Big Data Processsing

Rainer Gemulla, Simon Forbat

(Tage der Informatik, 2025)

About us

- Prof. Dr. Rainer Gemulla
- Simon Forbat
- University of Mannheim
  - Chair of PI1: Data Analytics
  - Part of the Data and Web Science Group
- Research
  - Machine learning with structured data (such as relational data)
  - Machine learning with semi-structured data (such as multi-relational graphs)
  - Combining the above with unstructured knowledge (such as text)
  - **Efficient and scalable methods and systems for data-intensive processing**

Today

- Big data processing for data analytics
  - Key concepts and fundamentals
  - MapReduce and Apache Spark
- PySpark, the Python interface to Spark
  → workshop
- Slides are partly taken from lecture on “Large-scale data management”
  - We’ll omit many details, but I kept them in the slides for your reference

Overview

1. Background
2. MapReduce
3. Dataflows
4. Summary

Data grows

- Netflix streaming platform processes trillions of events & petabytes of data per day (2018)
- Facebook warehouse stores >300PB of data, +600TB each day (2014)
- Airbnb Hadoop cluster stores >11PB of data (2016)
- Pinterest transports >800 billion messages and >1.2PB of data per day to its warehouse (2018)
- Twitter's warehouse stores >500B of data and handles 10s of millions of writes / second (2017)
- 10s of trillions of websites (Google index: >100PB)
- Billions of triples in the Linked Open Data Cloud
- ...

The Need for Parallelism

If we'd just use one machine:

- Too slow
- Too little storage
- Too expensive
- Availability and reliability difficult
- Network overload
- No elasticity
- …

But parallelization / distributed programming is difficult. So what do we do?

Concept: DSL

- You know: Relational database management systems (RDBMS)
  - Write a SQL query (“easy”):
    ```
    SELECT year, sum(total)
    FROM sales
    GROUP BY year
    ```
  - Let a (parallel) RDBMS handle the rest (“hard“)
- SQL is an example of a **domain-specific language (DSL)**
  - Designed for a particular application domain
  - Here: relational data management
- Mature technology (since 70s), queries & transactions, efficient and scalable

Example: Oracle Exadata DB Machine

- Up to 2,880 CPU cores per rack for database processing
- Up to 33 TB memory per rack for database processing
- Up to 1,088 CPU cores per rack dedicated to SQL processing in storage
- Up to 21.25 TB of Exadata RDMA Memory per rack
- 100 Gb/sec RoCE Network
- Complete redundancy for high availability
- From **2 to 15 database servers** per rack
- From 3 to 17 storage servers per rack
- Up to 462.4 TB of performance-optimized flash capacity (raw) per rack
- Up to 2 PB of capacity-optimized flash capacity (raw) per rack
- Up to 4.2 PB of disk capacity (raw) per rack

Oracle Exadata Databas
Machine X10M

It‘s not “just“ volume, it‘s Big Data

> *Big data is **high-volume**, **high-velocity** and **high-variety** information assets that demand **cost-effective**, **innovative** forms of information processing for **enhanced insight and decision making**.* – Gartner

Gartner’s 3Vs

- **V**olume
  - Large amount of data
- **V**elocity (speed)
  - New data is arriving fast
- **V**ariety (diversity)
  - Data has **many formats and shapes**
  - Relational data, text, tweets, graphs, images, XML, …
  - Variety of **complex tasks to solve**
- In short: big, fast, diverse

Compute grows

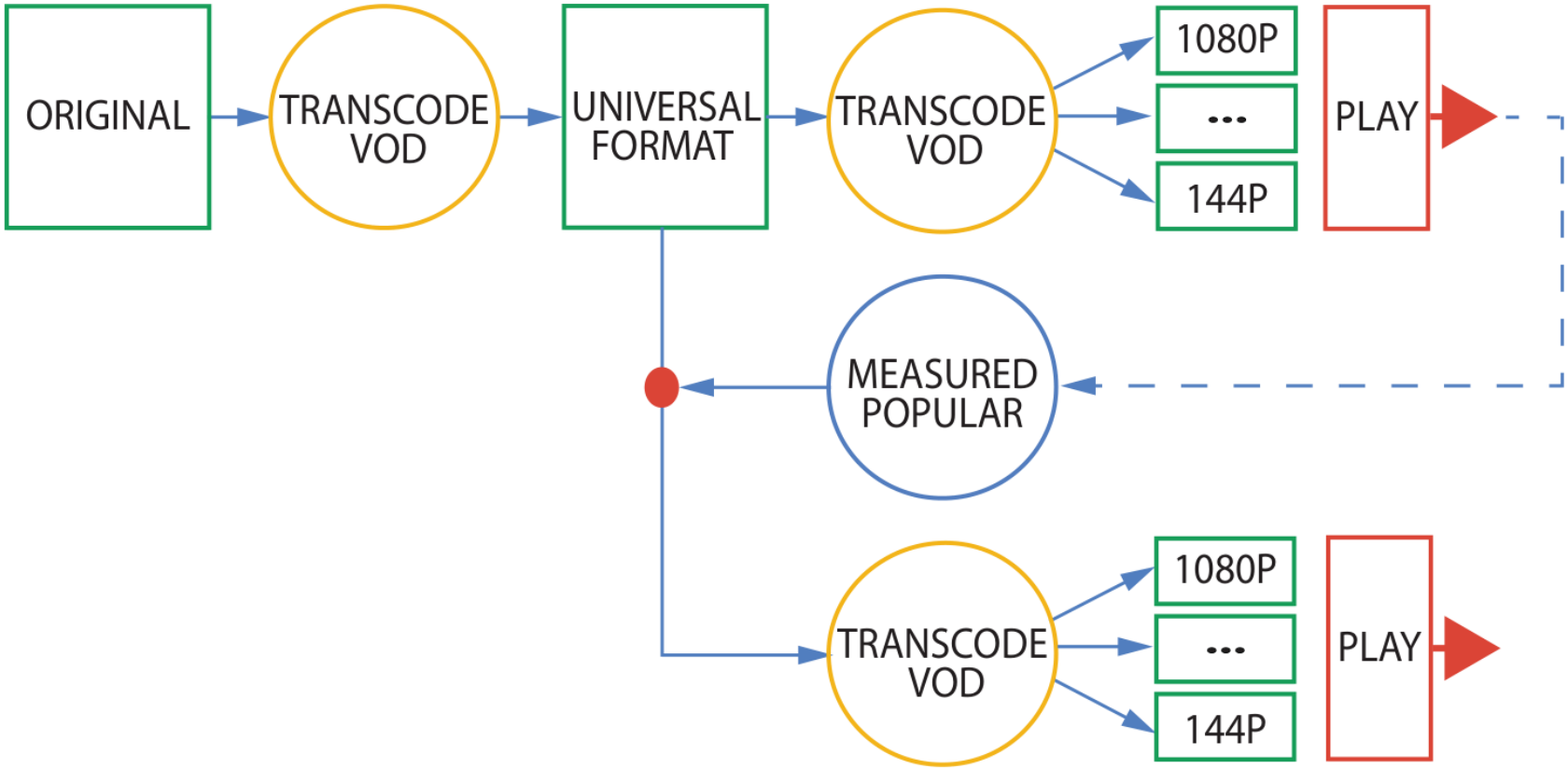


Figure 2.5: The YouTube video processing pipeline. Videos are transcoded multiple times depending on their popularity. VOD = video on demand.

Google Datacenter

Concept: Warehouse-Scale Computer

- **Warehouse-Scale Computer** = 10000s of servers acting as a single large "computer"
  - Internet services, cloud computing, …
  - Performance/$ important
- Large amounts of storage and compute available
- How to use such infrastructures for data-intensive processing? Our focus:
  - From relational data to "arbitrary data"
  - From SQL queries to "programs"
  - From 10s of machines to 10000s of machines

Overview

1. Background
2. MapReduce
3. Dataflows
4. Summary

MapReduce

- A programming model and an implementation
  - Introduced by Google in 2004
- Goal 1: **scale out**
  - To very large datasets
  - To very large clusters of *commodity machines*
  - To a large number of users
- Goal 2: **simple to use** (for programmers)
  - Two functions: Map and Reduce
  - Abstract away data placement, parallelization, failure handling, cost models, …

Concept: Functions as Values

- Many programming languages allow **functions as values** (“first-class citizens”)
- Python:

```
def run(f):
    print(f(1), end="")

def g(x):
    return "Hello "

run(g)
run(lambda x: f"World {x}!")
```

- Ouput: `Hello World 1!`

MapReduce in a Nutshell

- Framework models common data analysis pattern
  1. Read collection of **data items**
  2. Process each data item *individually* (**Map**)
  3. **Shuffle** and group data
  4. Aggregate each *group* of data items (**Reduce**)
  5. Write new collection of data items
- Operates on (key, value)-pairs
- MapReduce fixes this computational structure
- But Map and Reduce are user-specified functions
  $\rightarrow$ used to express actual task at hand

Example: Counting words

- Count how often each word occurs in a text collection
- Map function
  - Extracts words from single document
  - (document id, text) -> list(word, freq)
  - Output list has one element per word in text
- Shuffle: MapReduce then groups counts per word
- Reduce function
  - Aggregates counts of word
  - (word, list(freq)) -> list(word, total freq)
  - Here: output list has just one element

Counting words: Map

Input (1, “One ring to rule them all, one ring to find them.”)
(2, “One ring to bring them all and in the darkness bind them.”)

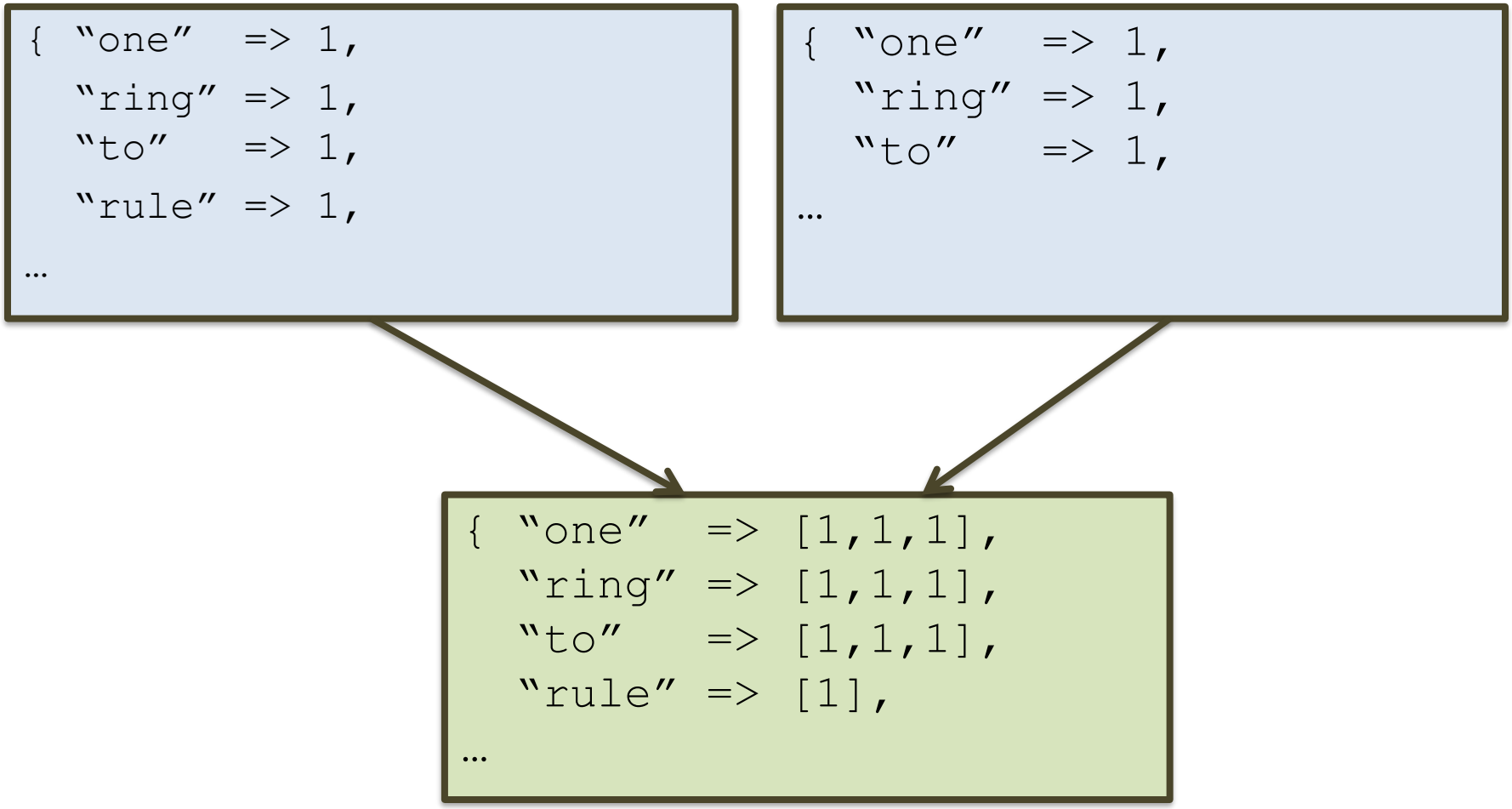
Line 1

```
{ “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “rule” => 1,
  “them” => 1,
  “all”  => 1,
  “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “find” => 1,
  “them” => 1 }
```

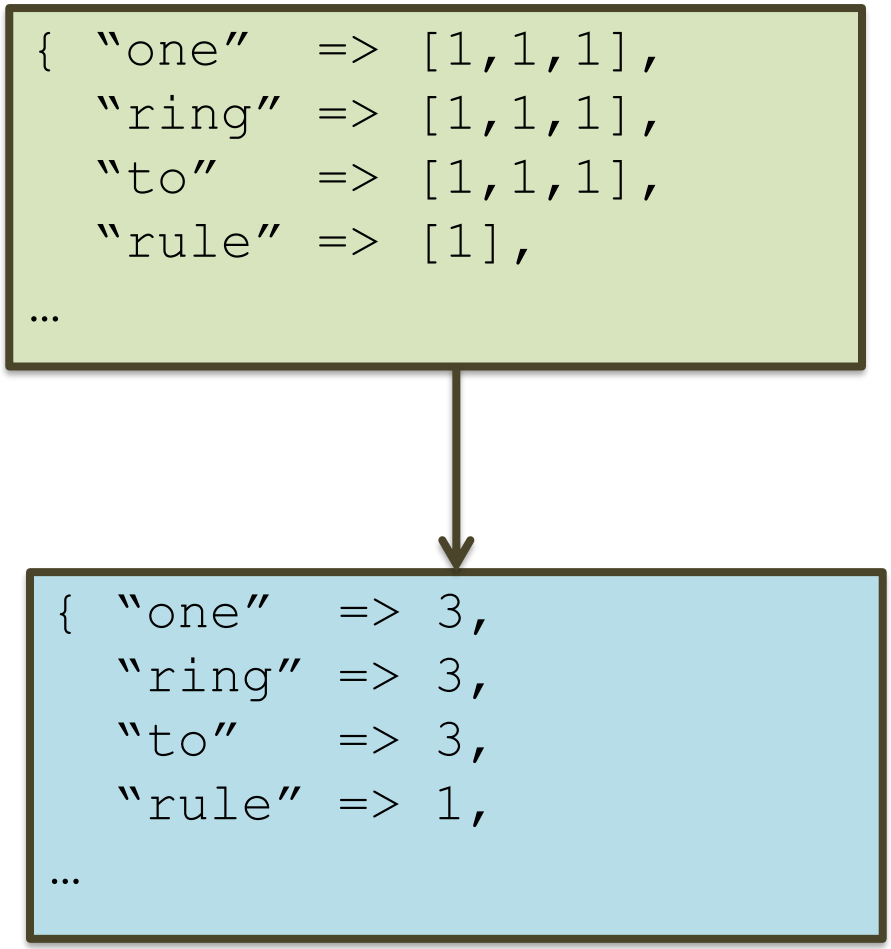
Line 2

```
{ “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “bring”=> 1,
  “them” => 1,
  “all”  => 1,
  “and”  => 1,
  “in”   => 1,
  “the”  => 1,
  “darkness” => 1,
  “bind” => 1,
  “them” => 1 }
```

Counting words: Group by word



Counting words: Reduce

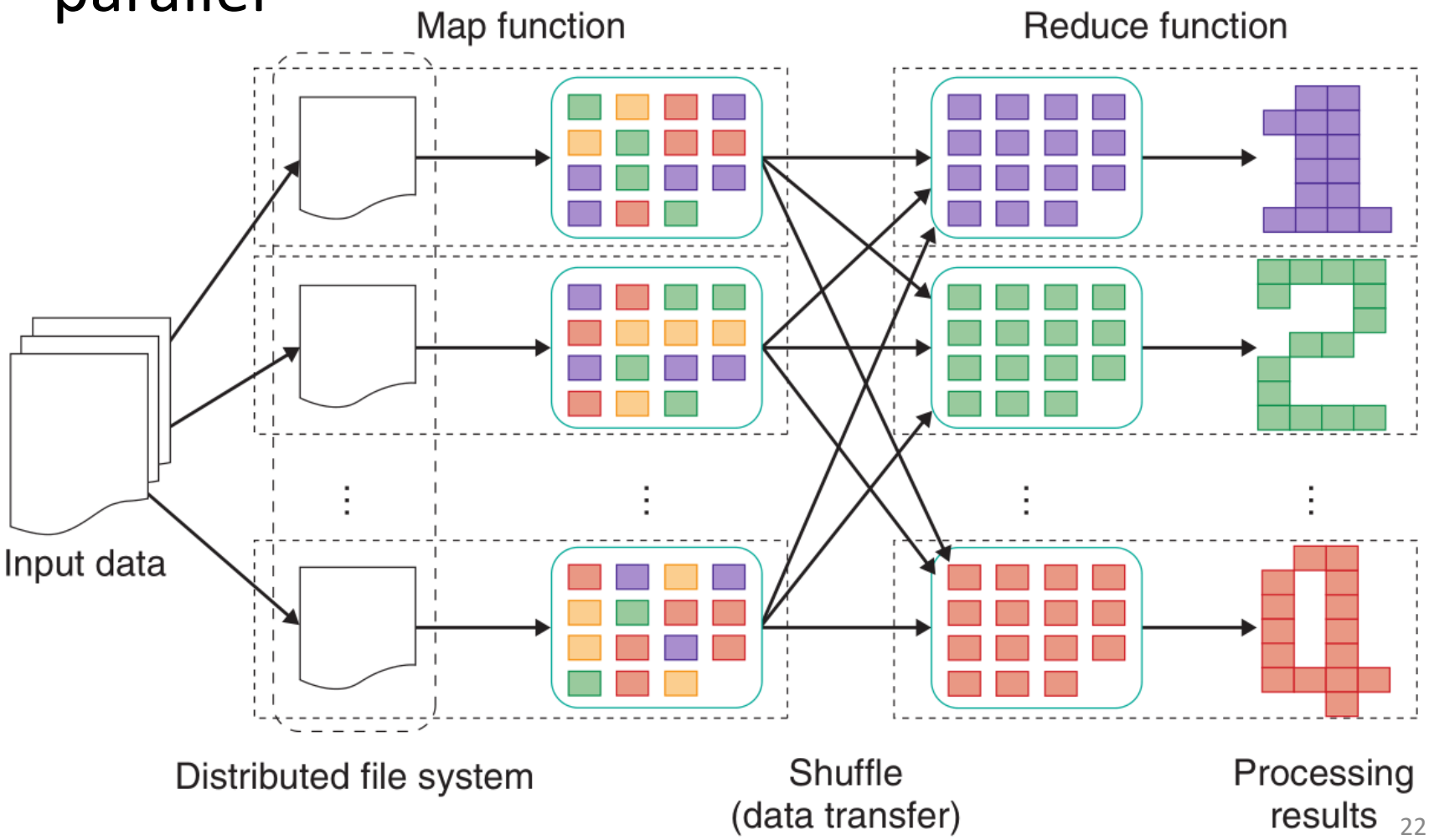


Counting words: Pseudo code

```
Map(Integer docId, String text):
        for each word w in text:
                emit(w, 1)


Reduce(String word, Iterator<Integer> counts):
        int result = 0
        for each count in counts:
                result += count
        emit(word, result)
```

A glimpse at parallelization

Different Map and Reduce invocations can run in parallel



Key ideas of MapReduce runtime

1. **Partition data** across a cluster of machines
   - E.g., via a distributed file system (GFS / HDFS)
   - Allows reading multiple partitions simultaneously
   - More machines, more read throughput
2. **Ship code to data**
   - Machines that store data also run Map/Reduce functions
   - Workers run Map function on their local part of the data → no need to communicate (large) data
   - Saves communication bandwidth
3. **Query fault tolerance**
   - In case of failure, don't rerun from scratch
   - Instead: repeat only parts that affected by failure
   - Essential for scale out / long running tasks

Examples

- Map-only
  - E.g., text processing (such as sentiment detection)
  - E.g., image/video processing (such as conversion)
  - E.g., data preprocessing (as part of ETL)
- +Reduce
  - Adds grouping facilities
  - Original app: Build an inverted index
    - Input: (website, { words })
    - Output: (word, { websites })
  - E.g., run SQL-like queries
  - E.g., data mining and machine learning

Overview

1. Background
2. MapReduce
3. Dataflows
4. Summary

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

Beyond MapReduce (1)

Idea 1: **Higher-level languages**

- Write programs in a suitable higher-level language
- Programs are “compiled into” MapReduce jobs
- Addresses (mainly) programmability

Early examples

Hive
(Facebook)


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

Pig
(Yahoo)

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

Beyond MapReduce (2)

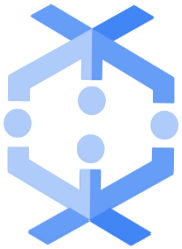
Idea 2: From MapReduce to **dataflow engines**

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

Examples

Spark

Flink



Google Cloud Dataflow

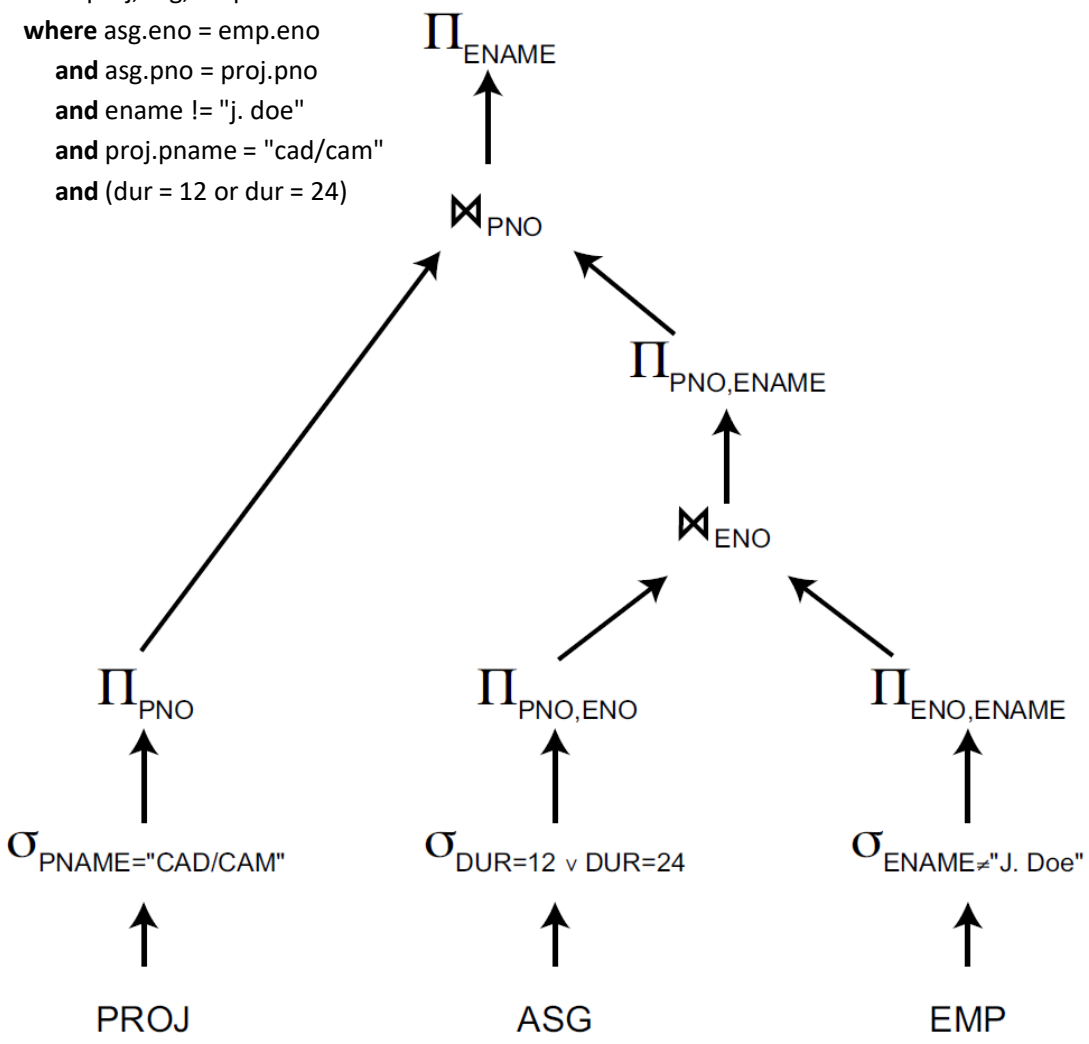
Beam (API)

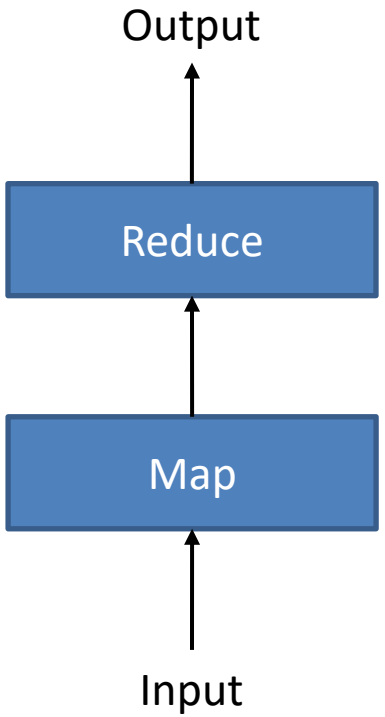
Concept: Dataflow programming

- **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 directed acyclic graph (DAG)
  - Vertices = operators
  - Edges = inputs and outputs
  - Operators are “black boxes”

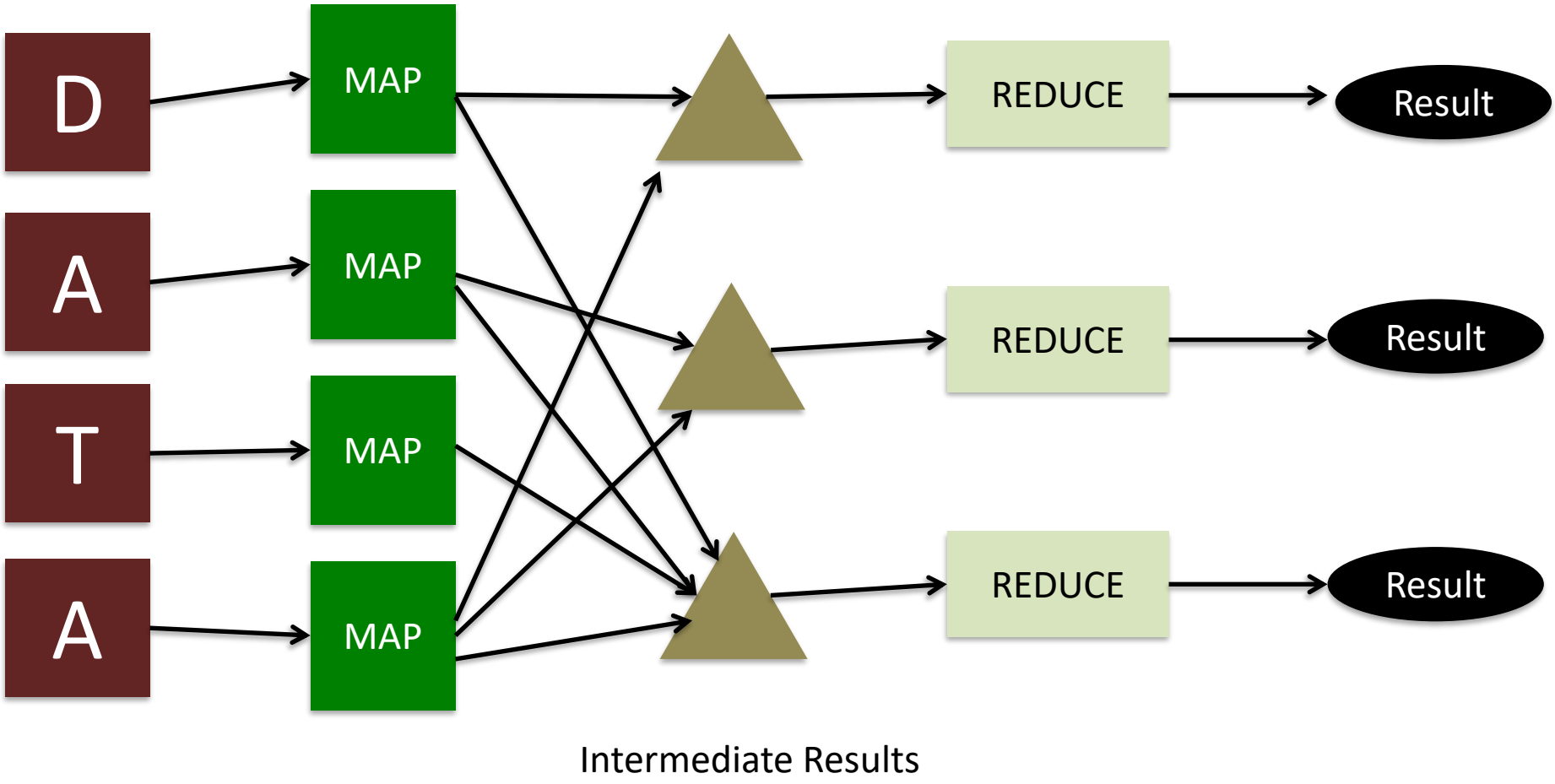
Example (logical) dataflows

```
select ename
from proj, asg, emp
where asg.eno = emp.eno
    and asg.pno = proj.pno
    and ename != "j. doe"
    and proj.pname = "cad/cam"
    and (dur = 12 or dur = 24)
```



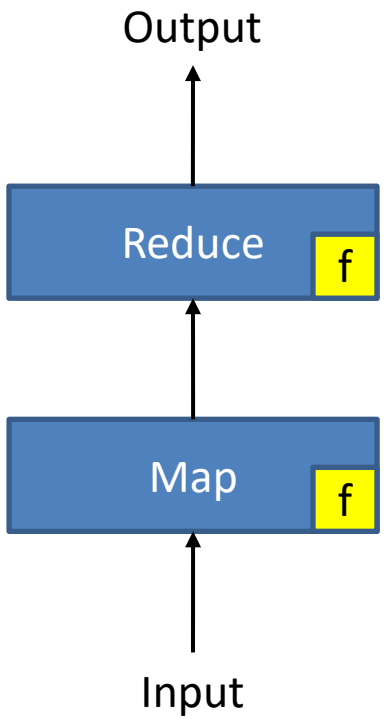


Example (physical) dataflow



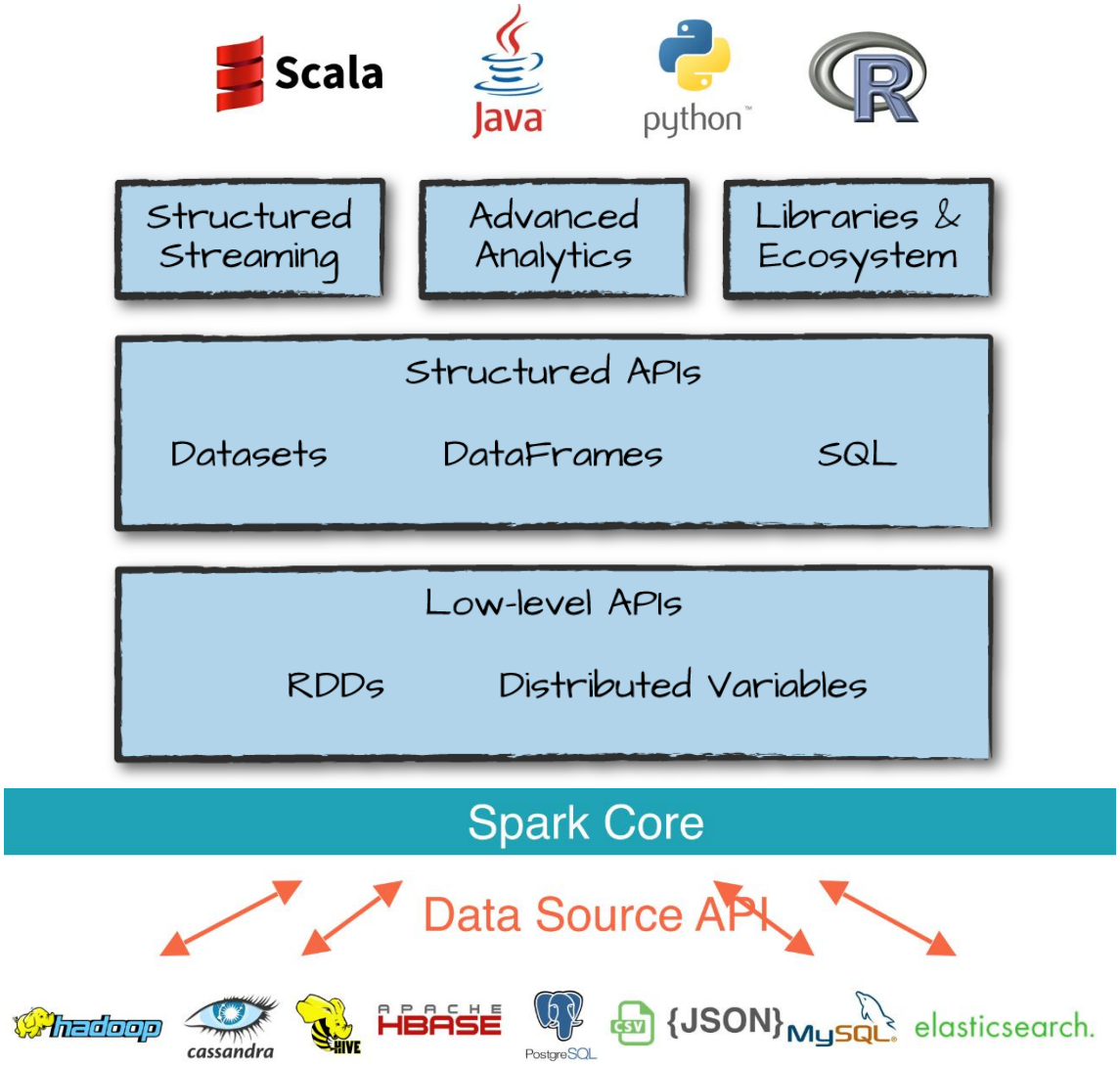
MapReduce as dataflow

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



Apache Spark

- A “unified engine for large-scale data analytics”
- An open-source Apache project (http://spark.apache.org/)
- Run on cluster or in cloud



Programmability (WordCount)

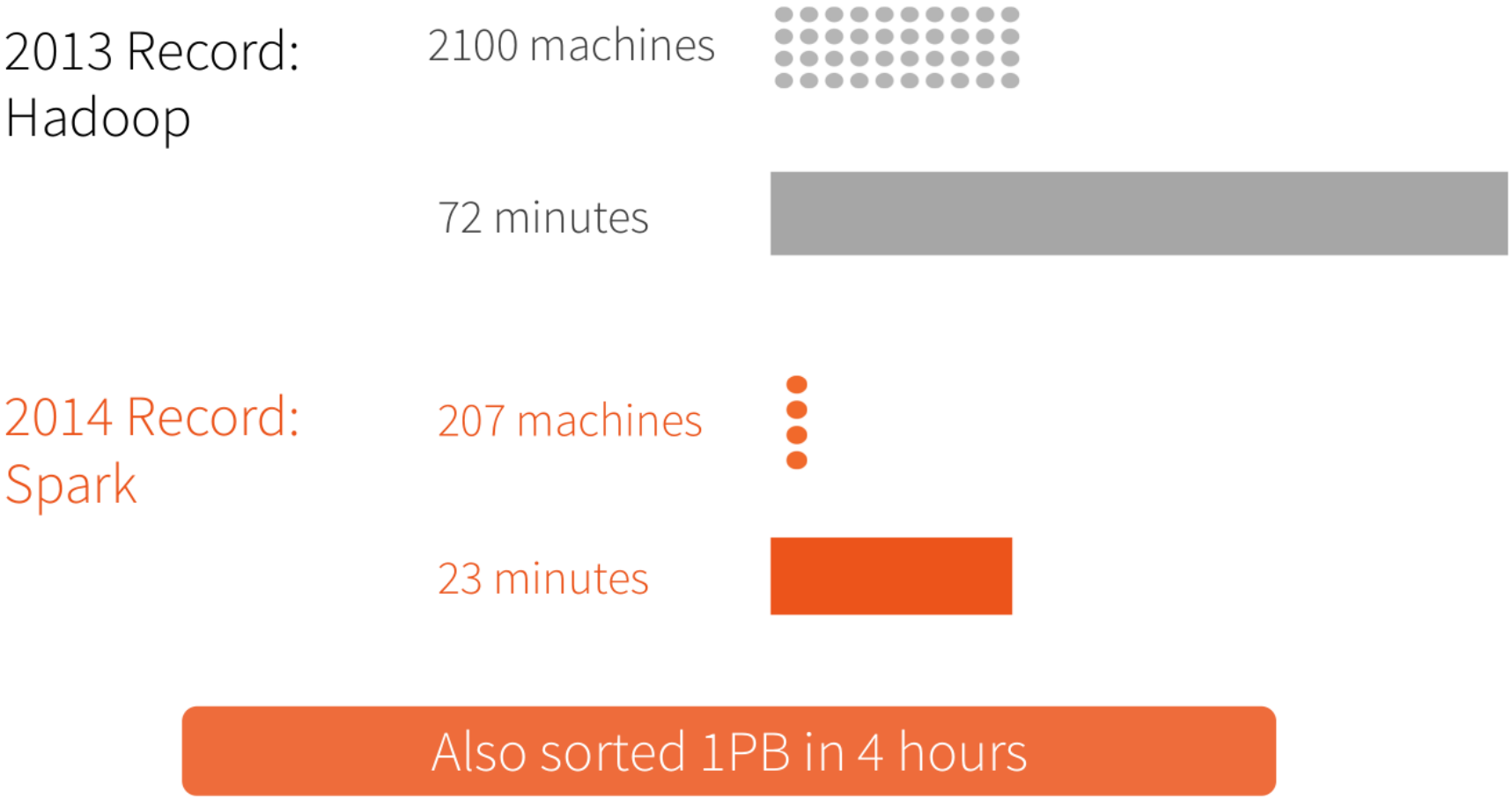
```java
package org.myorg;

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, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      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 RDD Scala API

Performance (Sorting 100TB)

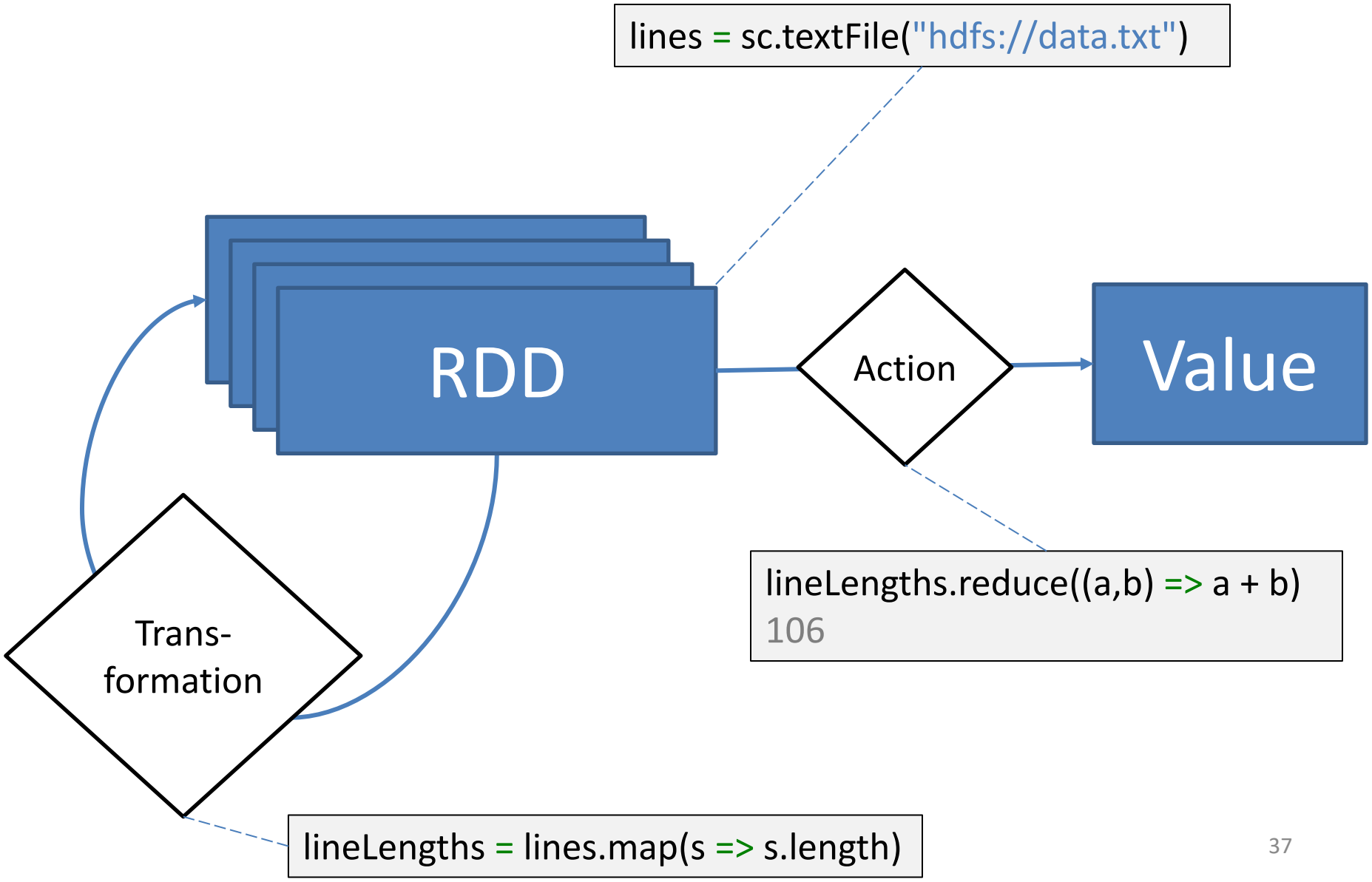


sortbenchmark.org, Xin 2015

Core abstraction: RDD

- Write programs in terms of distributed datasets and operations on them
- **Resilient distributed dataset (RDD)**
  - Immutable collections of objects (conceptually)
  - Spread across a cluster, stored in RAM, on disk, …
  - Built through parallel transformations
  - Automatically rebuilt on failure
  - Immutable
- Operations
  - **Transformations** (map, filter, groupBy, …)
  - **Actions** (count, collect, save, …)

Working with RDDs (1)



Input

- 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.)

Transformation

- **Transformation**
  - Takes an RDD and produces an RDD
  - Computed “lazily” (i.e., not immediately)
- Example: map(*f*)
  - Create new RDD obtained by applying *f* to each element of a given RDD

- **val** lineLengths = lines.map(s => s.length)
  lineLengths: RDD[Int] = MapPartitionsRDD […]
- lineLengths.collect
  res: Array[Int] = Array(49, 57)

Action

- **Action**
  - Takes an RDD and returns a value (to driver program, to external storage, …)
  - Triggers computation
- Example: collect()
  - Retrieve objects in RDD
- Example: reduce(*f*)
  - Parameterized by UDF *f*: (*T*, *T*) -> *T*
  - Applied repeatedly until only on value remains

- lineLengths.reduce((a,b) => a + b)
- 106

Language support

- Scala (Spark native)
  - `val lines = sc.textFile(...)`
    `lines.filter(_.contains("error")).count()`
- Java
  - `var lines = sc.textFile(...)`
    `lines.filter(s -> s.contains("error")).count()`
- Python
  - `lines = sc.textFile(...)`
    `lines.filter(lambda s: "error" in s).count()`
  - No static typing & potentially slower, else fine

*Common transformations (1)

transformation	*description*
map(*func***)**	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func***)**	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func***)**	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed***)**	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset***)**	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

*Common transformations (2)

transformation	*description*
groupByKey([*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*]**)**	when called on a dataset of `(K, V)` pairs where `K` implements `Ordered`, returns a dataset of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, (V, W))` pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
cartesian(*otherDataset***)**	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

*Common actions (1)

action	*description*
reduce(*func***)**	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n***)**	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed***)**	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

*Common actions (2)

action	*description*
`saveAsTextFile(`*`path`*`)`	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*`path`*`)`	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`*`)`	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Spark as a dataflow engine

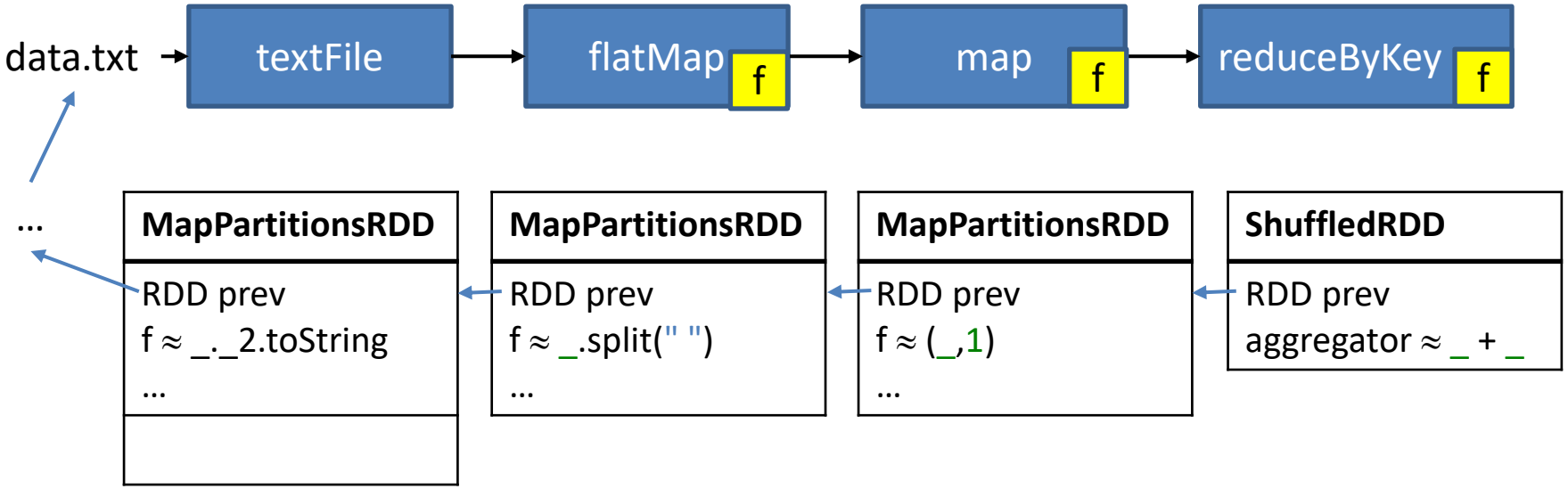
- Spark has RDDs and operators
- We can view it as a dataflow engine
  - *RDDs* correspond to logical dataflows
  - *Transformations* combine / add operators to dataflows
  - *Actions* compile and execute a dataflow
- An RDD is not really a collection of values
  - Instead: A dataflow that produces a collection of values when executed

Concept: Embedded DSL

- Spark provides an **embedded DSL**
  - I.e., a domain-specific language embedded in host language
- Difference to a “library“
  - Individual “operations“ are not executed, but used to construct a program
  - That program is then analyzed and executed elsewhere
  - Here: dataflow program, executed on Spark cluster

Example (1)

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



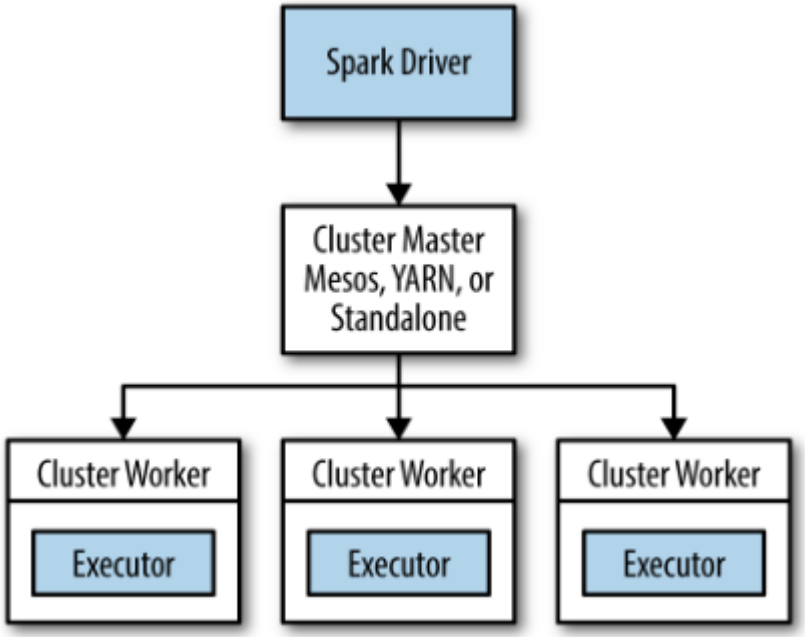
Example (2)

- 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 []
```

* Spark 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 on multiple executors (one per partition)



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

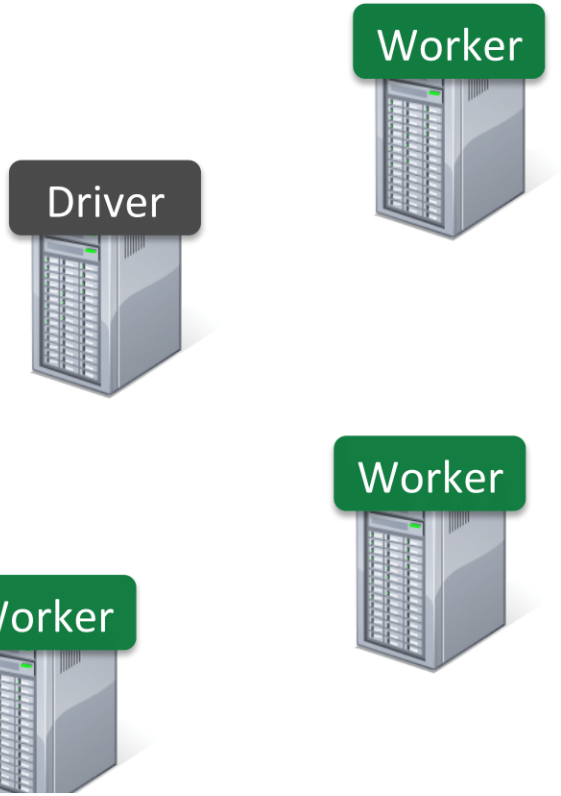
```
val counts = sc.textFile("hdfs://data.txt") \
    .flatMap(_.split(" ")).map((_, 1)) \
    .reduceByKey(_ + _).cache

counts.collect          // runs job, caches results
counts.collect          // reuses cached results
```

Example: Log Mining

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

Example: Log Mining

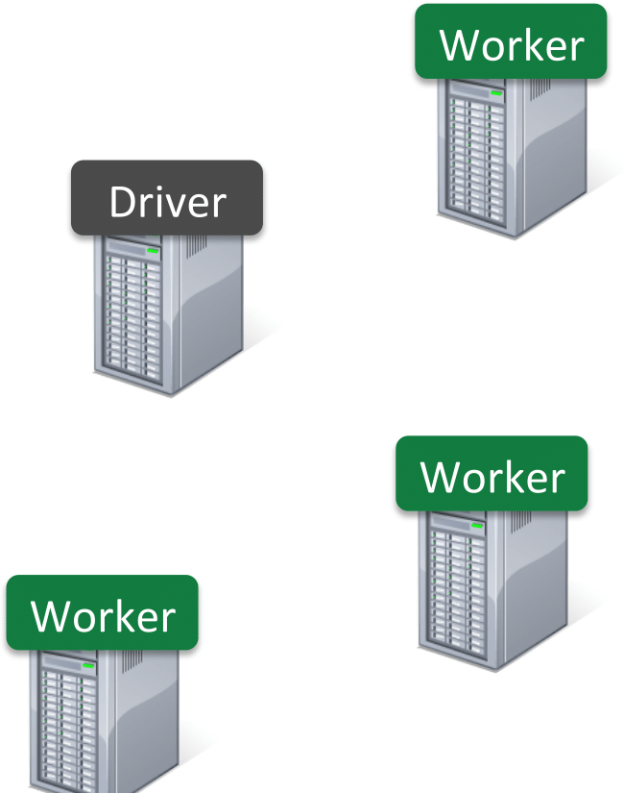
Load error messages from a log into memory, then interactively search for various patterns



Example: Log Mining

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

```
lines = spark.textFile(“hdfs://...”)
```

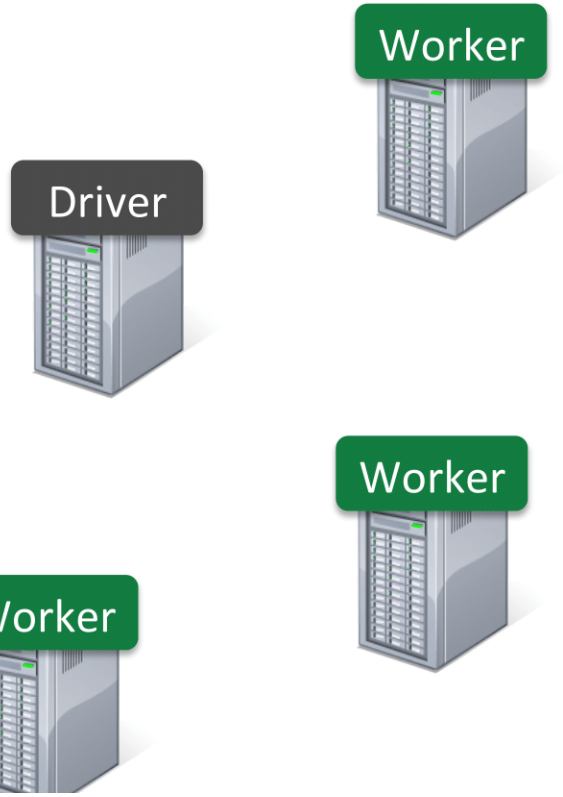


Example: Log Mining

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

Base RDD

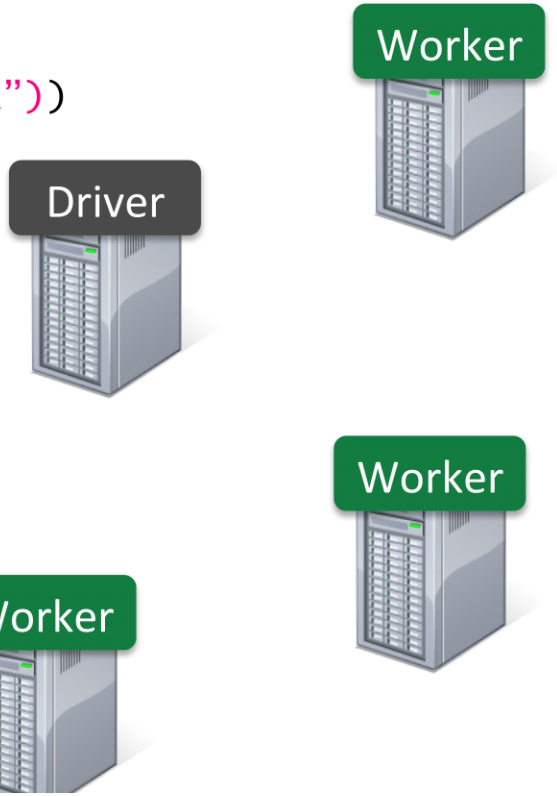
```
lines = spark.textFile(“hdfs://...”)
```



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

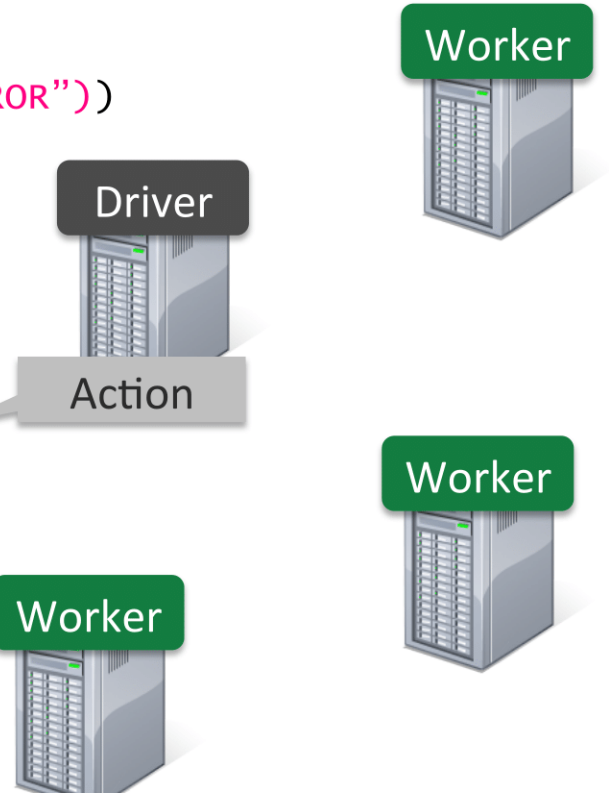
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()
```

```
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()
```

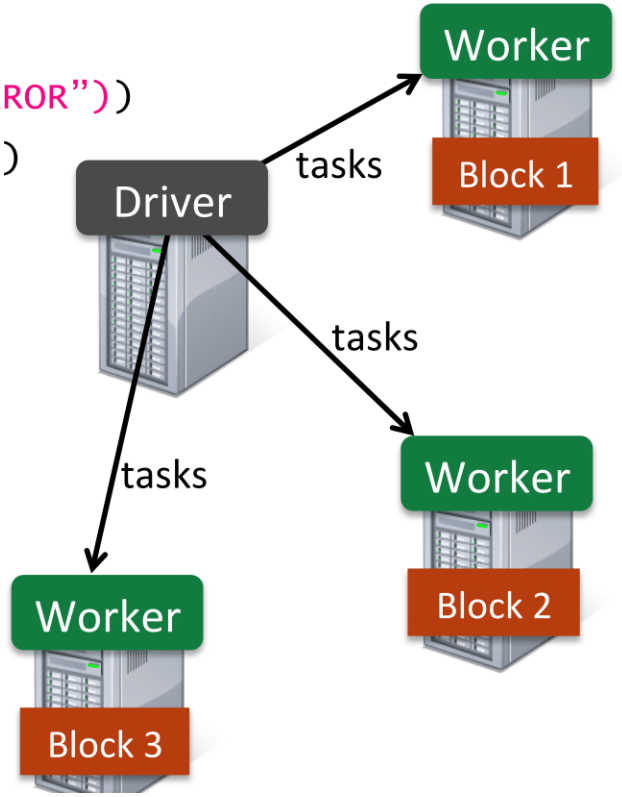
```
messages.filter(lambda s: “mysql” in s).count()
```

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()
```

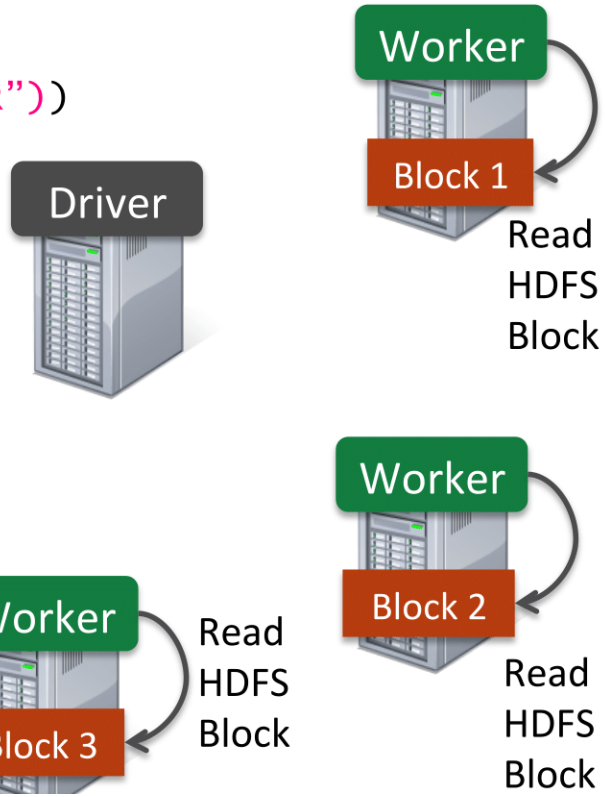
```
messages.filter(lambda s: “mysql” in s).count()
```



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()
```

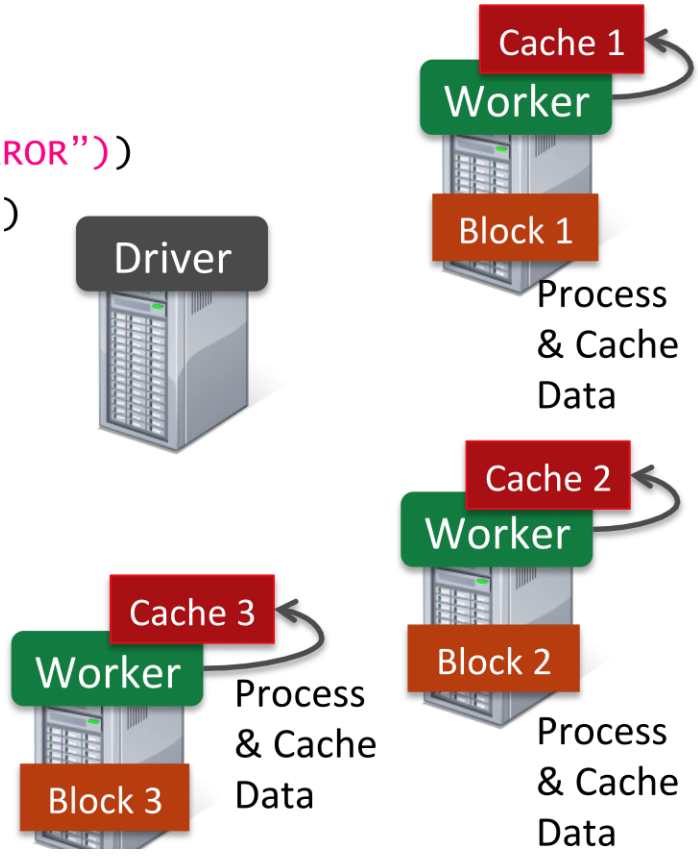


Example: Log Mining

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()
```

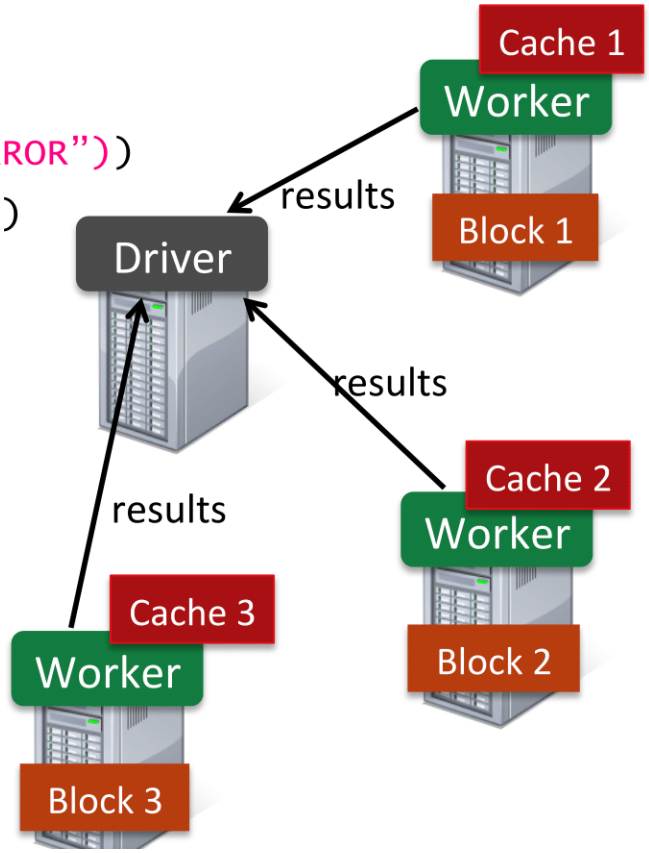


Example: Log Mining

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()
```

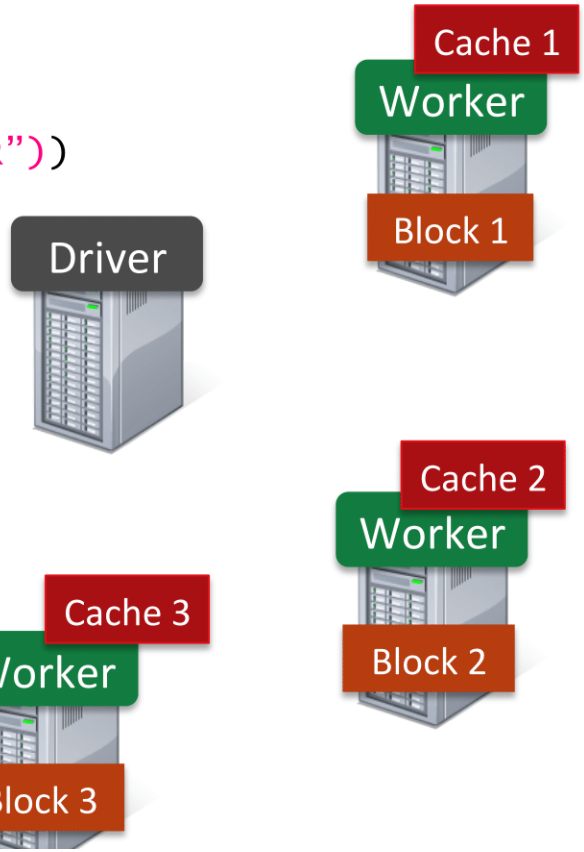


Example: Log Mining

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()
```

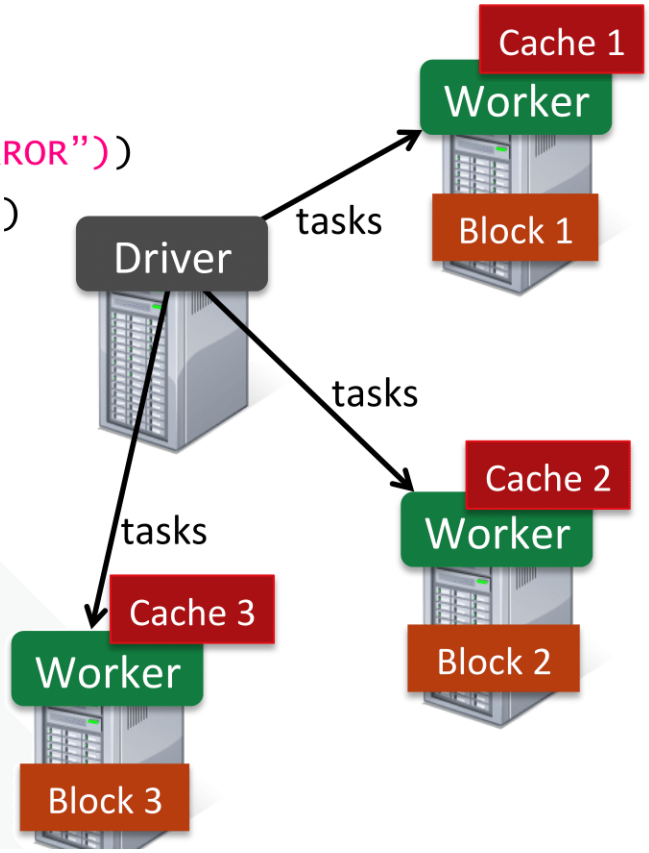


Example: Log Mining

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()
```

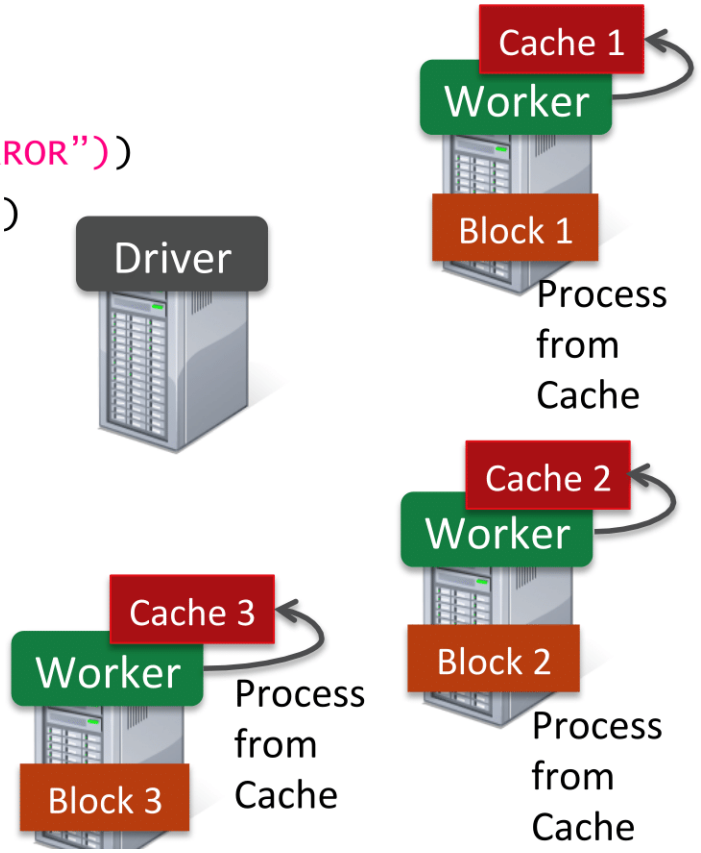


Example: Log Mining

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()
```

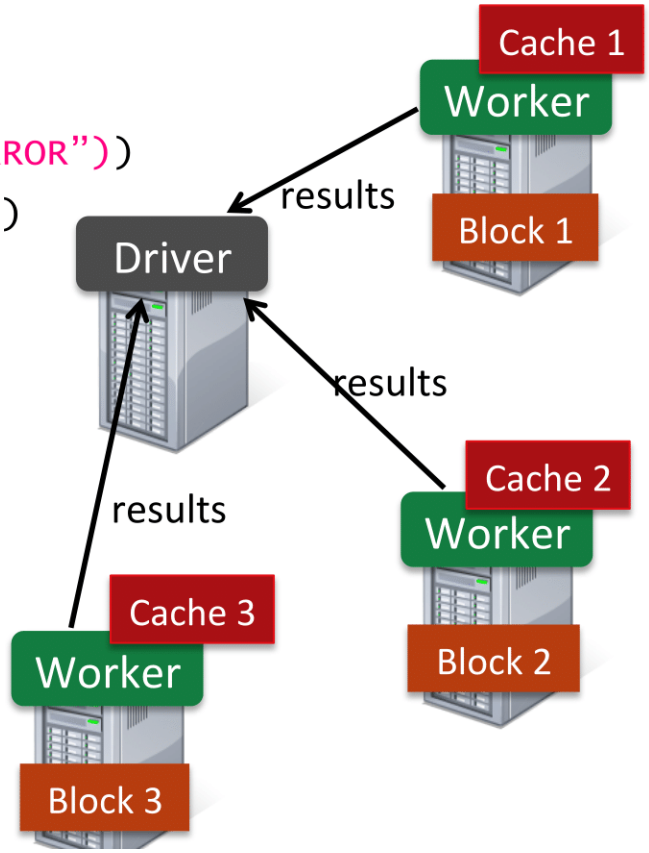


Example: Log Mining

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()
```

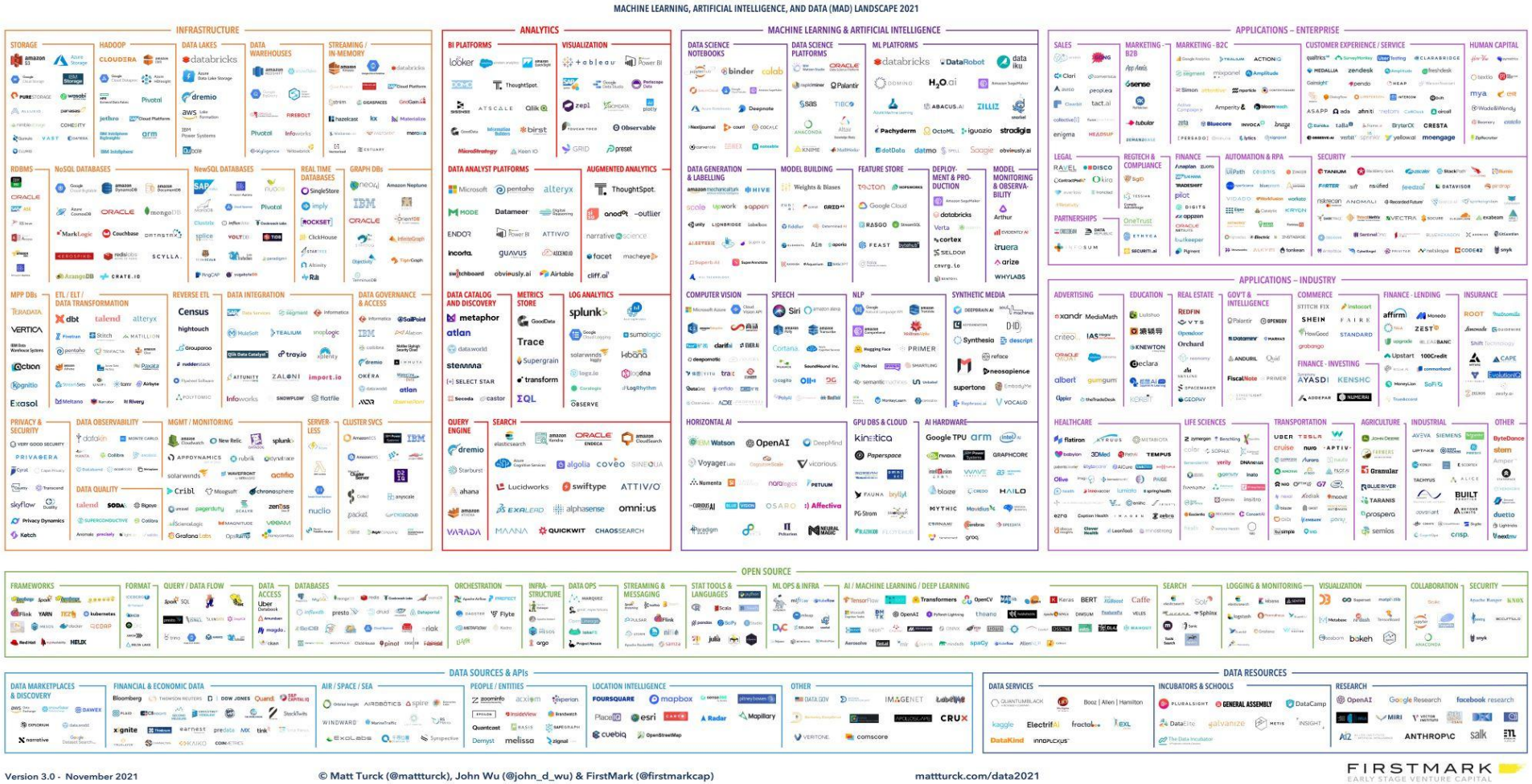
```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Summary

- Big data processing
  - Volume, velocity, variety
- Requirements
  - Scalability, efficiency, fault tolerance, elasticity, programmability, flexibility, cost, …
- Generally
  - Provide suitable programming interface (SQL, MapReduce, dataflow)
  - Do the heavy lifting automatically
- Similar ideas in many other applications
  - E.g., data stream processing
  - E.g., deep learning

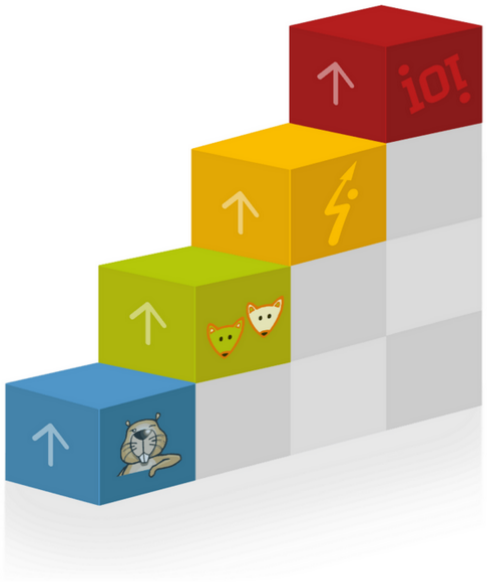
There is much more!



Literature

- Ö/V, Principles of Distributed Database Systems
  - Ch. 10: *Big Data Processing*
- Apache Spark
  http://spark.apache.org/
- Karau, Konwinski, Wendell, Zaharia
  Learning Spark
  2nd edition, O'Reilly, 2020

Informatik-Biber
Jugendwettbewerb Informatik
Bundeswettbewerb Informatik
Informatik-Olympiade



Bundeswettbewerb Informatik

Der Bundeswettbewerb Informatik richtet sich an Jugendliche bis 21 Jahre, die noch kein Studium oder keine Berufstätigkeit begonnen haben. Er beginnt jedes Jahr am 1. September, dauert etwa ein Jahr und besteht aus drei Runden. Die Aufgaben der ersten Runde können ohne größere Informatikkenntnisse gelöst werden, die Aufgaben der zweiten Runde sind deutlich schwieriger.

Begabung weiterentwickeln

Der Bundeswettbewerb ermöglicht den Teilnehmenden, ihr Wissen zu vertiefen und ihre Begabung weiterzuentwickeln. So trägt er dazu bei, Jugendliche mit besonderem fachlichen Potenzial zu erkennen.

Mit vielen Chancen verbunden

Eine Teilnahme am Bundeswettbewerb ist mit vielen Chancen verbunden. So können die Jugendlichen an verschiedenen Informatik-Workshops teilnehmen. Der Bundeswettbewerb ist fachlich so anspruchsvoll, dass die Gewinnerinnen und Gewinner in der Regel in die Studienstiftung des deutschen Volkes aufgenommen werden. Aus den Besten werden die deutschen Teilnehmerinnen und Teilnehmer an der Internationalen Informatik-Olympiade ermittelt.

Zum Bundeswettbewerb

Your turn

Work through today's lab.