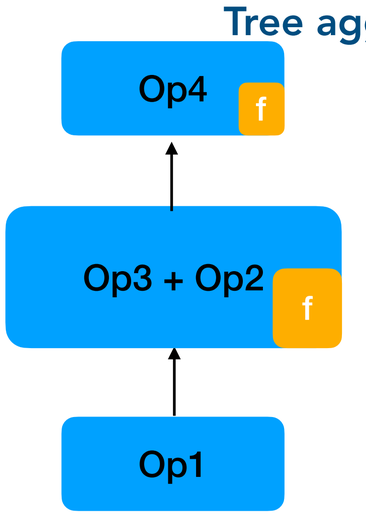
- **Executes on a cluster**
  - Automatic parallelization
  - Automatic failure handling

Typical optimization workflow in a dataflow engine

Logical plan → Optimised logical plan → Physical plan







From distributed systems to big data platforms

A bit of history

2003

Google publishes about its **cluster architecture** & **distributed file system** (GFS)

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

2004

Google publishes about its **MapReduce** programming model on top of GFS (C++, closed-source)

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
jeff@google.com, sanjay@google.com
Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution

2006

Apache & Yahoo develop Hadoop & HDFS (Java, open source)

YAHOO!

2007

Hadoop becomes an independent Apache project

hadoop

From distributed systems to big data platforms

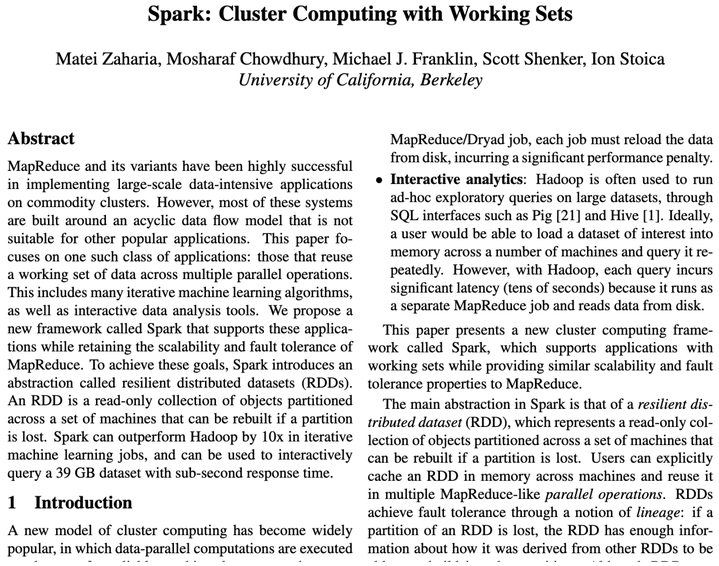
A bit of history

2008

Hadoop vendors rise

2010

Spark developed in AMPLab revisits big data processing for more **interactive** data analysis and **iterative** jobs



Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed

MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.

- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be

2014

Apache Spark becomes top level project

2017

Databricks receives a huge amount from Microsoft and gains momentum

From distributed systems to big data platforms

A bit of history

2008

Hadoop vendors rise

2010

Stratosphere project from TU Berlin started with the goal of building an easy-to-use distributed execution engine

Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing

Dominic Battré, Stephan Ewen, Fabian Hueske
Odej Kao, Volker Markl, Daniel Warneke

Technische Universität Berlin
Einsteinufer 17
10587 Berlin
Germany
firstname.lastname@tu-berlin.de

ABSTRACT

We present a parallel data processor centered around a programming model of so called *Parallelization Contracts* (PACTs) and the scalable parallel execution engine *Nephele* [18]. The PACT programming model is a generalization of the well-known map/reduce programming model, extending it with further *second-order functions*, as well as with *Output Contracts* that give guarantees about the behavior of a function. We describe methods to transform a PACT program into a data flow for Nephele, which executes its sequential building blocks in parallel and deals with communication, synchronization and fault tolerance. Our definition of PACTs allows to apply several types of optimizations on the data flow during the transformation.

The system as a whole is designed to be as generic as (and compatible to) map/reduce systems, while overcoming several of their major weaknesses: 1) The functions *map* and *reduce* alone are not sufficient to express many data processing tasks both naturally and efficiently. 2) Map/reduce ties a program to a single fixed execution strategy, which is robust but highly suboptimal for many tasks. 3) Map/reduce makes no assumptions about the behavior of the functions. Hence,

Keywords

Web-Scale Data, Cloud Computing, Map Reduce

1. INTRODUCTION

The term *Web-Scale Data Management* has been coined for describing the challenge to develop systems that scale to data volumes as they are found in search indexes, large scale warehouses, and scientific applications like climate research. Most of the recent approaches build on massive parallelization, favoring large numbers of cheap computers over expensive servers. Current multicore hardware trends support that development. In many of the mentioned scenarios, parallel databases, the traditional workhorses, are refused. The main reasons are their strict schema and the missing scalability, elasticity and fault tolerance required for setups of 1000s of machines, where failures are common.
Many new architectures have been suggested, among which the *map/reduce* paradigm [5] and its open source implementation Hadoop [1] have gained the most attention. Here, programs are written as *map* and *reduce* functions, which process key/value pairs and can be executed in many data-

2014

Flink becomes an Apache top level project and **Data Artisans** startup was founded

2018

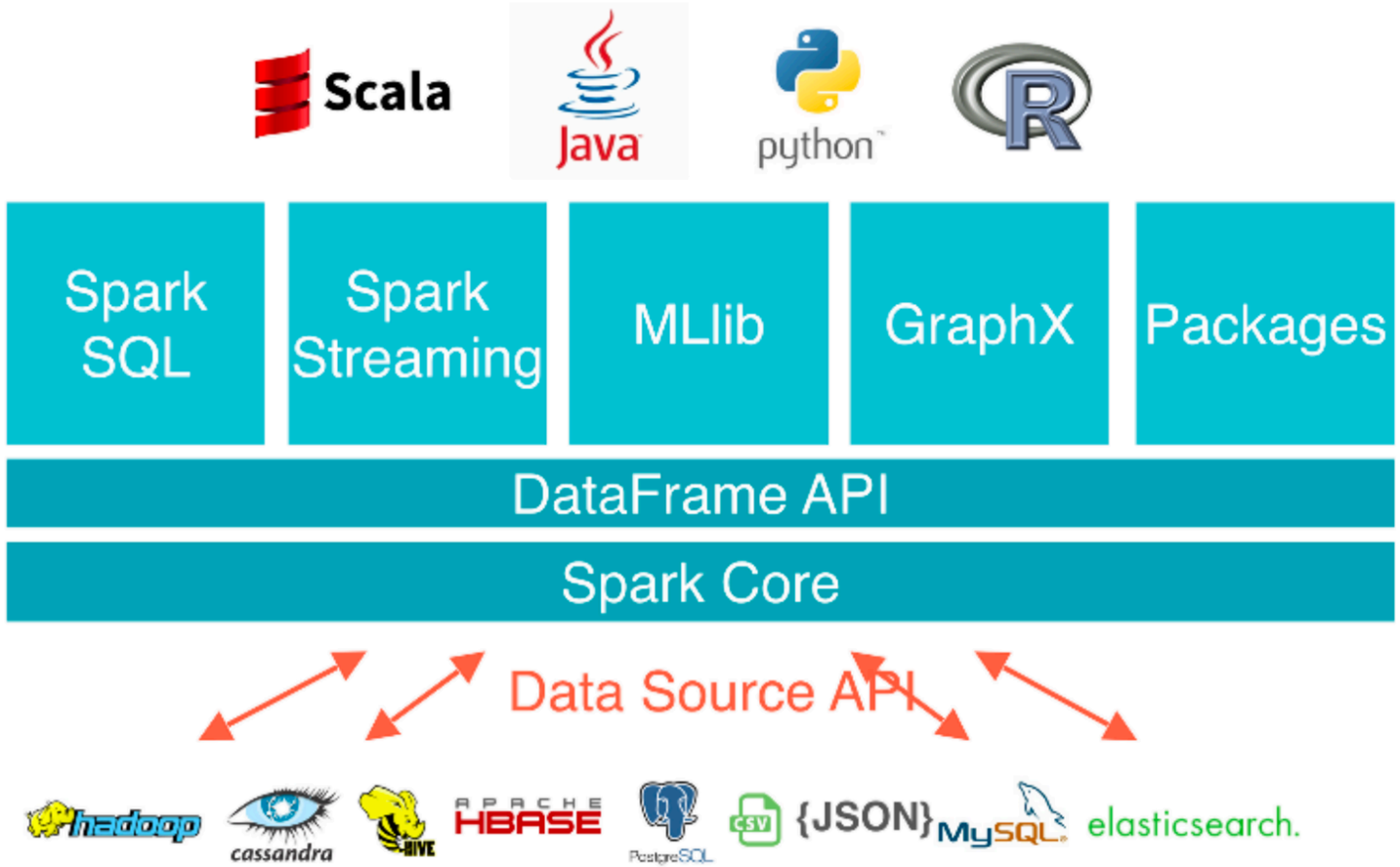
Alibaba buys Data Artisans and creates Ververica

Today's Lecture

- ✦ Dataflow engines paradigm
- ✦ **Introduction to Apache Spark**

Apache Spark

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



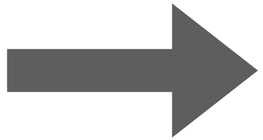
Spark's goal

- Support **iterative** jobs (common in ML algorithms)
  - Re-use datasets across parallel operations
- Support **interactive** analytics
  - Keep data in memory
- Retain MapReduce's **fault-tolerance** & **scalability**
  - Use lineage to reconstruct lost partitions
- Increase **programmability**
  - Functional programming interface

Apache Spark - programmability

```
 1 package org.myorg;
 2
 3 import java.io.IOException;
 4 import java.util.*;
 5
 6 import org.apache.hadoop.fs.Path;
 7 import org.apache.hadoop.conf.*;
 8 import org.apache.hadoop.io.*;
 9 import org.apache.hadoop.mapreduce.*;
10 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
11 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
12 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
14
15 public class WordCount {
16
17  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
18    private final static IntWritable one = new IntWritable(1);
19    private Text word = new Text();
20
21    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
22      String line = value.toString();
23      StringTokenizer tokenizer = new StringTokenizer(line);
24      while (tokenizer.hasMoreTokens()) {
25        word.set(tokenizer.nextToken());
26        context.write(word, one);
27      }
28    }
29  }
30
31  public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
32
33    public void reduce(Text key, Iterable<IntWritable> values, Context context)
34      throws IOException, InterruptedException {
35      int sum = 0;
36      for (IntWritable val : values) {
37        sum += val.get();
38      }
39      context.write(key, new IntWritable(sum));
40    }
41  }
42
43  public static void main(String[] args) throws Exception {
44    Configuration conf = new Configuration();
45
46      Job job = new Job(conf, "wordcount");
47
48    job.setOutputKeyClass(Text.class);
49    job.setOutputValueClass(IntWritable.class);
50
51    job.setMapperClass(Map.class);
52    job.setReducerClass(Reduce.class);
53
54    job.setInputFormatClass(TextInputFormat.class);
55    job.setOutputFormatClass(TextOutputFormat.class);
56
57    FileInputFormat.addInputPath(job, new Path(args[0]));
58    FileOutputFormat.setOutputPath(job, new Path(args[1]));
59
60    job.waitForCompletion(true);
61  }
62
63 }
```

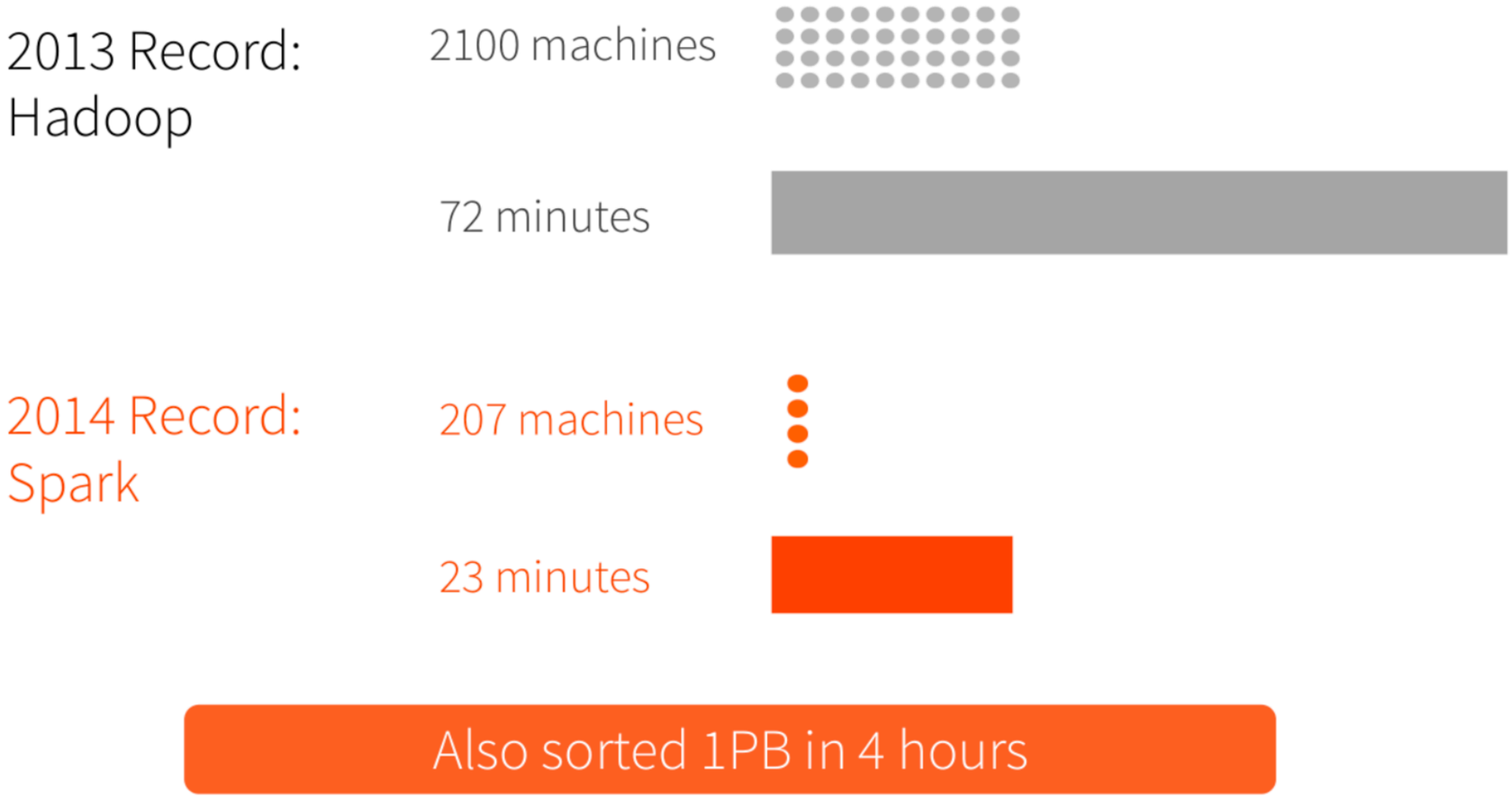
63 lines of MapReduce code



```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 with Scala API

Apache Spark - performance



https://www.slideshare.net/rxin/stanford-cs347-guest-lecture-apache-spark

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

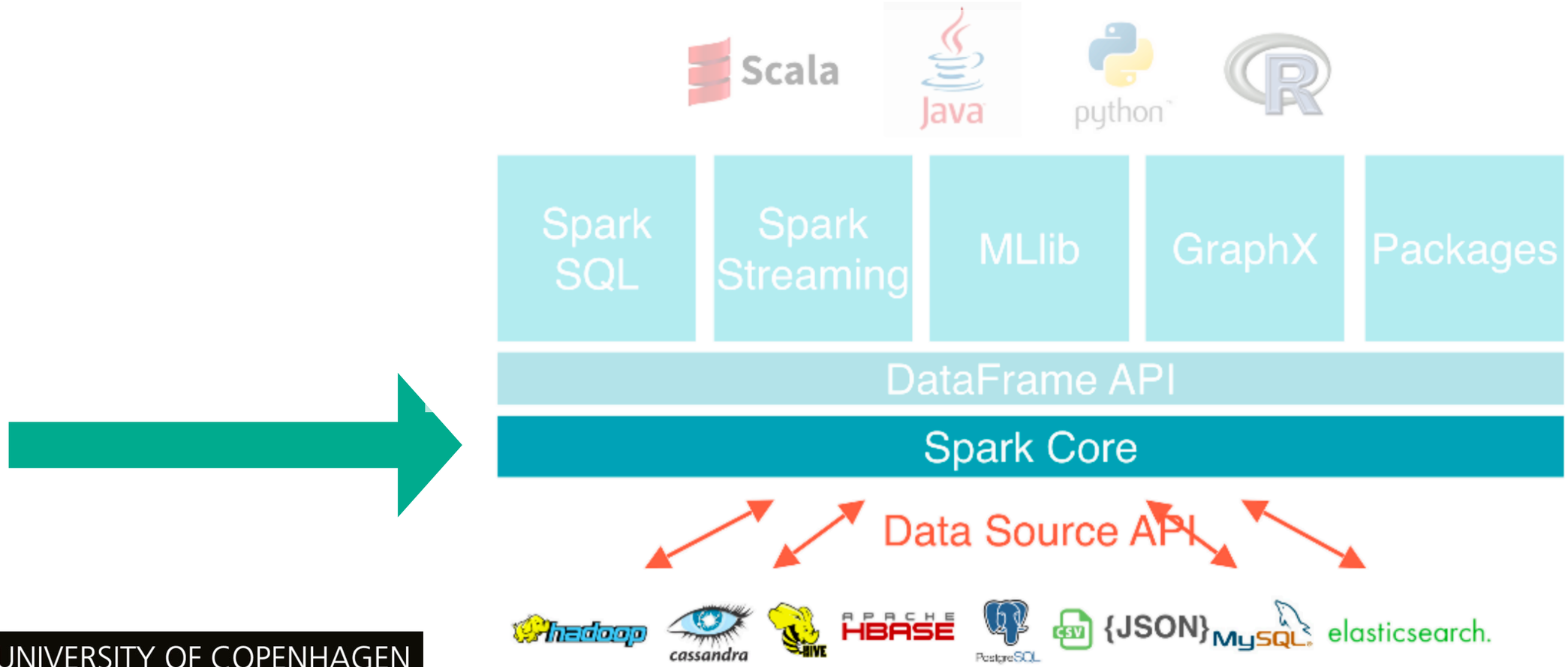
Performance

Java & Scala are faster due to static typing

…but Python is often fine

Apache Spark

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



Core abstraction: RDD

- **Resilient distributed dataset** (RDD)
  - Read-only, partitioned collections of objects (conceptually)
  - Spread across a cluster, stored in RAM, on disk, …
  - Built through parallel **coarse-grained** transformations
  - Automatically rebuilt on failure (via **lineage**)
- Write programs in terms of **distributed datasets** and **operations** on them

immutable

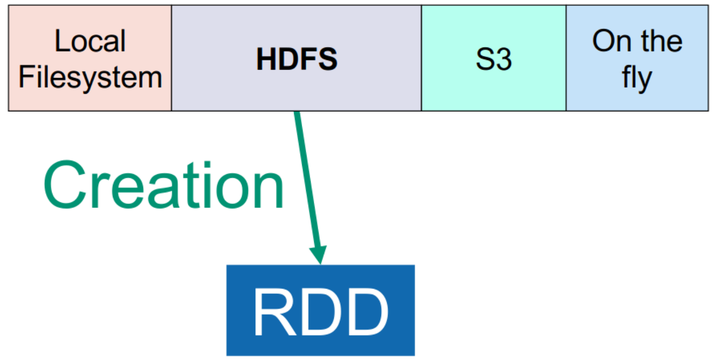
partitioned

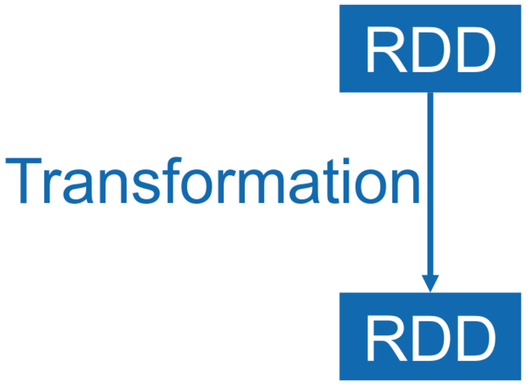
parallel

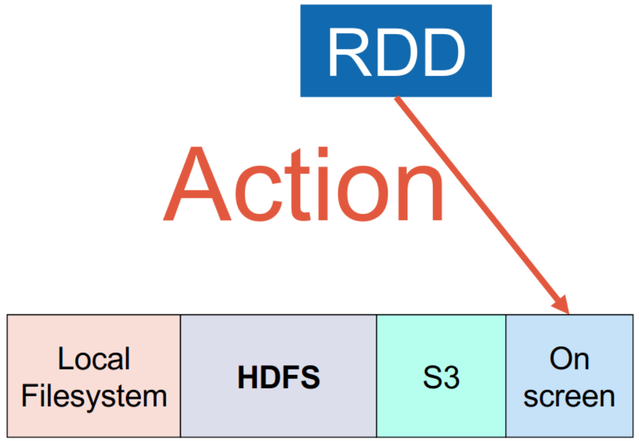
fault-tolerant

Operations

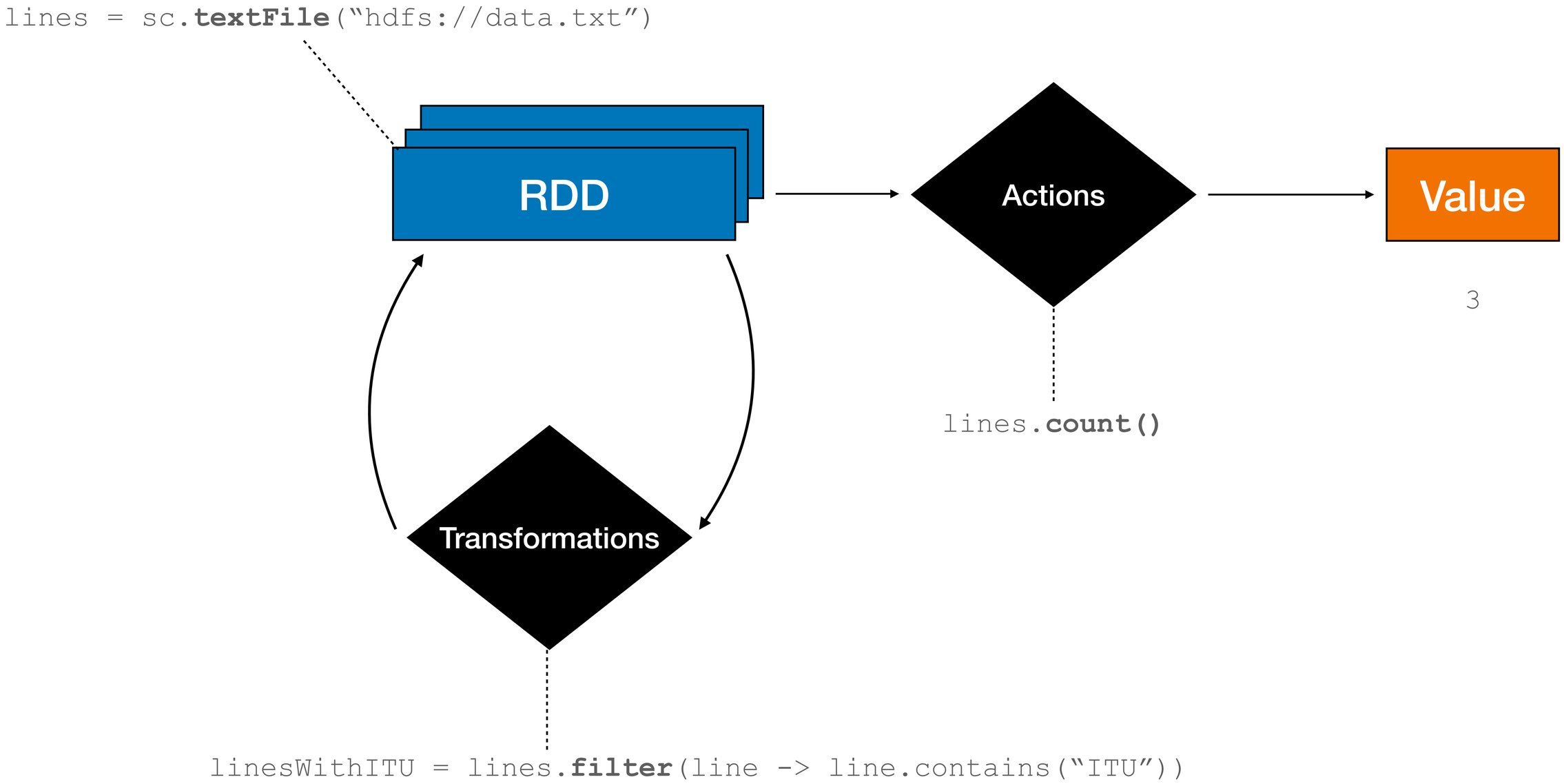
- **Creation** data -> RDD
  - Create an RDD from input data
- **Transformations** f(RDD) -> RDD'
  - Lazy execution (not computed immediately)
  - Eg., map, filter, groupBy, …
- **Actions** f(RDD) -> object/value/action
  - Triggers computation
  - Eg., count, collect, save, …







Example with RDDs



Creating RDDs

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

```
List<Integer> data = Arrays.asList(1, 2, 3, 4);
JavaRDD<Integer> numbers = sc.parallelize(data);
```

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

```
JavaRDD<String> lines = sc.textFile(”hdfs://data.txt”)
```

3. From RDDs

```
JavaRDD<Integer> lineLengths = lines.map(s -> s.length)
```

Transformations (I)

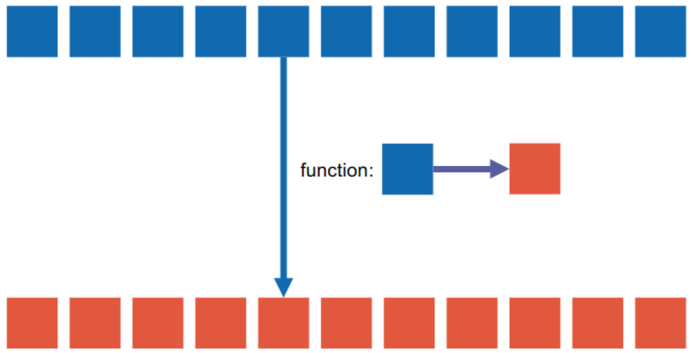
- ✦Takes an RDD and produces another RDD
- ✦ Computed lazily (i.e., not immediately but when required)
- ✦ Example: map()

```
JavaRDD<String> lines = sc.textFile(”hdfs://data.txt”)
JavaRDD<Integer> lineLengths = lines.map(s -> s.length)
```

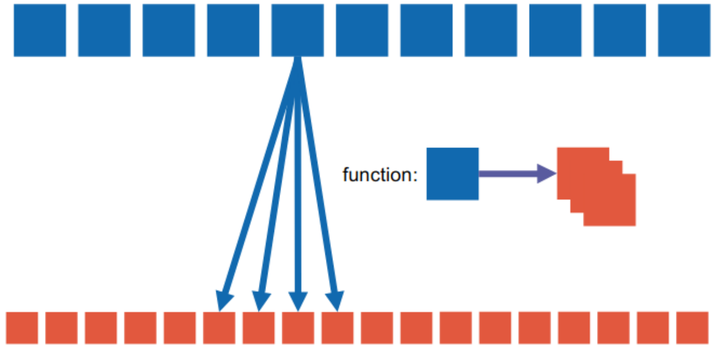
Transformations (II)

- **Map**:
- Parameterized by a UDF f :T → U
- Input is a single value (of type T), not a key-value pair
- Output exactly one value (of type U), not a collection of k/v pairs
- Cf. **flatMap** transformation: 1 input value; 0, 1, or more output values

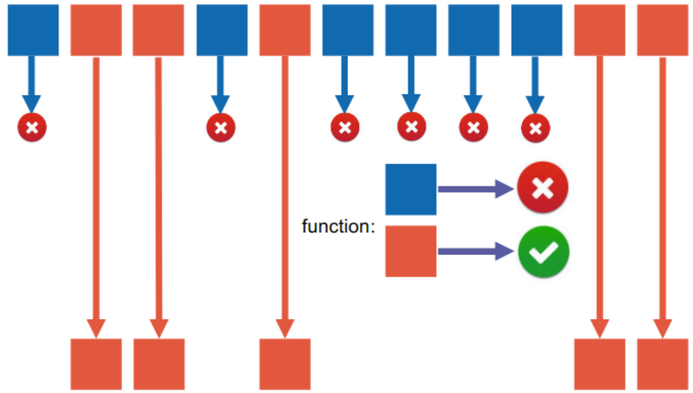
Transformations (III)



map ( )

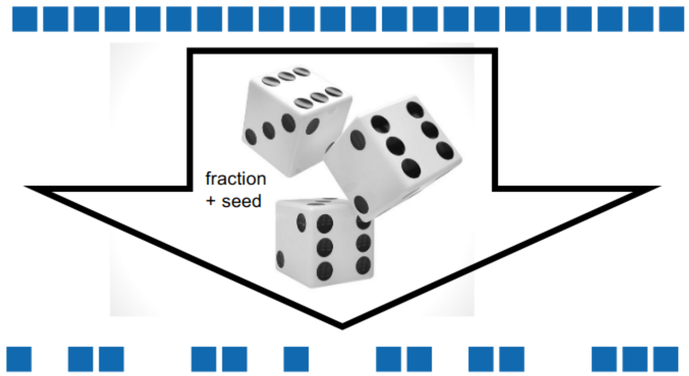


flatmap ( )

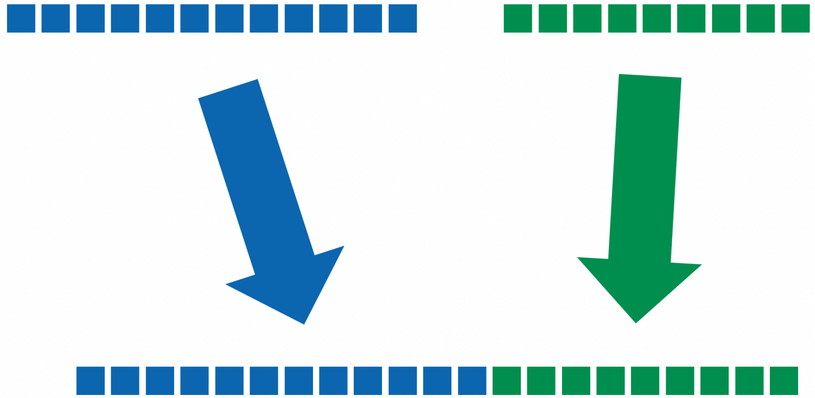


filter ( )

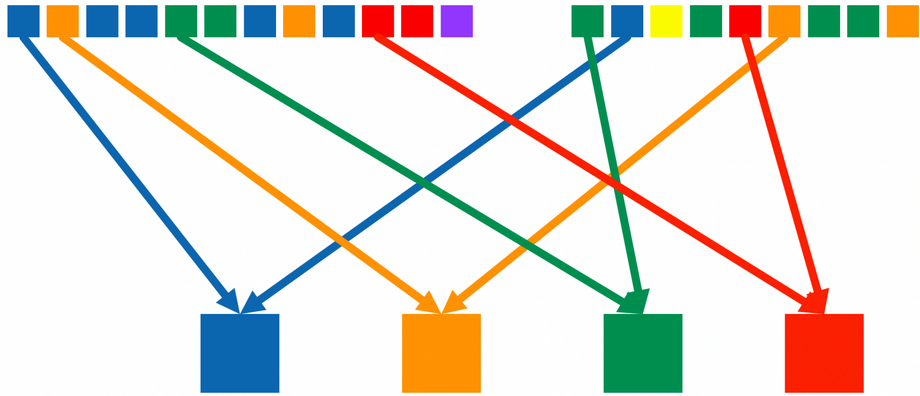
unary



sample ( )



union ( )



intersect ( )

binary

Transformations (IV)

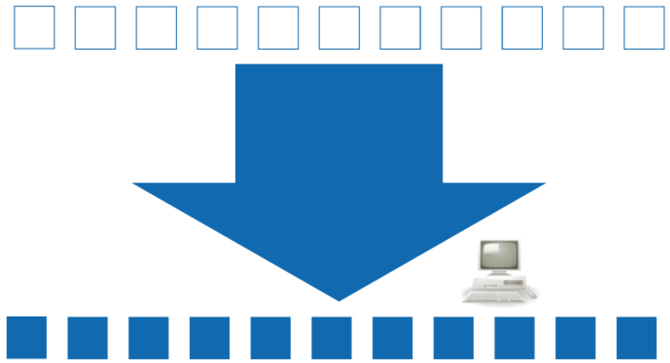
- **groupByKey**
  - No input UDF
  - Works on key-value pairs
  - (K, V) → (K, Iterable[V])
- **reduceByKey**
  - Parameterized by UDF f: (V, V) → V
  - Works on key-value pairs
  - (K, V) → (K, V)

Actions (I)

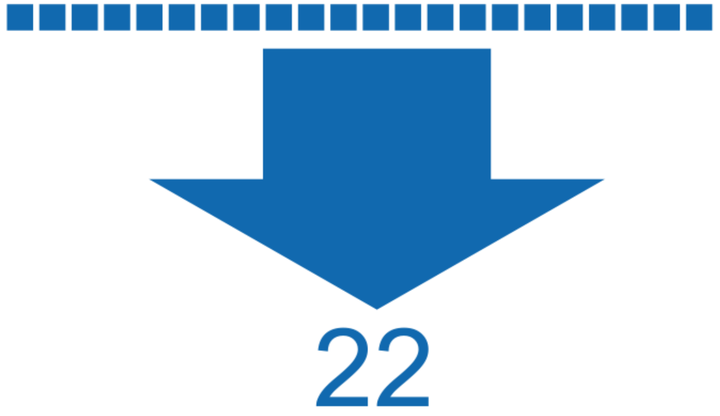
- Takes an RDD and returns a value (to driver program, to external storage, ...)
- Triggers computation
- Examples: reduce(), collect()

```
JavaRDD<String> lines = sc.textFile(”hdfs://data.txt”)
JavaRDD<Integer> lineLengths = lines.map(s -> s.length)
Integer totalLength = lineLengths.reduce((a,b) -> a+b)
```

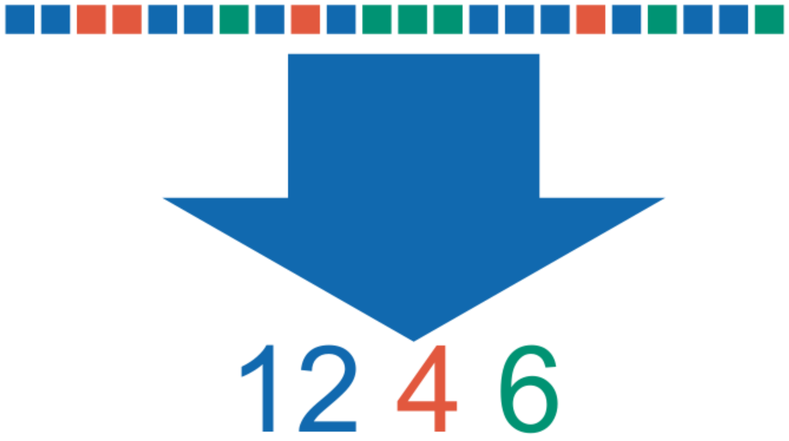
Actions (II)



collect ( )



count ( )



countByValue ( )

Working with RDDs - Example (I)

Input

One ring to rule them all, one ring to find them.
One ring to bring them all and in the darkness bind them.

```
Collection<String> output = sc.textFile(”hdfs://data.txt”)
.collect()
```

Output

Collection (“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 - Example (II)

Input

One ring to rule them all, one ring to find them.
One ring to bring them all and in the darkness bind them.


```
Collection<Integer> output = sc.textFile(”hdfs://data.txt”)
.map(s -> s.length)
.collect()
```


Output

Collection (49, 57)

Working with RDDs - Example (III)

Input

One ring to rule them all, one ring to find them.
One ring to bring them all and in the darkness bind them.

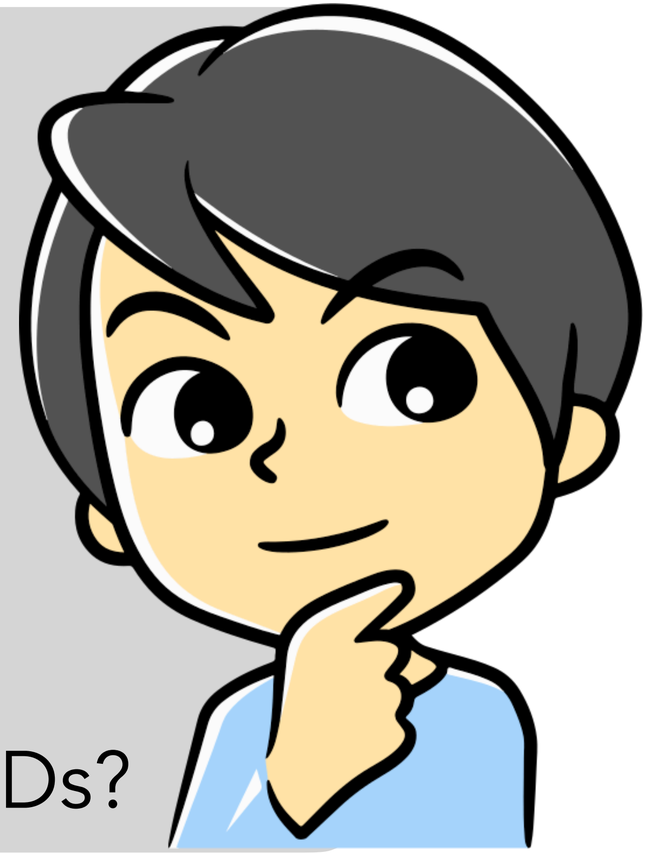
```
Integer output = sc.textFile(”hdfs://data.txt”)
.map(s -> s.length)
.reduce((a,b) -> a+b);
```

Output

106

Reflection time

- Compute the total price of the products for each product type available in store 10.
  - `prodId, prodName, prodType, price, storeId`
  - ```
    select prodType, sum(price)
    from products
    where storeId = 10
    groupby prodType
    ```
- How you would implement it in Spark using RDDs?


    ```

A solution

- Schema: `prodId, prodName, prodType, price, storeId`

- SQL query:

```
select prodType, sum(price)
from products
where storeId = 10
groupby prodType
```

```
JavaSparkContext sc = new JavaSparkContext(sparkConf);


lines = sc.textFile("hdfs://…products")
  .map(s -> new Record(s.split(“,”)))
  .filter(rec -> rec[4]=10)
  .mapToPair(rec -> new Tuple2(rec[2],rec[3]))
  .reduceByKey((a, b)-> a+b)
  .collect()
```

Summary

- Dataflow engines
  - Performance
  - Programmability
- Apache Spark
  - Resilient Distributed Datasets
  - Transformations vs Actions

In the next lecture …

- ✦ Apache Spark under the hood
- ✦ Libraries on top of RDDs

Readings

- Papers:
  - Spark: Cluster Computing with Working Sets. Hotcloud 2010
  - Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 2012