Apache Spark under the hood

Big Data Management

03-10-2025

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

In the previous lecture …

- ✦ Resilient Distributed Datasets (RDDs)
- ✦ Transformations vs Actions

In this lecture …

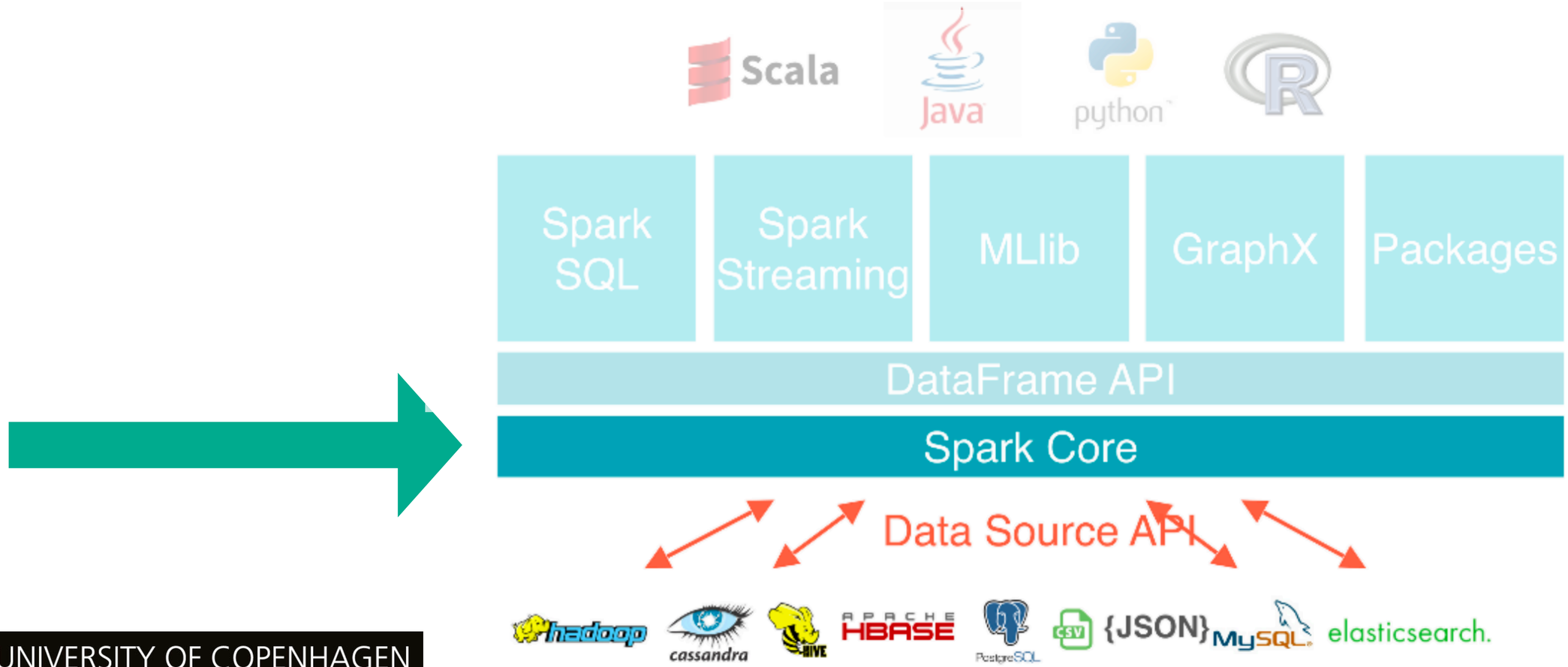
- RDD recall
- Spark under the hood
- Spark APIs
  - DataFrame
  - SQL

In this lecture …

- ✦ **RDD recall**
- ✦ Spark under the hood
- ✦ Spark APIs
  - ✦ DataFrame
  - ✦ SQL

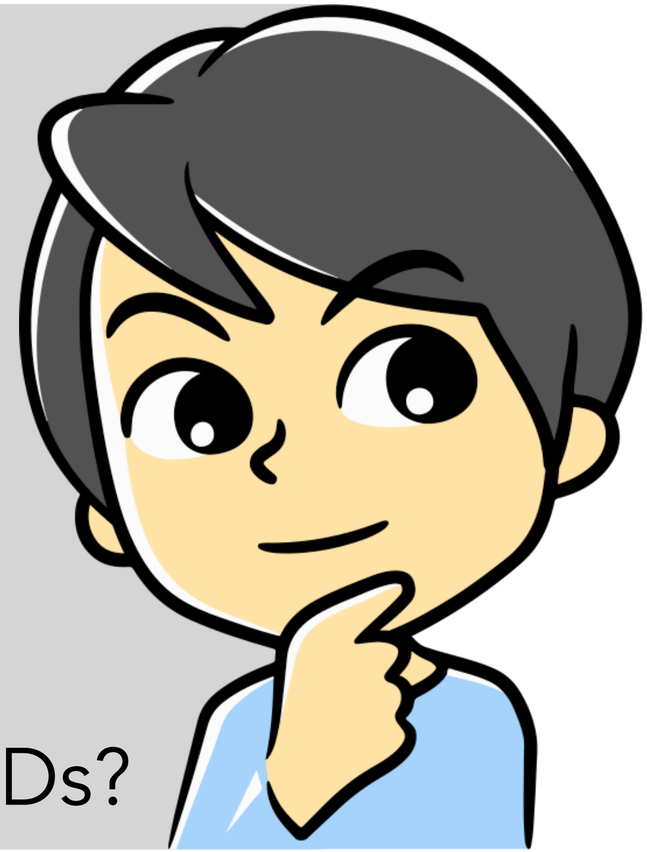
Apache Spark

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



Recap

- 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?


    ```

Recap

- 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 -> s.split(“,”)
  .filter(a -> a[4]=10)
  .mapToPair(a -> new Tuple2(a[2],a[3]))
  .reduceByKey((a, b)-> a+b)
  .collect()
```

Broadcast in Spark

- ✦ Global variables that needs to be in each worker
- ✦ Read-only
- ✦ Cached on each machine

```
val broadcastVar = sc.broadcast(Int(10))
val rdd = sc.parallelize(Array(2, 3, 4, 5, 6))
val result = rdd.map(x => x * broadcastVar.value).collect()
```

Spark as a dataflow engine

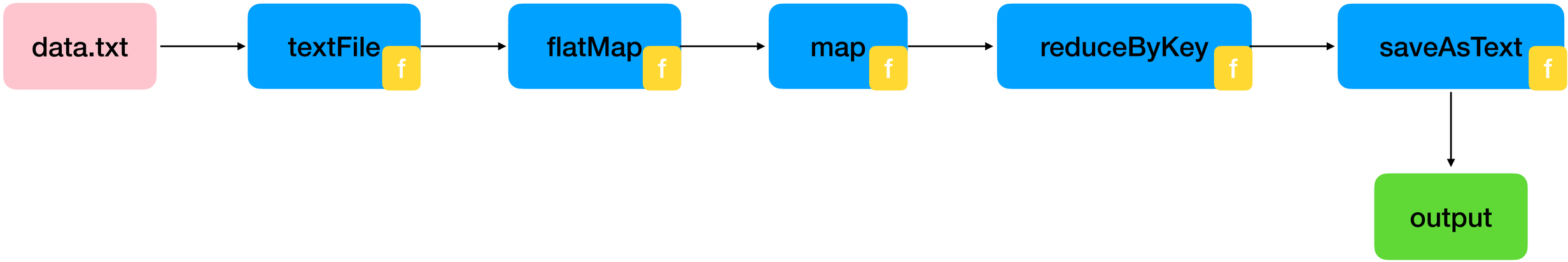
- Spark has RDDs and operators

- We can view it as a dataflow engine
  - An RDD job corresponds to a logical dataflow
  - 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

Spark RDD job as a logical plan

```
sc.textFile(”hdfs://data.txt”)
.flatMap(s -> s.split(" “))
.map(w -> (w, 1))
.reduceByKey(p -> (p.first()+p.second()))
.saveAsText(“hdfs://output.txt");
```



In this lecture …

- RDD recall
- **Spark under the hood**
- Spark APIs
  - DataFrame
  - SQL

RDD is an interface

- Abstract class in Scala (RDD[T])

- Consists of
  1. List of partitions (= blocks/splits in MapReduce/Hadoop)
  2. List of dependencies on parent RDDs
  3. Function to compute a partition from its parents
  4. (Optional) a partitioner (hash, range)
  5. (Optional) preferred locations for each partition

lineage

optimization

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

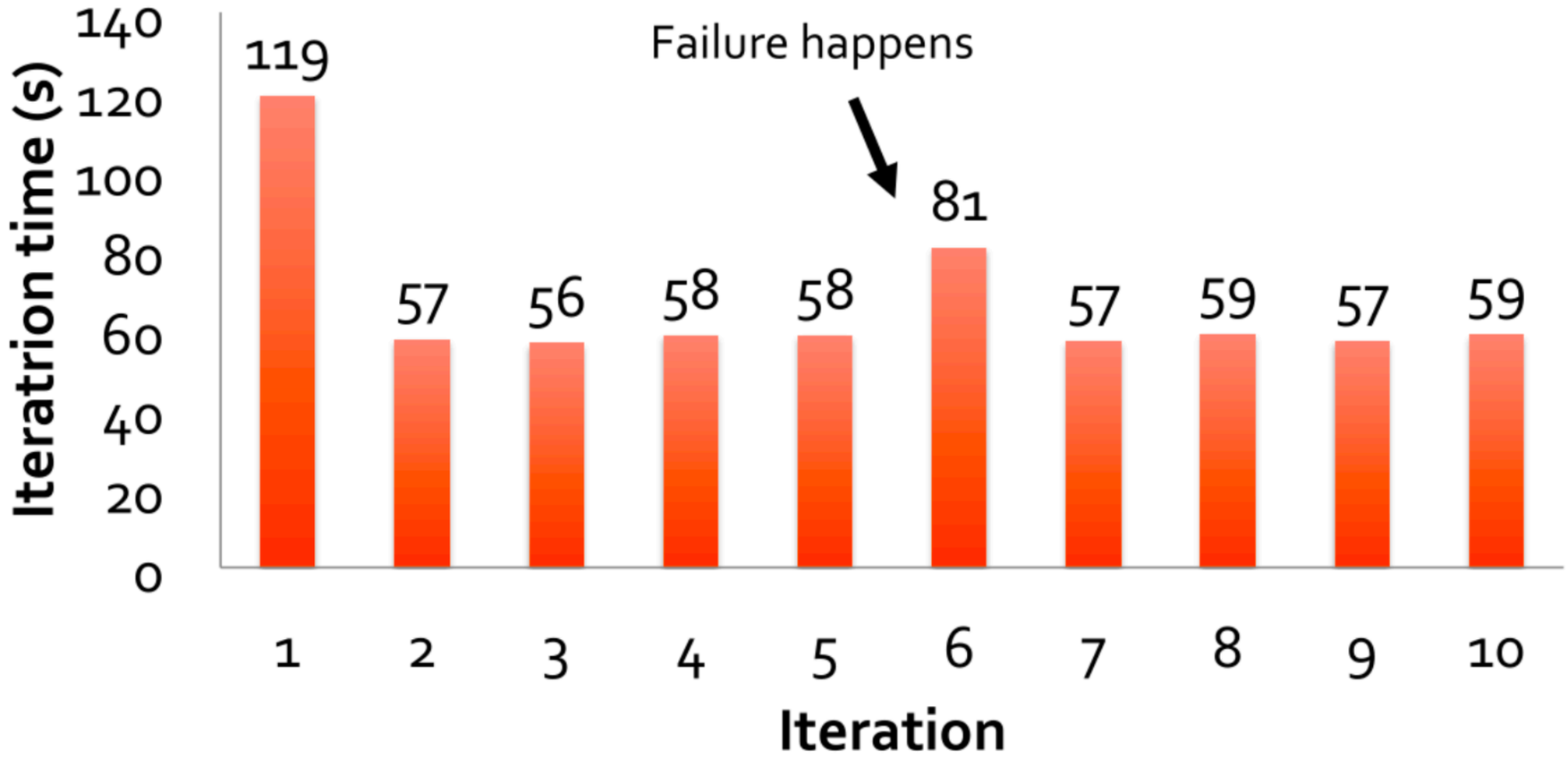
Example: FilteredRDD

- 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

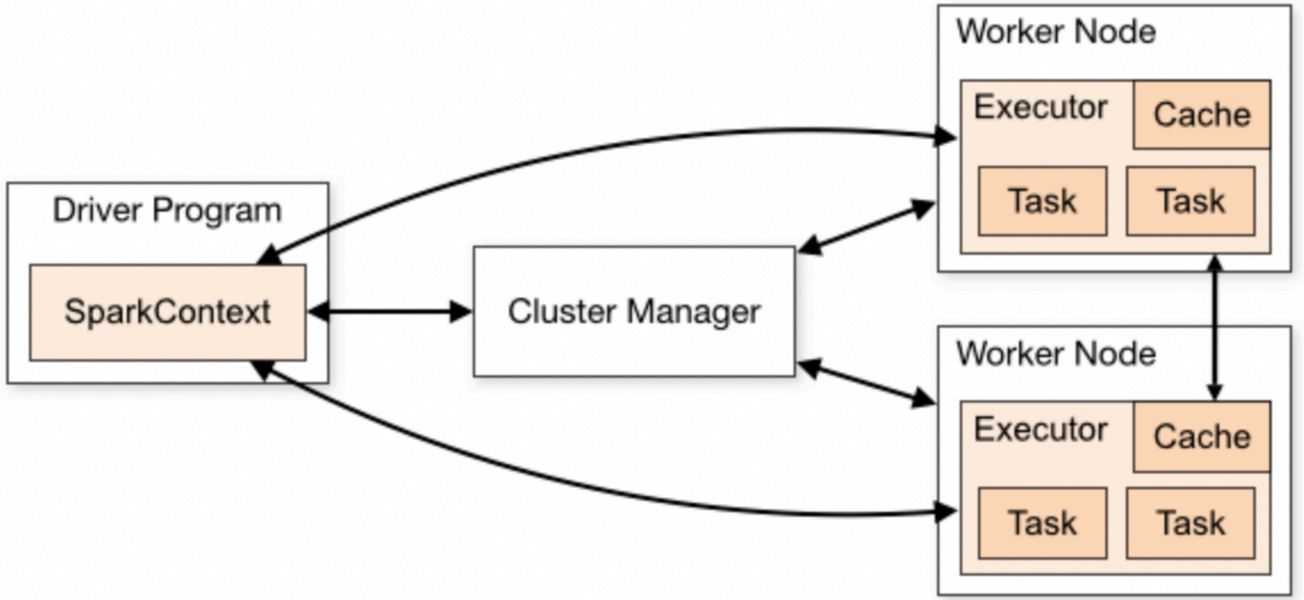
Fault-tolerance via lineage

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



Spark 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**
    - Tasks can run in parallel (one per partition)

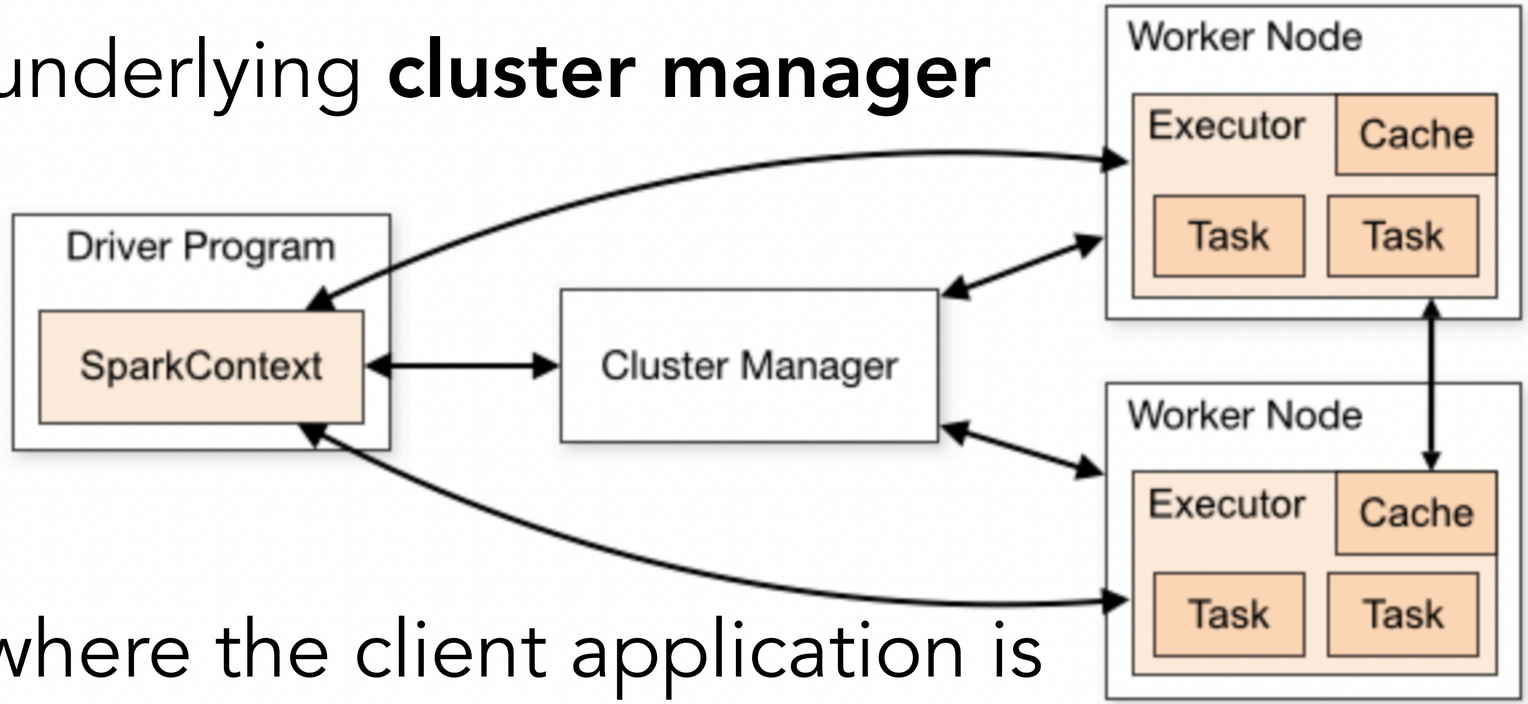


Spark Execution Modes

- Local mode (pseudo-cluster)
  - Non-distributed single JVM deployment
- Cluster mode
  - Distributed deployment

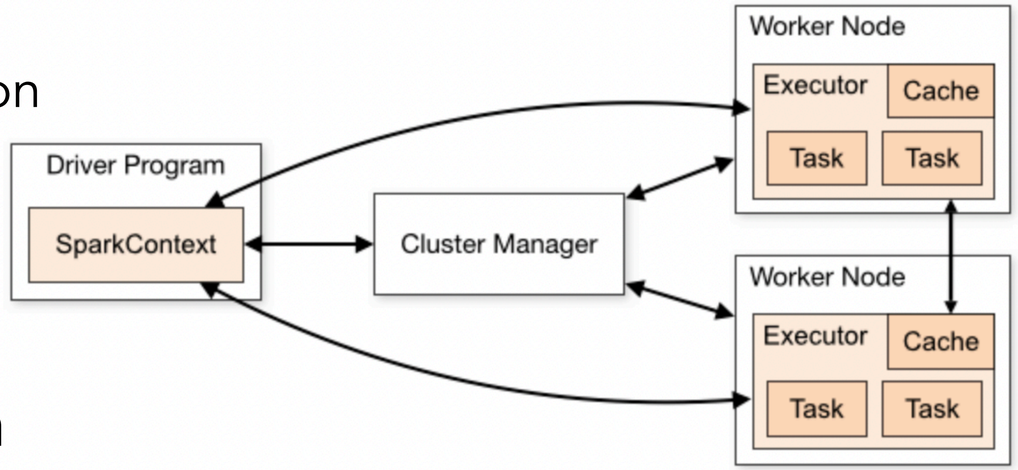
Spark Cluster Execution

- Standard way to deploy Spark in a **private cluster**
- Spark is agnostic to underlying **cluster manager**
  - Apache Mesos
  - Hadoop YARN
  - Kubernetes
- **Driver** by default is where the client application is



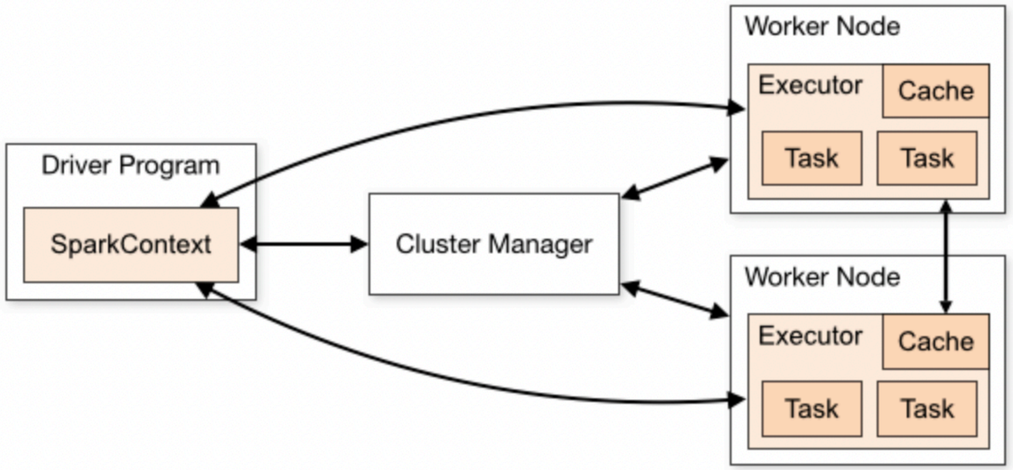
Spark Cluster Execution

- Spark applications are run as **independent sets of processes**, coordinated by a SparkContext in a driver program.
  - driver: process running the “main()” of the application
- Via the context the driver connects to the cluster manager which allocates resources.
- Each worker in the cluster is managed by an executor.
  - The executor manages **tasks computation** as well as **storage** and **caching** on each machine.



Spark Cluster Execution

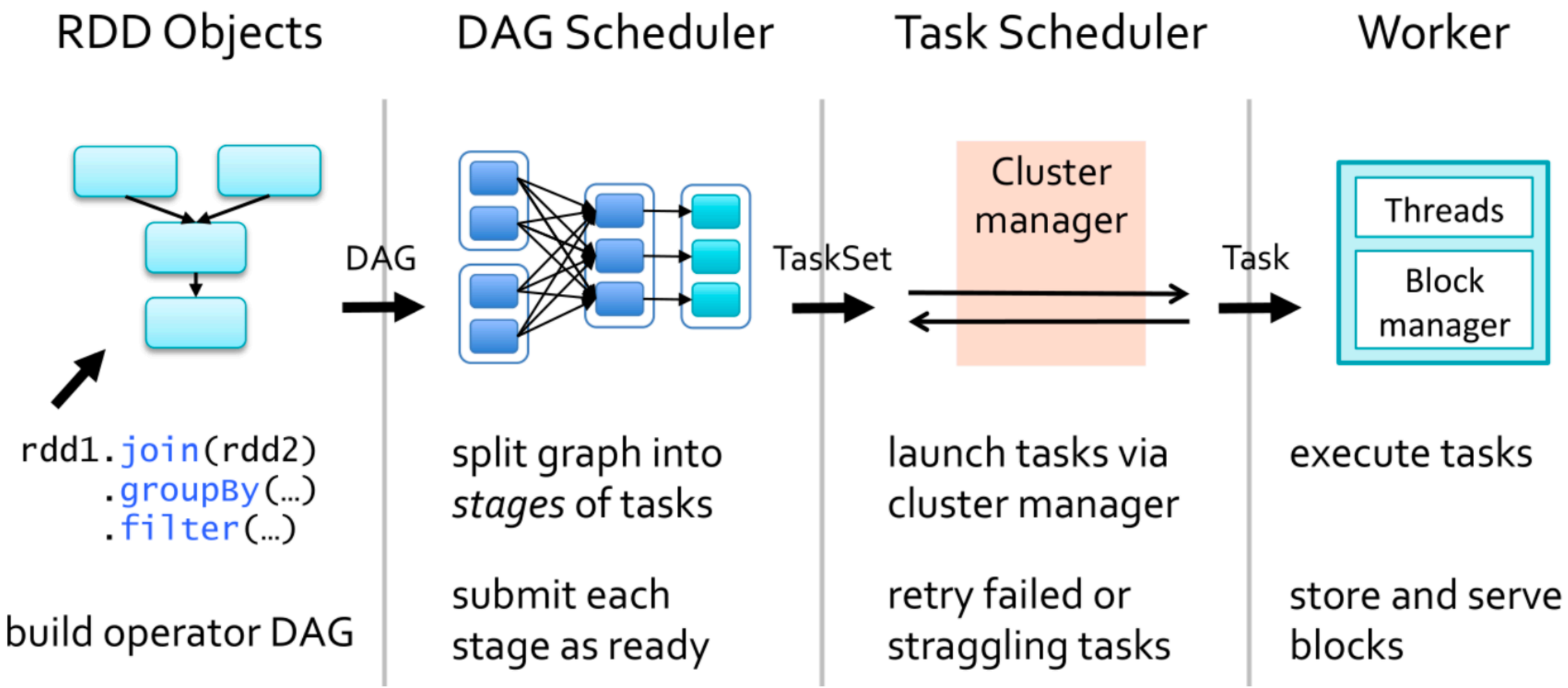
- The application code is sent from the driver to the executors
  - This happens when there is an **action**
- The executors specify the context and the various tasks to be run.
- The driver program must listen for and accept incoming connections from its executors throughout its lifetime.
- Tasks are launched in separate threads, typically one per core but can be configured.



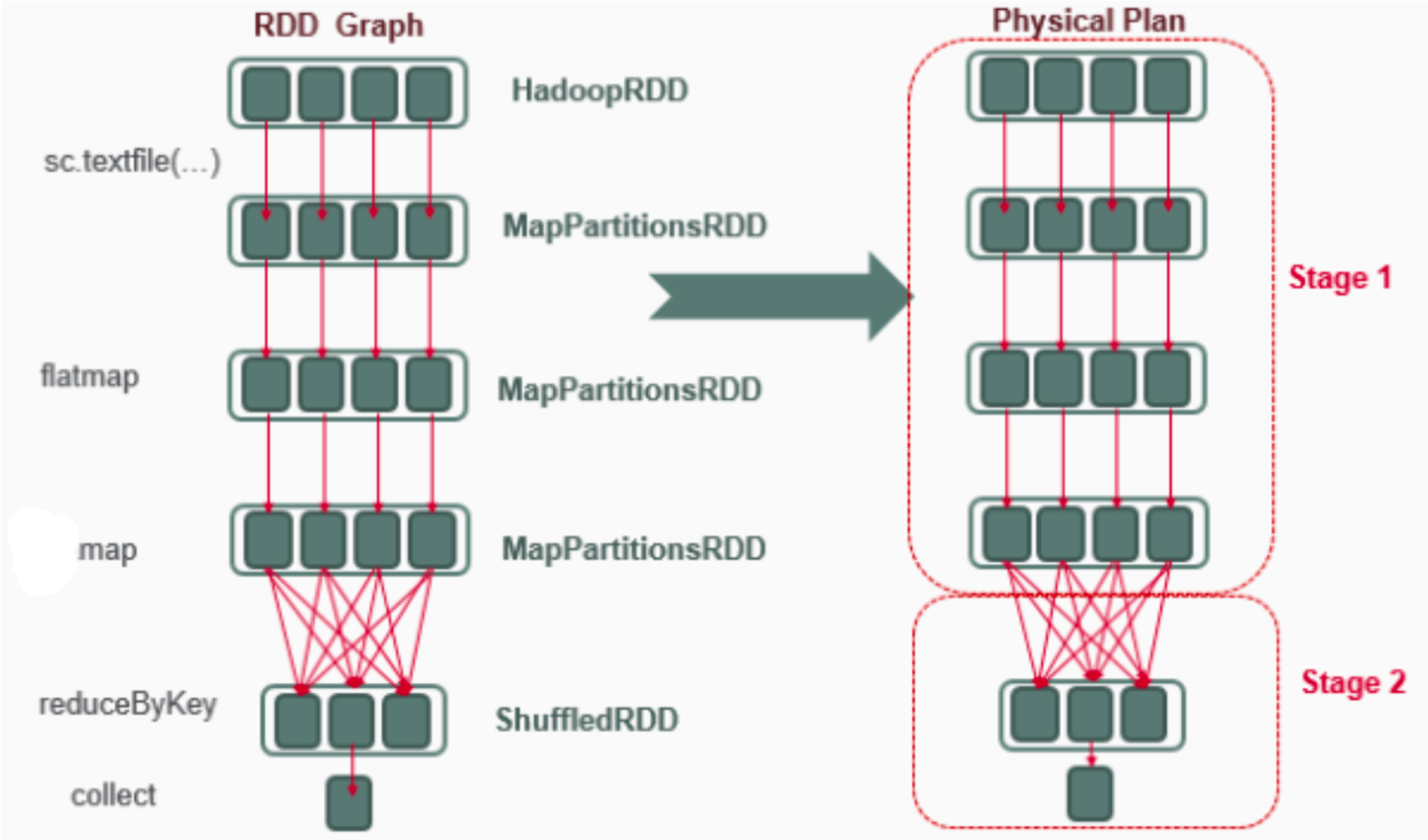
Spark Terminology

- **RDD**: distributed dataset with extra information
- **DAG**: logical graph of RDD operations
- **Task**: individual unit of work in 1 executor over 1 partition
- **Job** : set of tasks executed as a result of an **action**
- **Stage**: set of tasks in a job that can be executed in parallel without any network communication

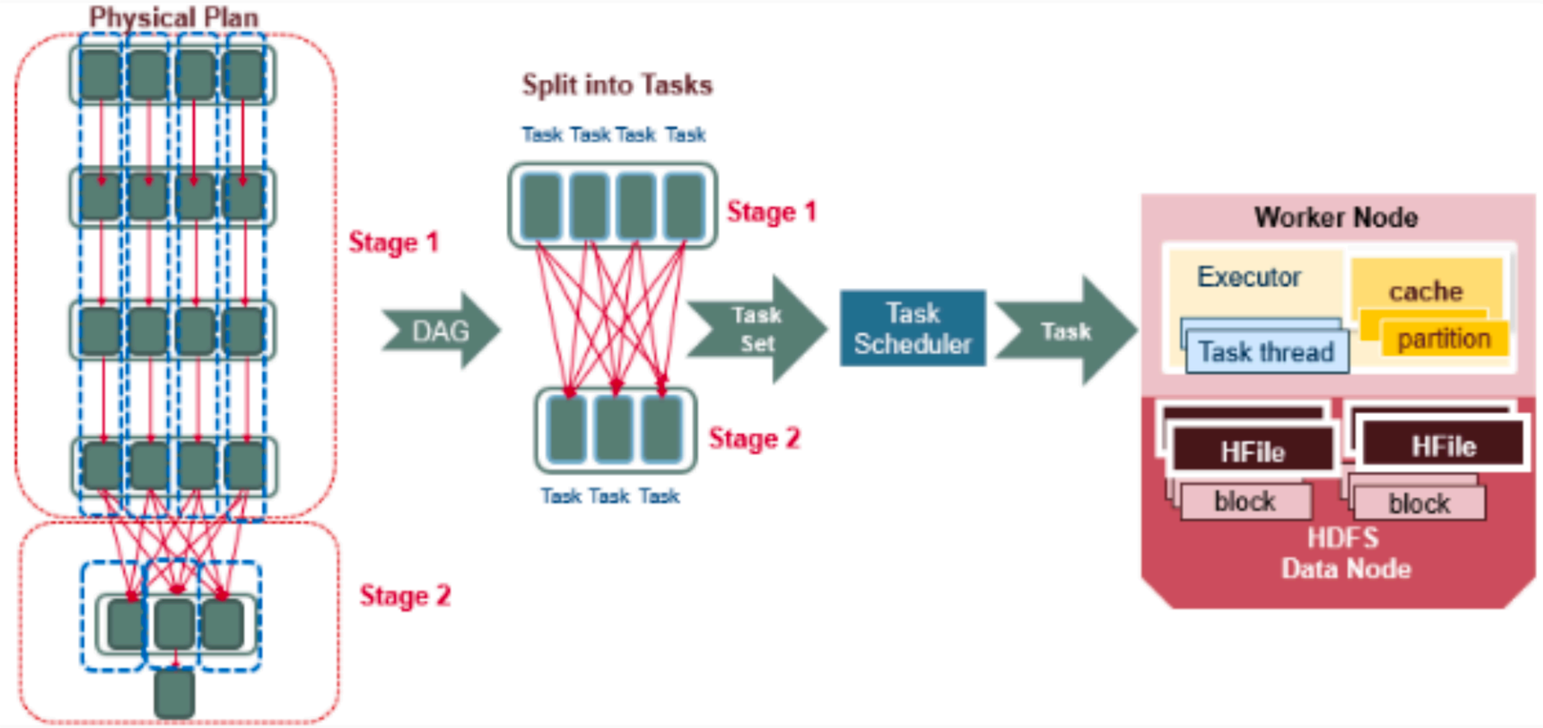
Execution process



Execution plans - example



Execution plans - example



Execution plans

- When using an **action:** logical representation —> physical data flow (**execution plan**)
- Multiple operations can me merged into **tasks**
  - Each task fetches its input, performs some operations, and produces output
- Tasks are grouped into **stages**
  - Tasks within a stage can be computed **without data movement/shuffling**
  - 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

Spark’s parallel processing

- **Sort** uses parallel **range partitioning sort**
- By default, **join** transformation performs a **parallel hash join**
- **groupByKey** repartitions data based on grouping key (default: **hash**; users can override)
- **reduceByKey** additionally uses **pre-aggregatio**n
- 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

- What does the following code do?

```
JavaPairRDD<String, Integer> wordsCounts =
sc.textFile(”hdfs://data.txt”)
.flatMap(s -> s.split(" “))
.map(w -> (w, 1))
.reduceByKey(p -> (p.first()+p.second()));
```

```
wordsCounts.collect();
wordsCounts.collect();
```

Reusing computation

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

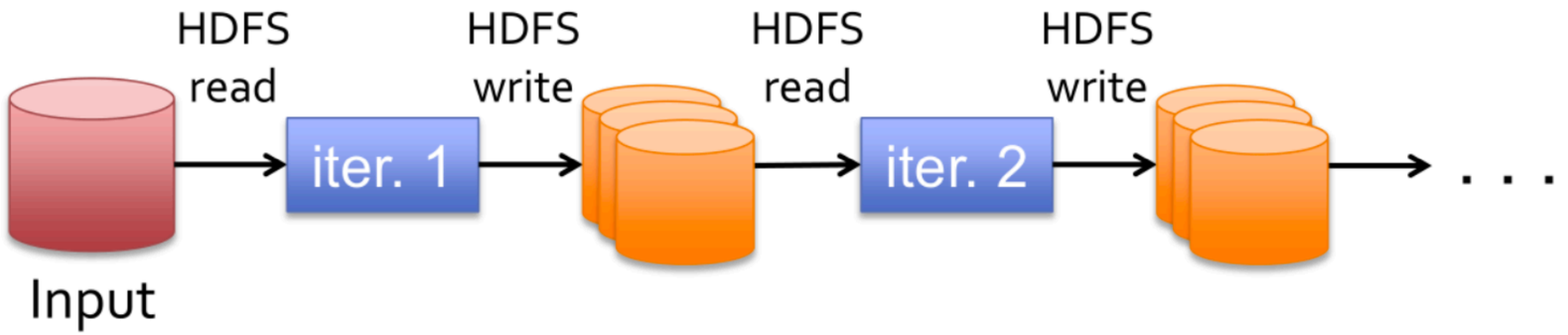
```
wordsCount= sc.textFile(”hdfs://data.txt”)
.flatMap(s -> s.split(" “))
.map(w -> (w, 1))
.reduceByKey(p -> (p.first()+p.second()))
.cache()
```

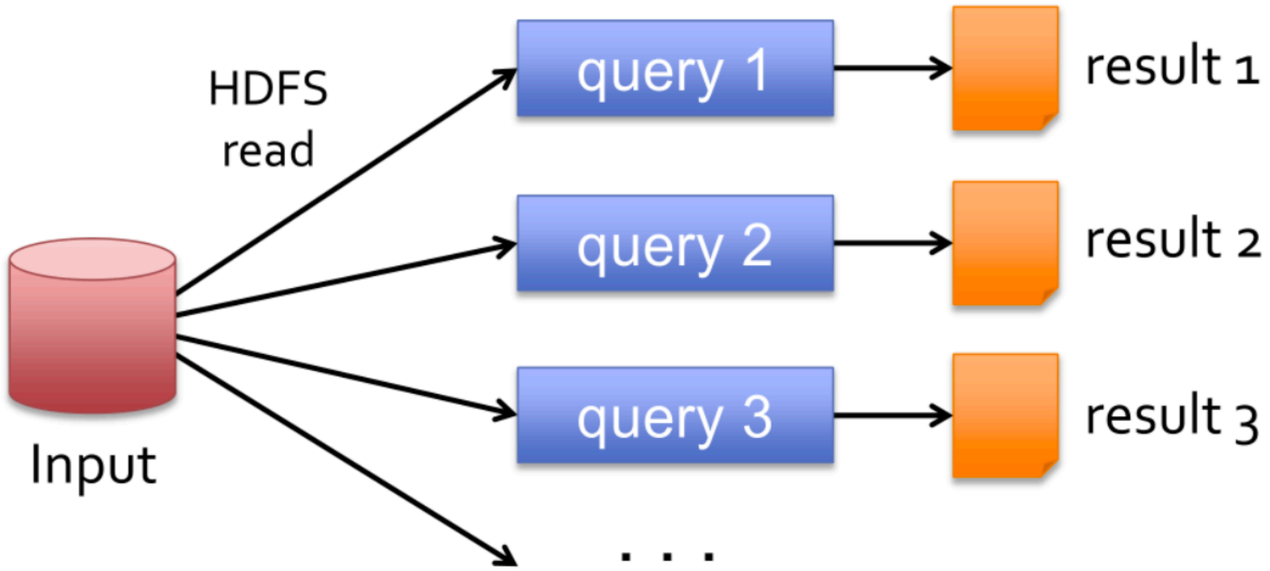
```
wordsCounts.collect();
wordsCounts.collect();
```

runs job, caches results

uses cached results

Data sharing in MapReduce

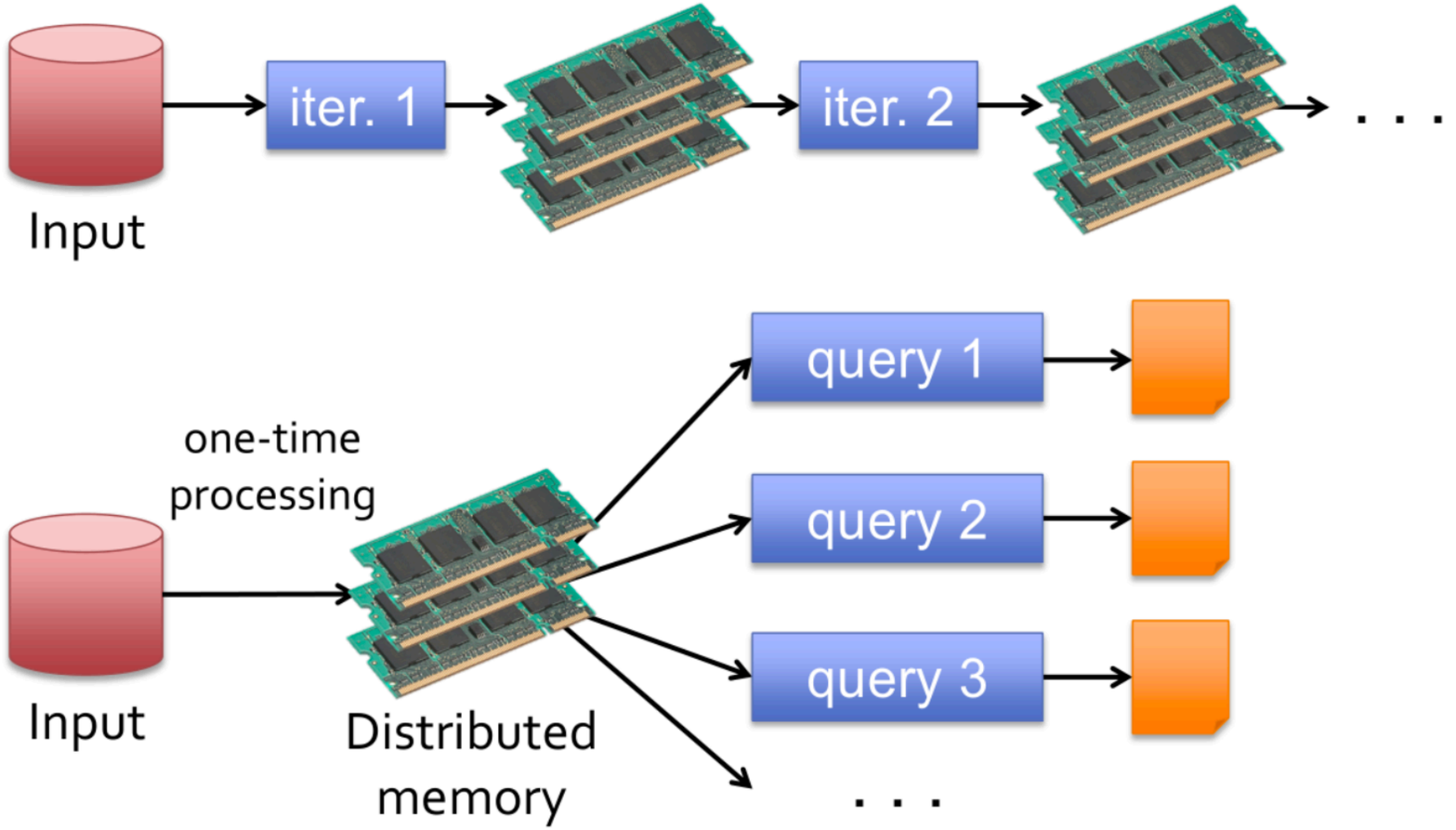




Slow due to replication, serialization, and disk IO

event.cwi.nl/lsde

Data sharing in Spark

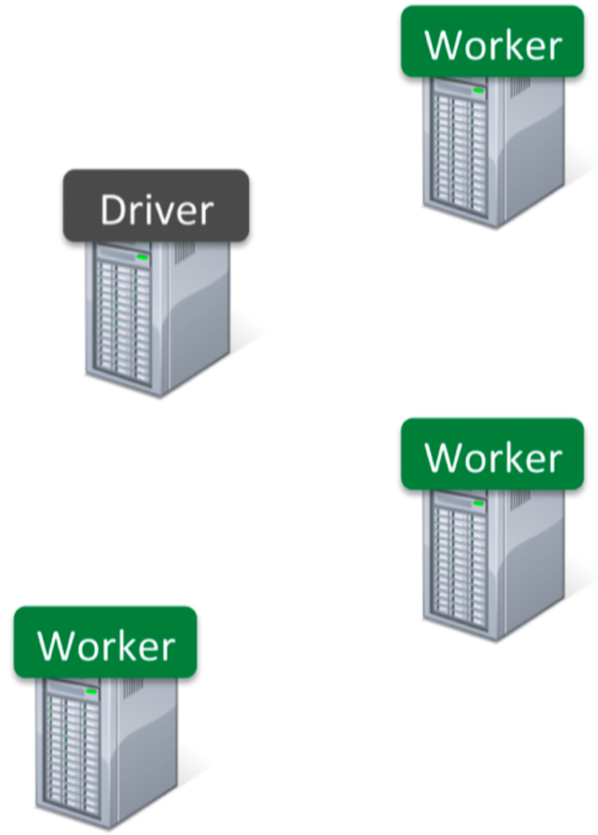


~10 × faster than network and disk

event.cwi.nl/lsde

Example: log mining

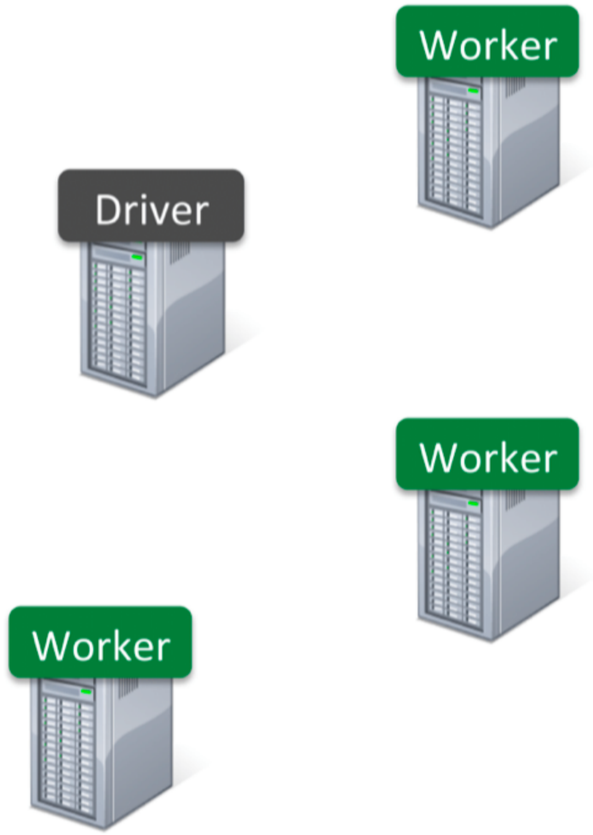
Load error messages from a log into memory, then interactively search for various patterns



Example: log mining

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

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

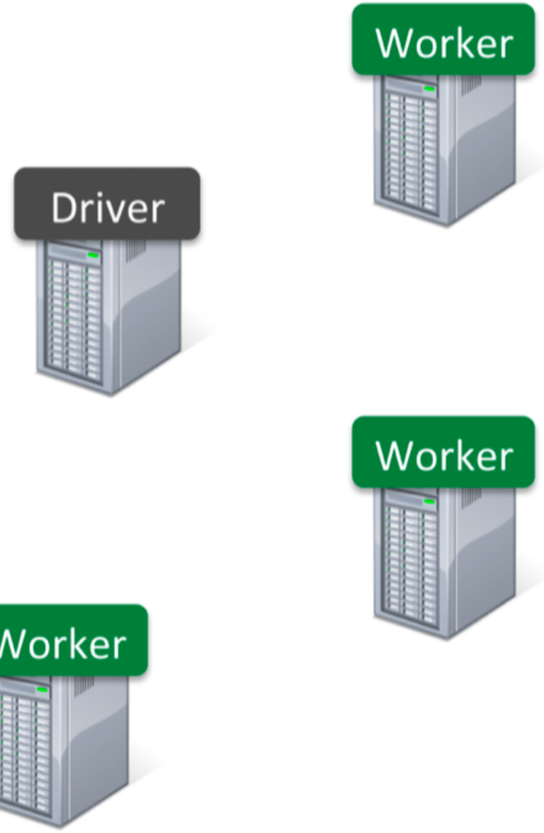


Example: log mining

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

BaseRDD

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



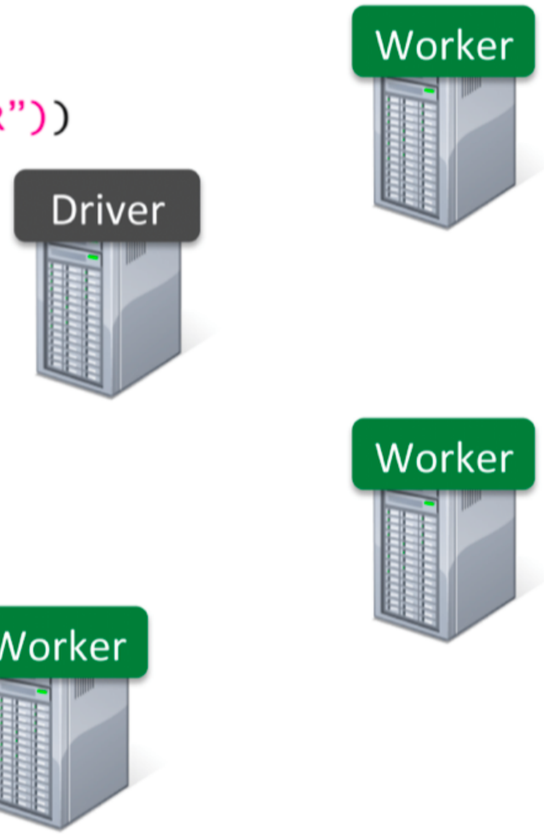
Example: log mining

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

BaseRDD

TranformedRDD

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



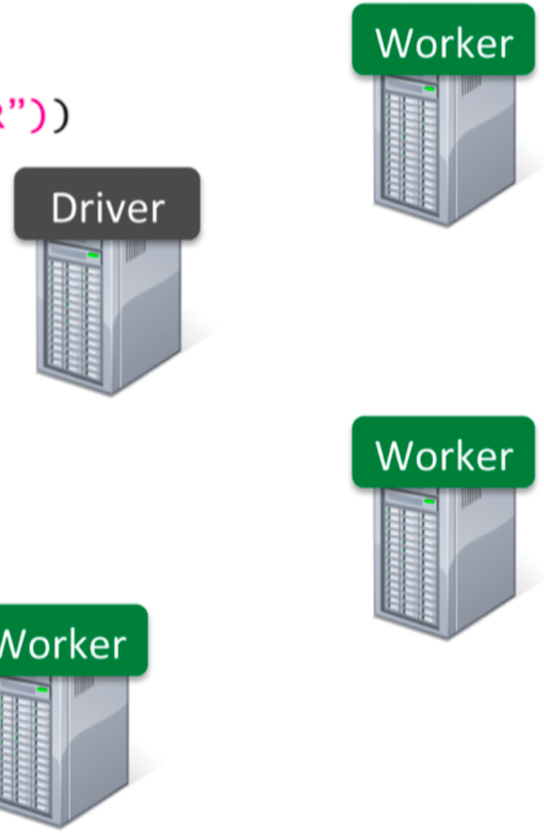
Example: log mining

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()
```

Action

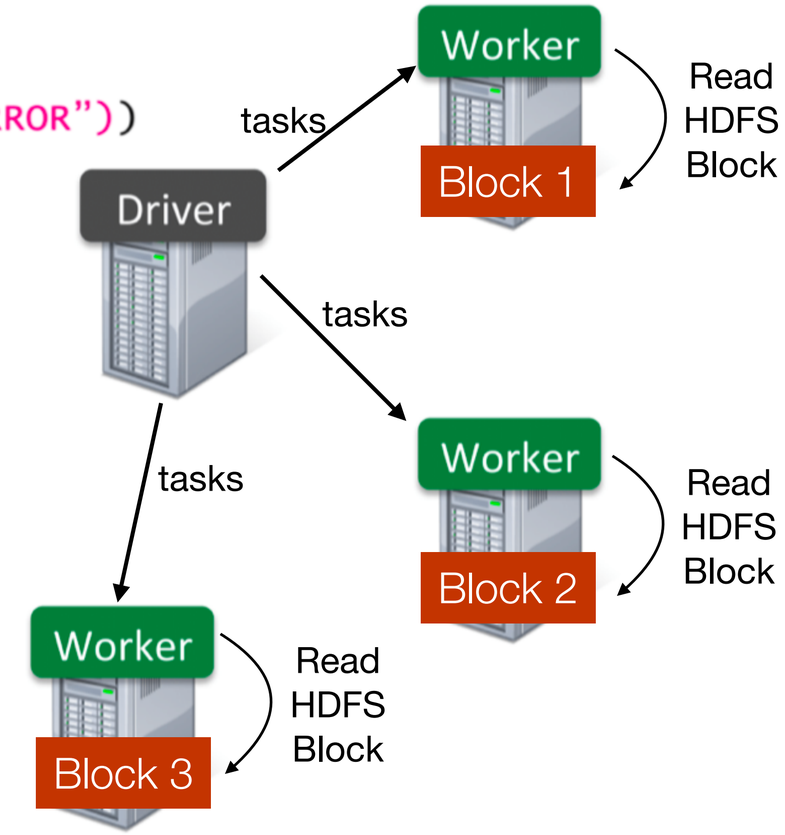


Example: log mining

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()
```

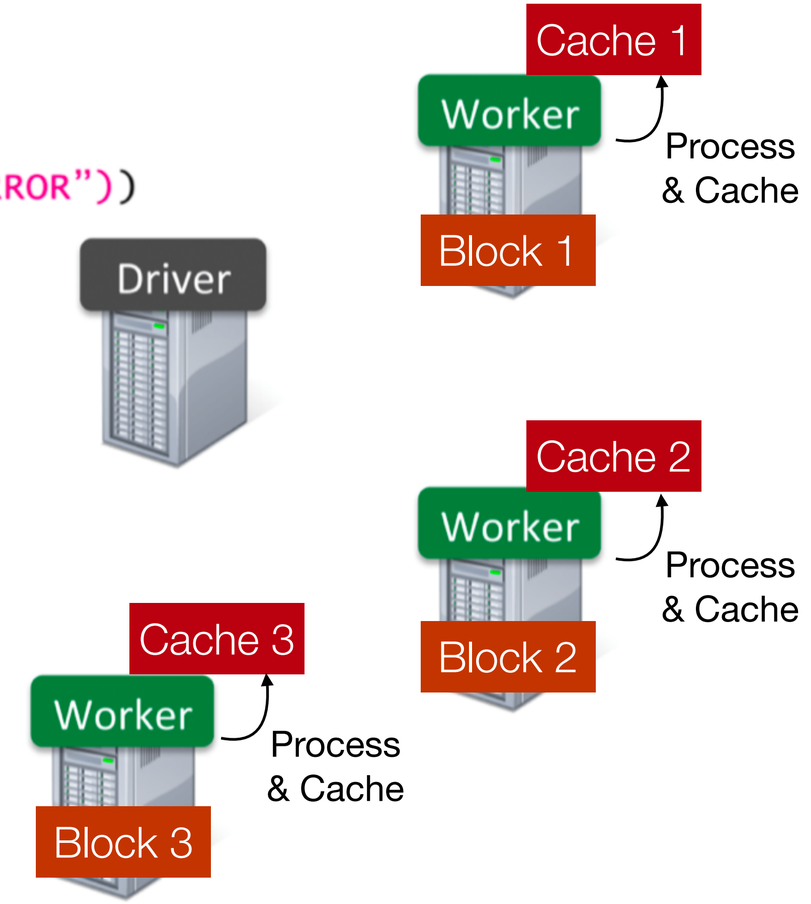


Example: log mining

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()
```

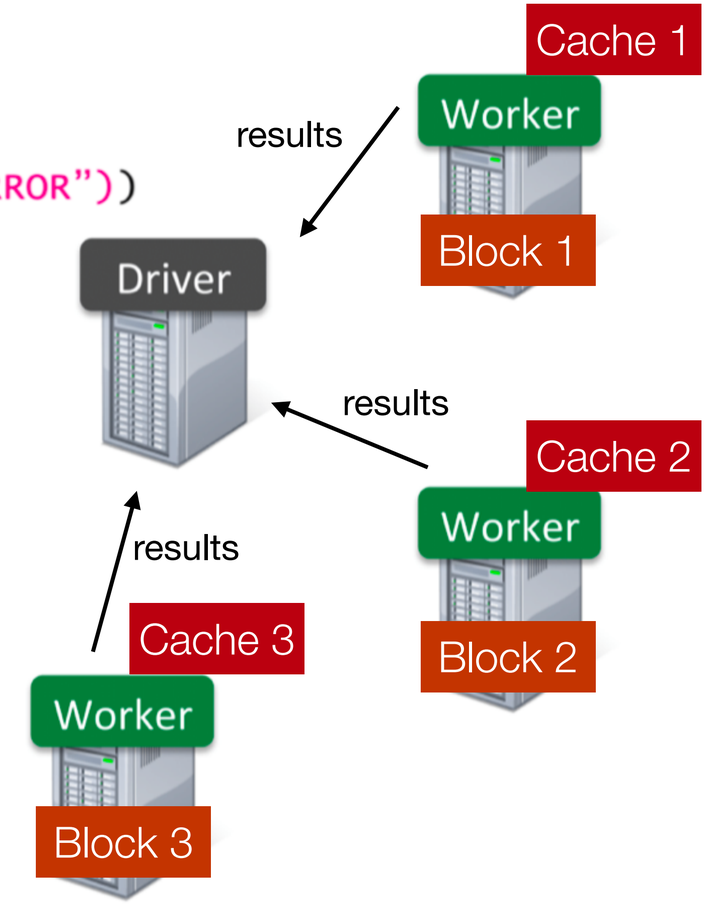


Example: log mining

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()
```

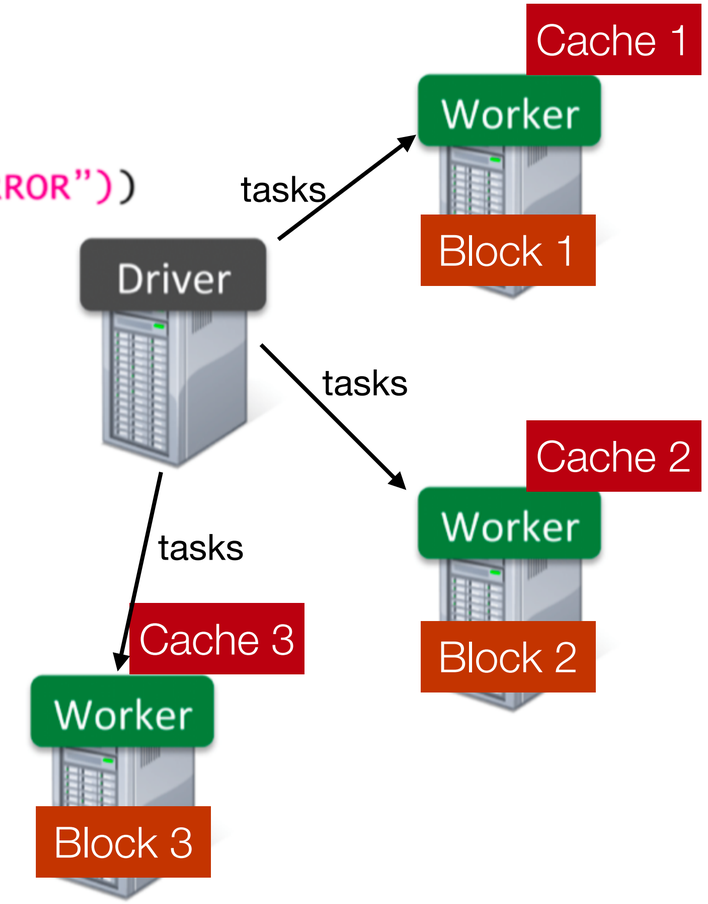


Example: log mining

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()
```

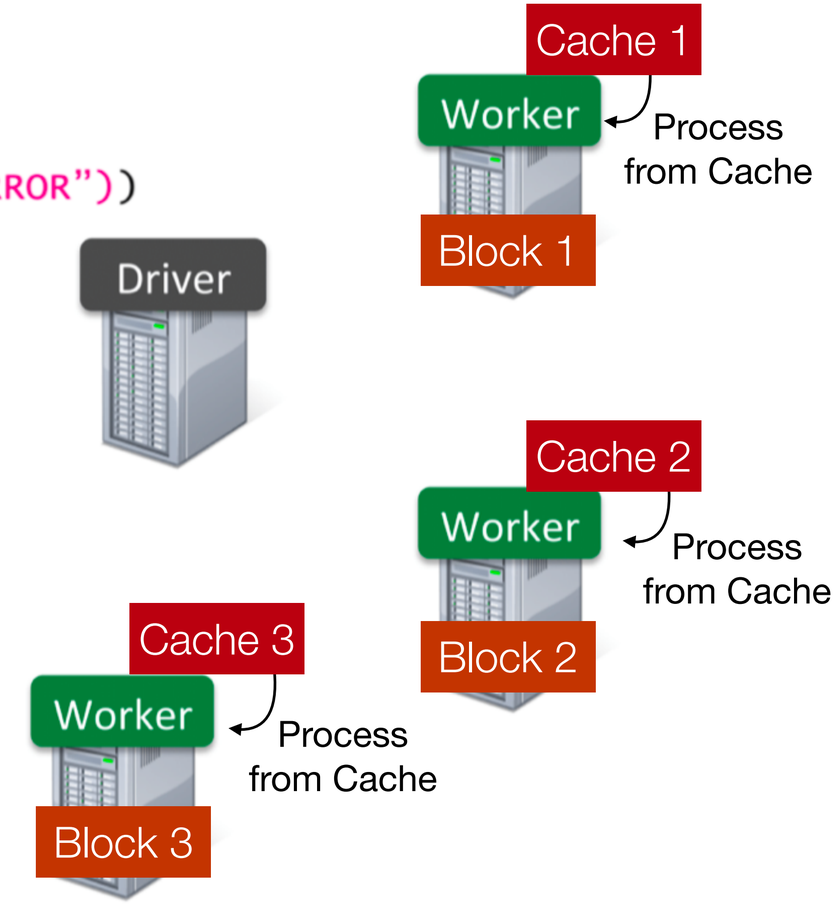


Example: log mining

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()
```

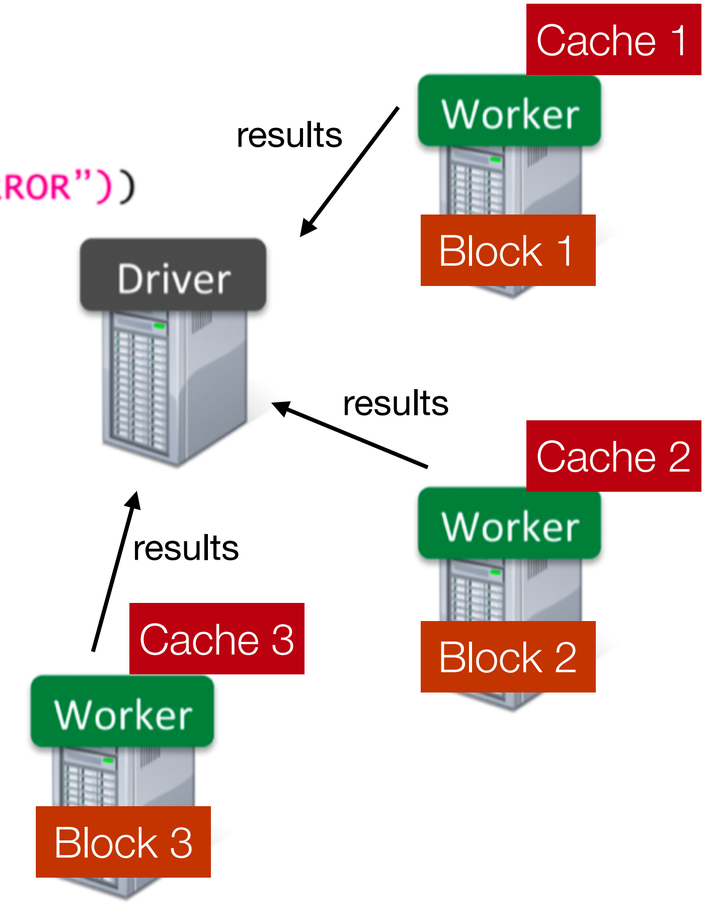


Example: log mining

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()
```



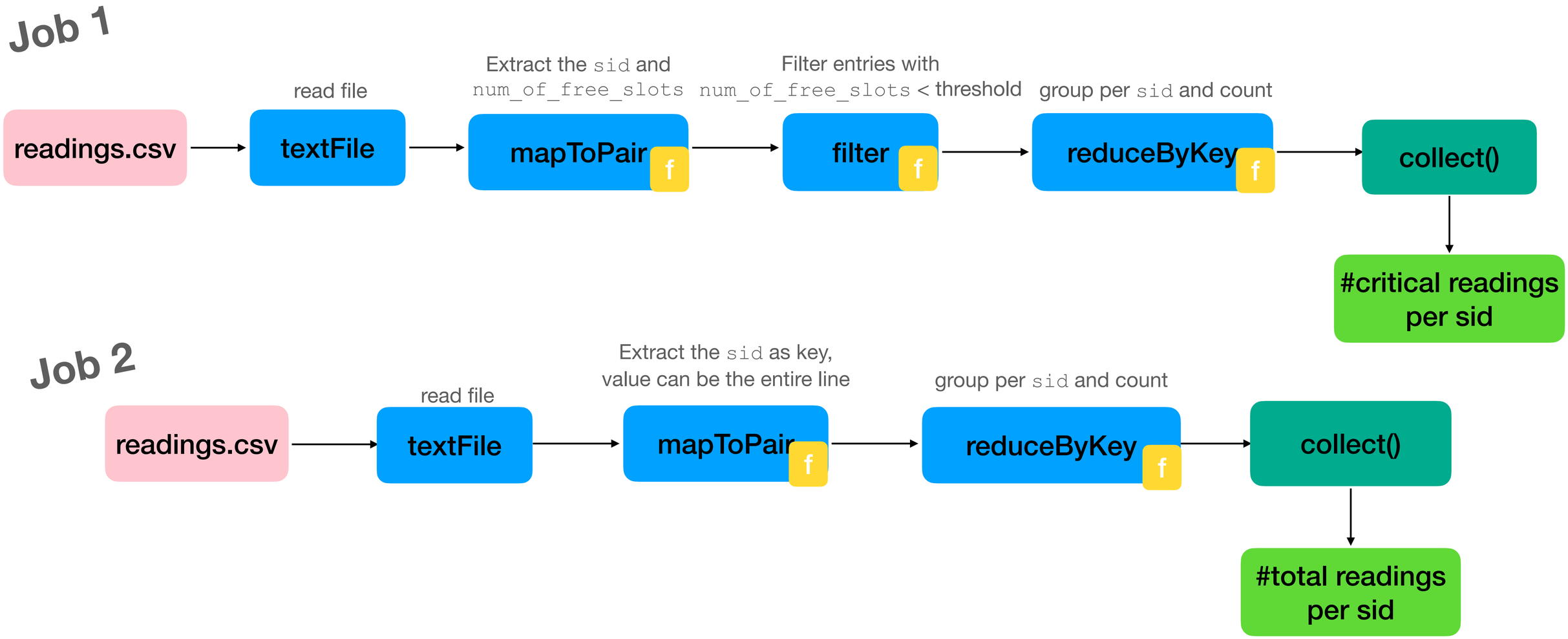
Real-life example — Bike sharing

Real-life example — Bike sharing

- **Problem**: Compute the percentage of critical situations in each station and output the ones > 80%
  - Critical situation: #free slots < b, eg. b=3
  - Percentage of critical situations per station sid=
    (number of critical readings in sid)/(total number of readings in sid)
- **Input**: Occupancy of the stations

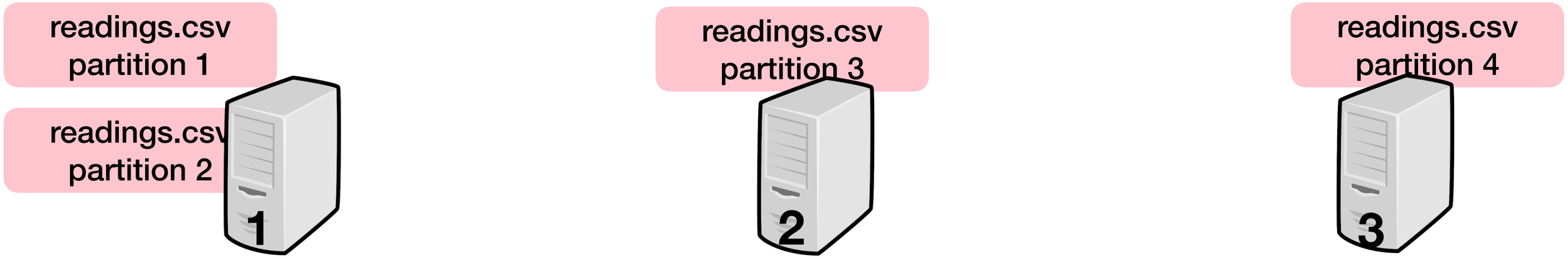
```
sid,date,hour:minute,num_of_bikes,num_of_free_slots
```

Naive solution

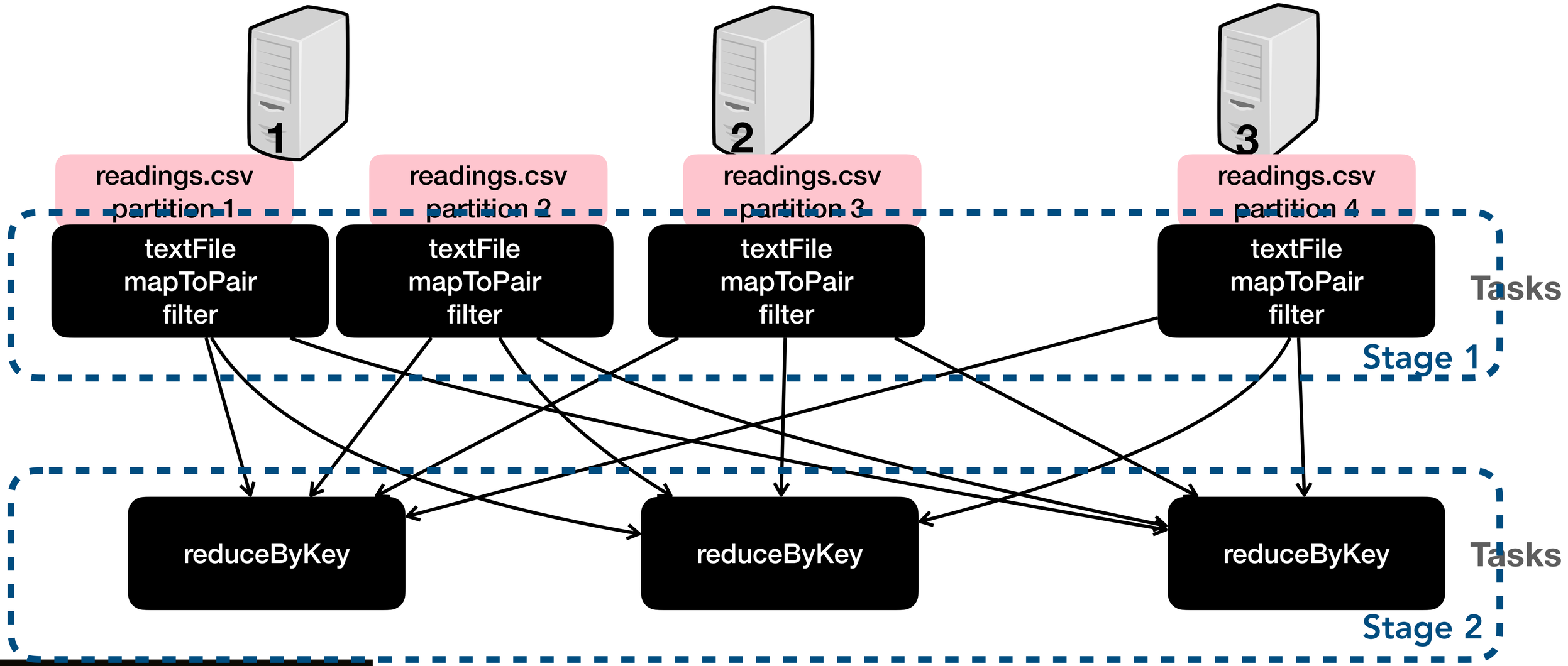


`readings.csv: sid,date,hour:minute,num_of_bikes,num_of_free_slots`

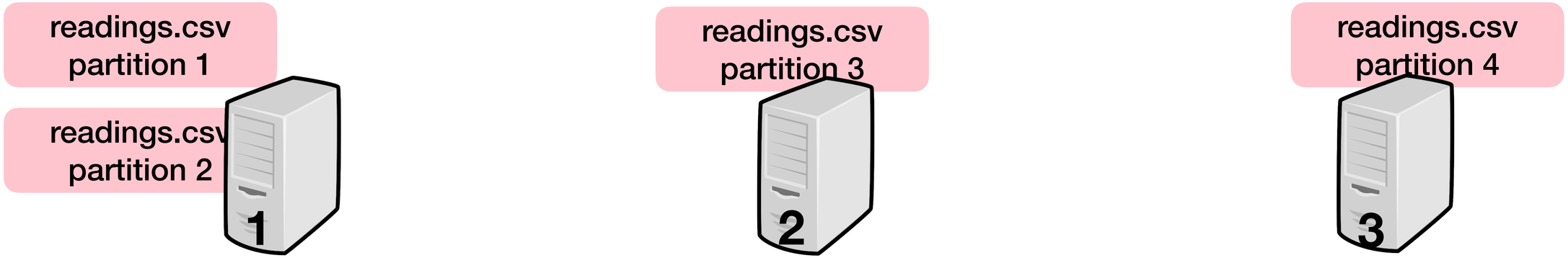
Execution of Job 1 — after action is called



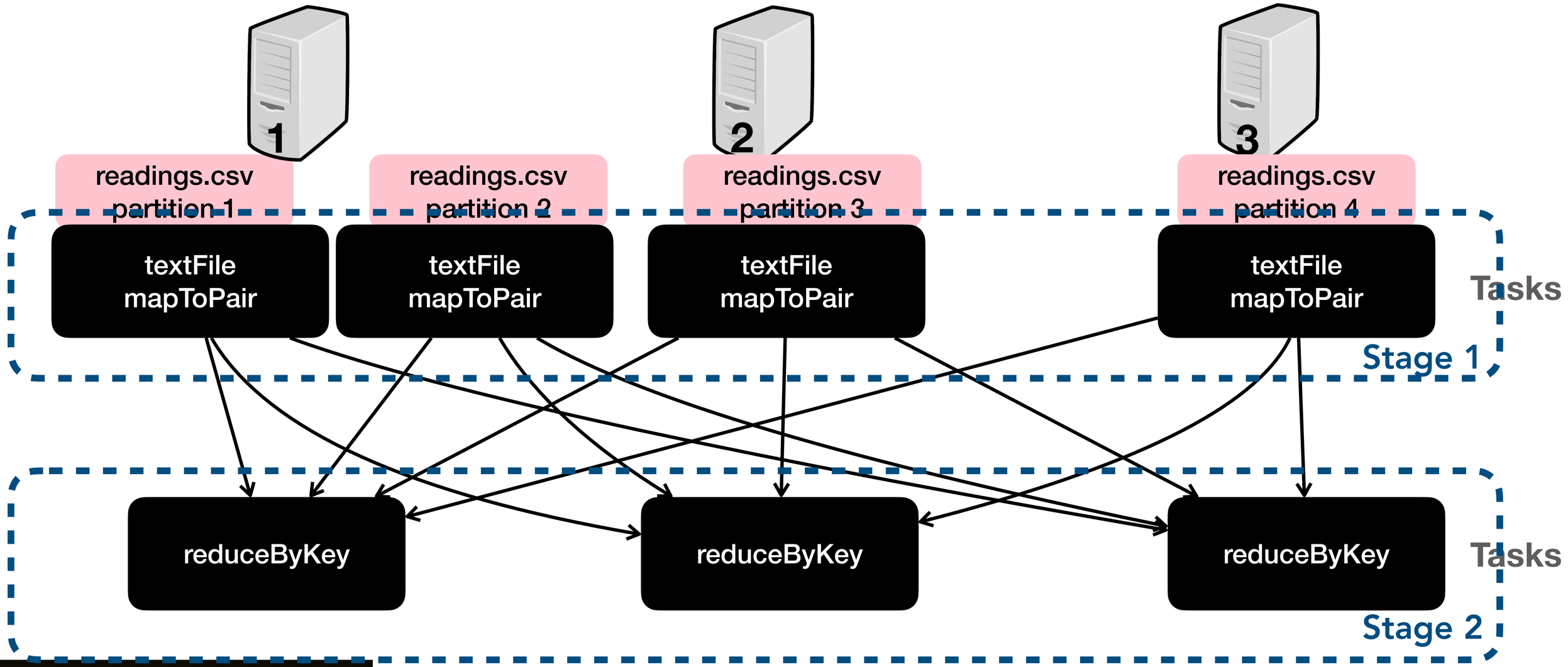
Execution of Job 1 — after action is called



Execution of Job 2 — after action is called

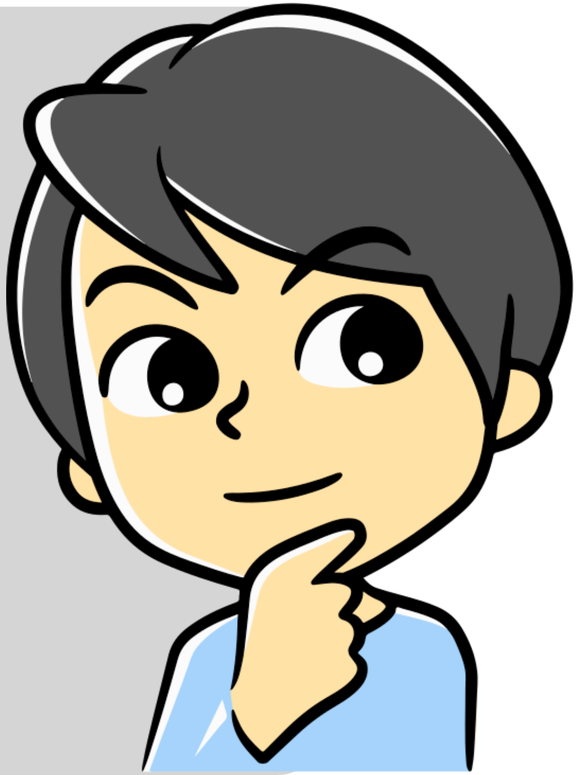


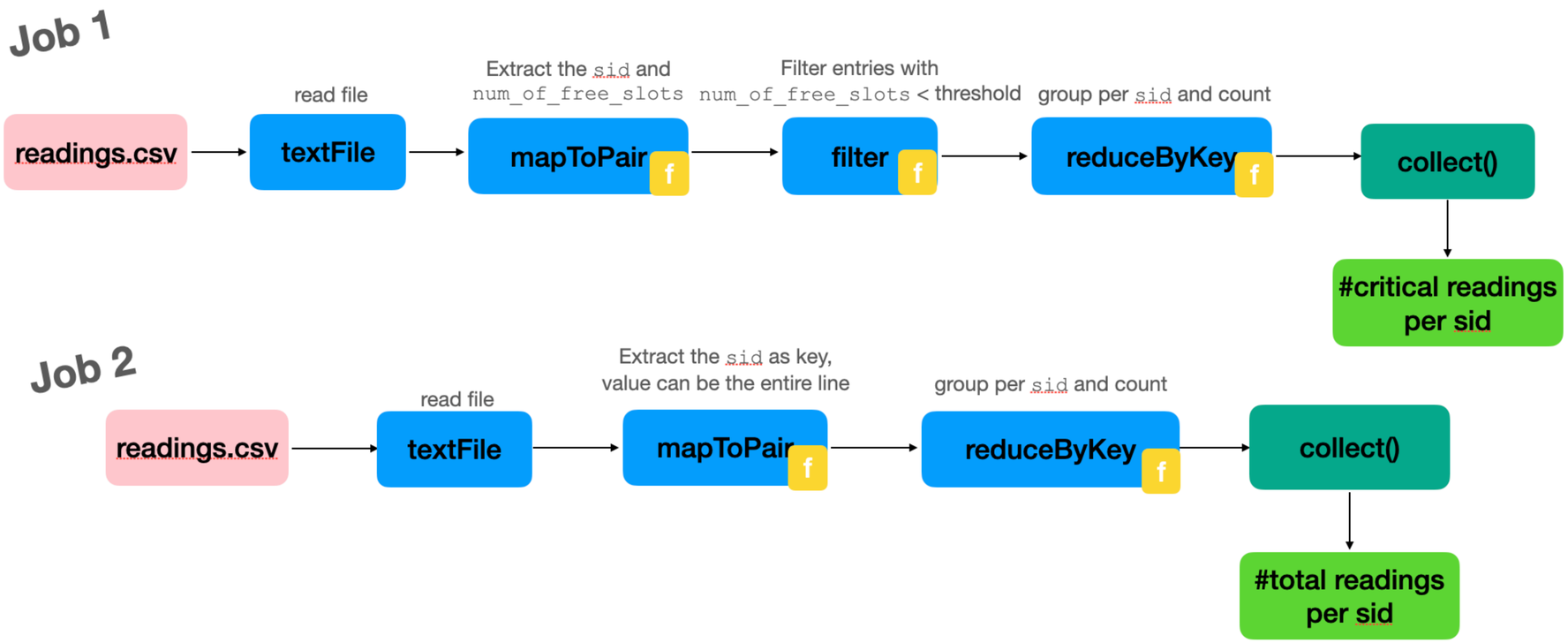
Execution of Job 2 — after action is called



Reflection time

- How can we optimize the previous program?
- Can we solve the problem with 1 job?





`readings.csv: sid,date,hour:minute,num_of_bikes,num_of_free_slots`

Other real-life examples

- ✦ Log analysis
- ✦ Reviews analysis (See assignment!)
- ✦ E-commerce platforms
- ✦ Finance Industry

In this lecture …

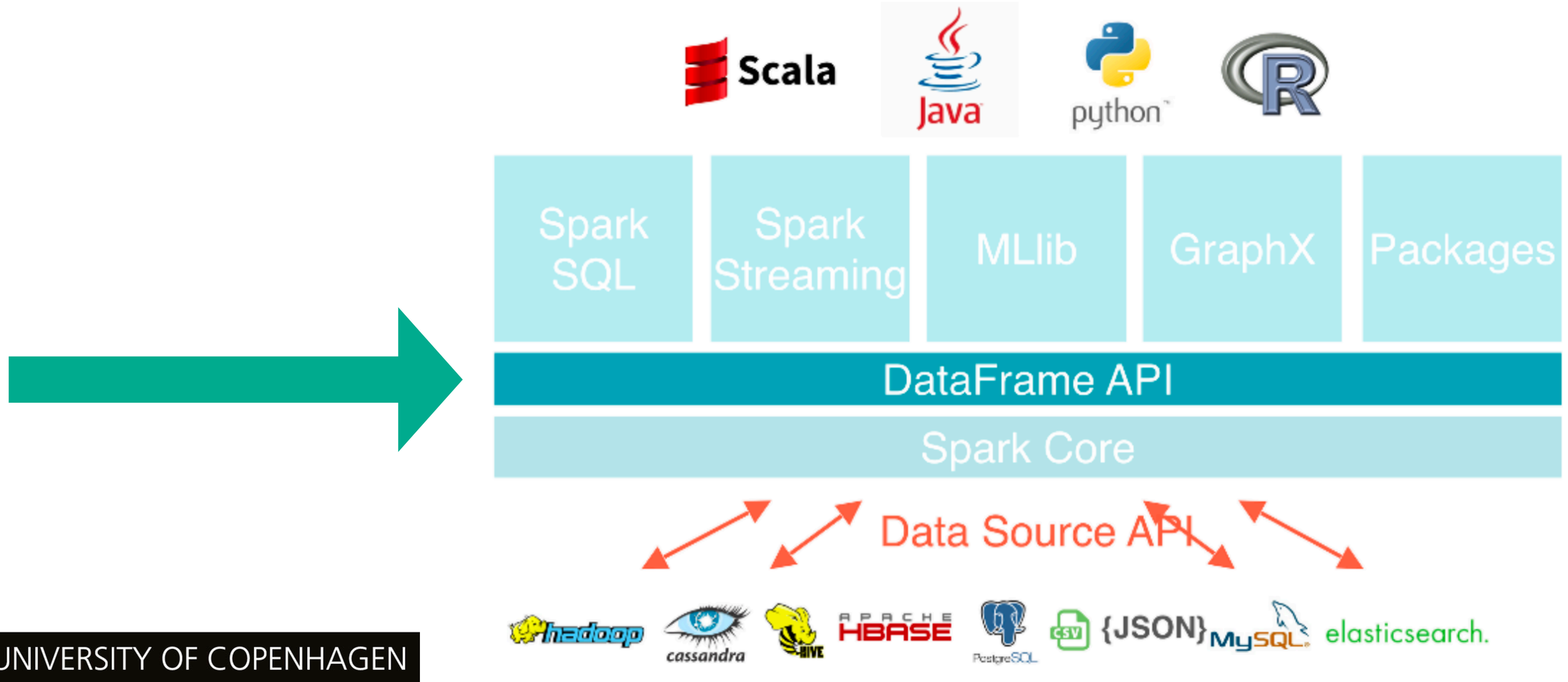
- RDD recall
- Spark under the hood
- **Spark APIs**
  - **DataFrame**
  - **SQL**

Spark's APIs and libraries

- **DataFrames**
- **SparkSQL** for running SQL queries
- **Streaming** for processing real-time data
- **GraphX** for graph processing
- **MLlib** for machine learning
- Can be composed with each other

Apache Spark

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



Spark dataframes

- **Dataframe**: an RDD of rows organised in named columns
- Similar to **relational tables**, pandas dataframes
- Created from file, RDD, etc.
- Variety of data types: vectors, text, images, …
- Special DSL; e.g., data.groupBy(“dept”).avg(“age”)
- Can push some operations to data sources (e.g., DBMS)
- Can be stored efficiently column by column

Dataframes internals

- ✦ Represented internally as a logical dataflow plan
- ✦ Lazy execution
- ✦ Optimised by a query optimizer
- ✦ Schema inference

Schema inference — column names

```
df = spark.read.csv("path/to/your/file.csv", header=True, inferSchema=True)
```

From CSV

```
foo,bar
1,true
2,true
3,false
4,true
5,true
6,false
7,true
```

dataset.csv

foo	bar
integer	boolean
1	true
2	true
3	false
4	true
5	true
6	false
7	true

DataFrame

From JSON

```
{ "foo" : 1, "bar" : true}
{ "foo" : 2, "bar" : true}
{ "foo" : 3, "bar" : false}
{ "foo" : 4, "bar" : true}
{ "foo" : 5, "bar" : true}
{ "foo" : 6, "bar" : false}
{ "foo" : 7, "bar" : true}
```

dataset.json

foo	bar
integer	boolean
1	true
2	true
3	false
4	true
5	true
6	false
7	true

DataFrame

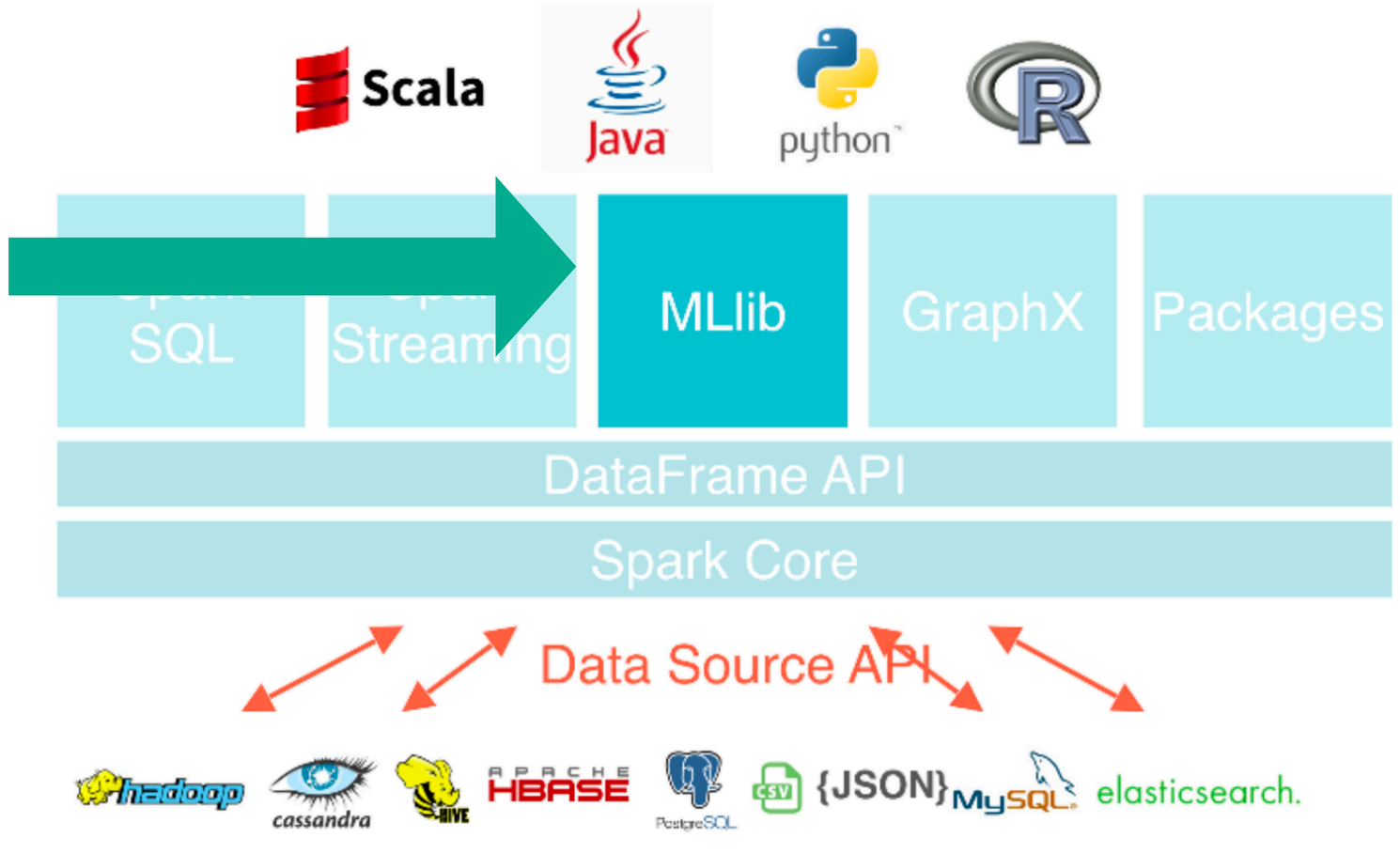
Schema inference — data types

```
df = spark.read.csv("path/to/your/file.csv", header=True, inferSchema=True)
```

- If True: Tells Spark to automatically infer the data types
- If False:
  - Define schema manually
  - If there is no custom schema, all columns as strings by default

Apache Spark

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



Spark's machine learning library

- MLlib: Spark's scalable machine learning library
- Multiple algorithms:
  - Classification: logistic regression, naive Bayes,...
  - Regression: generalized linear regression, survival regression,...
  - Recommendation: alternating least squares (ALS)
  - Clustering: K-means, Gaussian mixtures (GMMs),...
  - …
- End-to-end ML process:
  - Feature transformations: standardization, normalization, hashing,...
  - ML Pipeline construction
  - Model evaluation and hyper-parameter tuning
  - ML persistence: saving and loading models and Pipelines

MLlib example (I)

Create training data

```
training = spark.createDataFrame([
    (1.0, Vectors.dense([0.0, 1.1, 0.1])),
    (0.0, Vectors.dense([2.0, 1.0, -1.0])),
    (0.0, Vectors.dense([2.0, 1.3, 1.0])),
    (1.0, Vectors.dense([0.0, 1.2, -0.5]))], ["label", "features"])
```

Create and fit ML model

```
lr = LogisticRegression(maxIter=10, regParam=0.01)
val model = lr.fit(training)
```

Create test data

```
test = spark.createDataFrame([
    (1.0, Vectors.dense([-1.0, 1.5, 1.3])),
    (0.0, Vectors.dense([3.0, 2.0, -0.1])),
    (1.0, Vectors.dense([0.0, 2.2, -1.5]))], ["label", "features"])
```

Test model

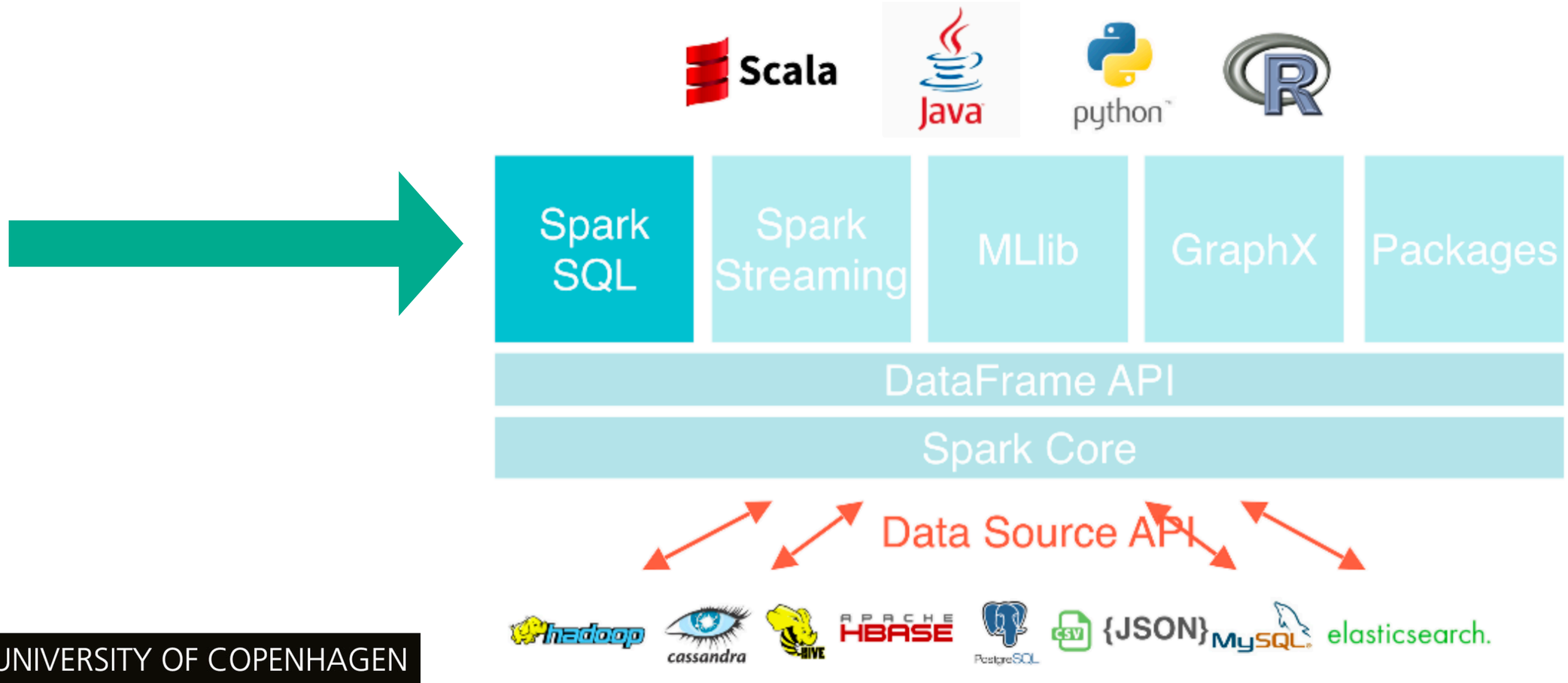
```
pred = model.transform(test)
```

Transform results to pandas

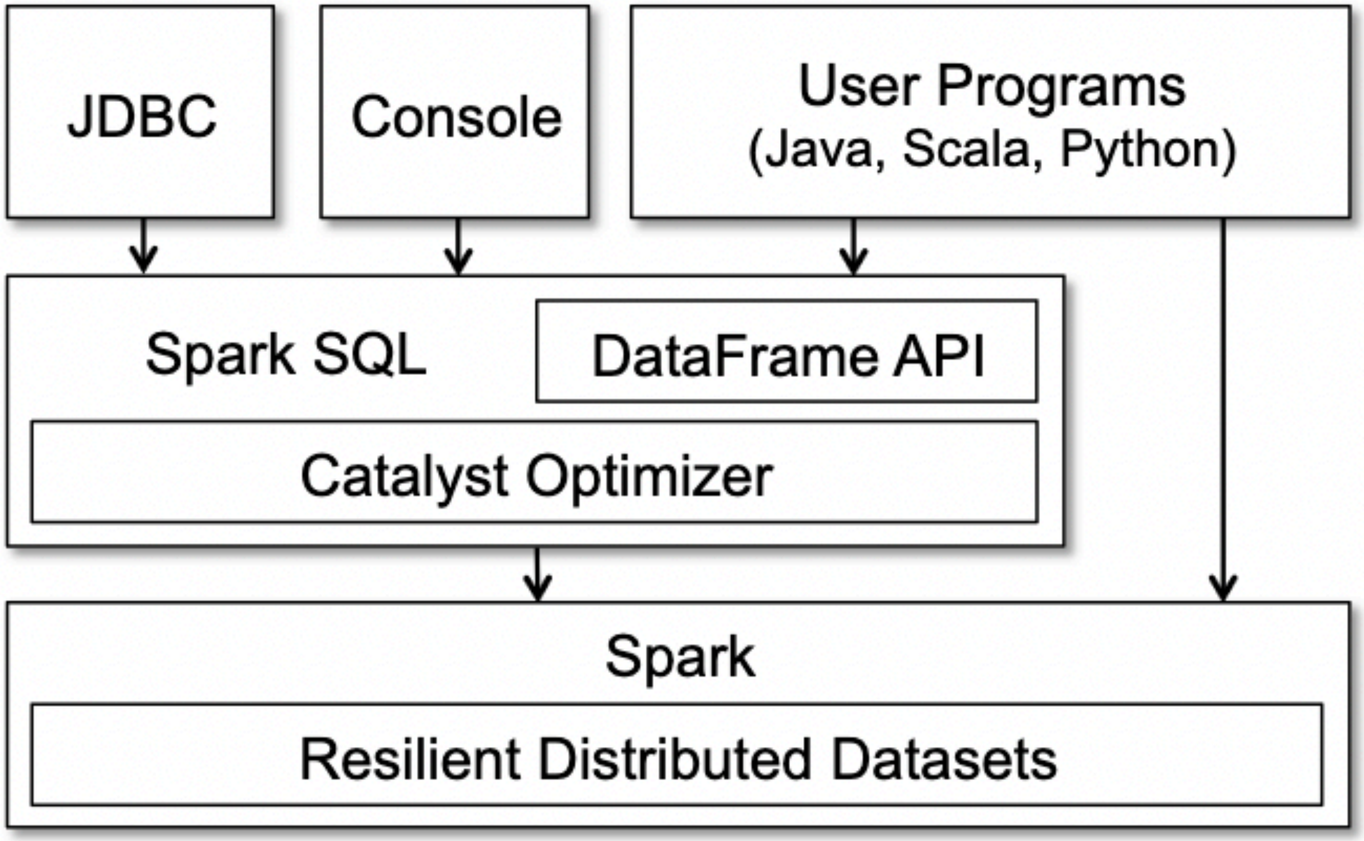
```
pred.toPandas()[['label', 'prediction']]
```

Apache Spark

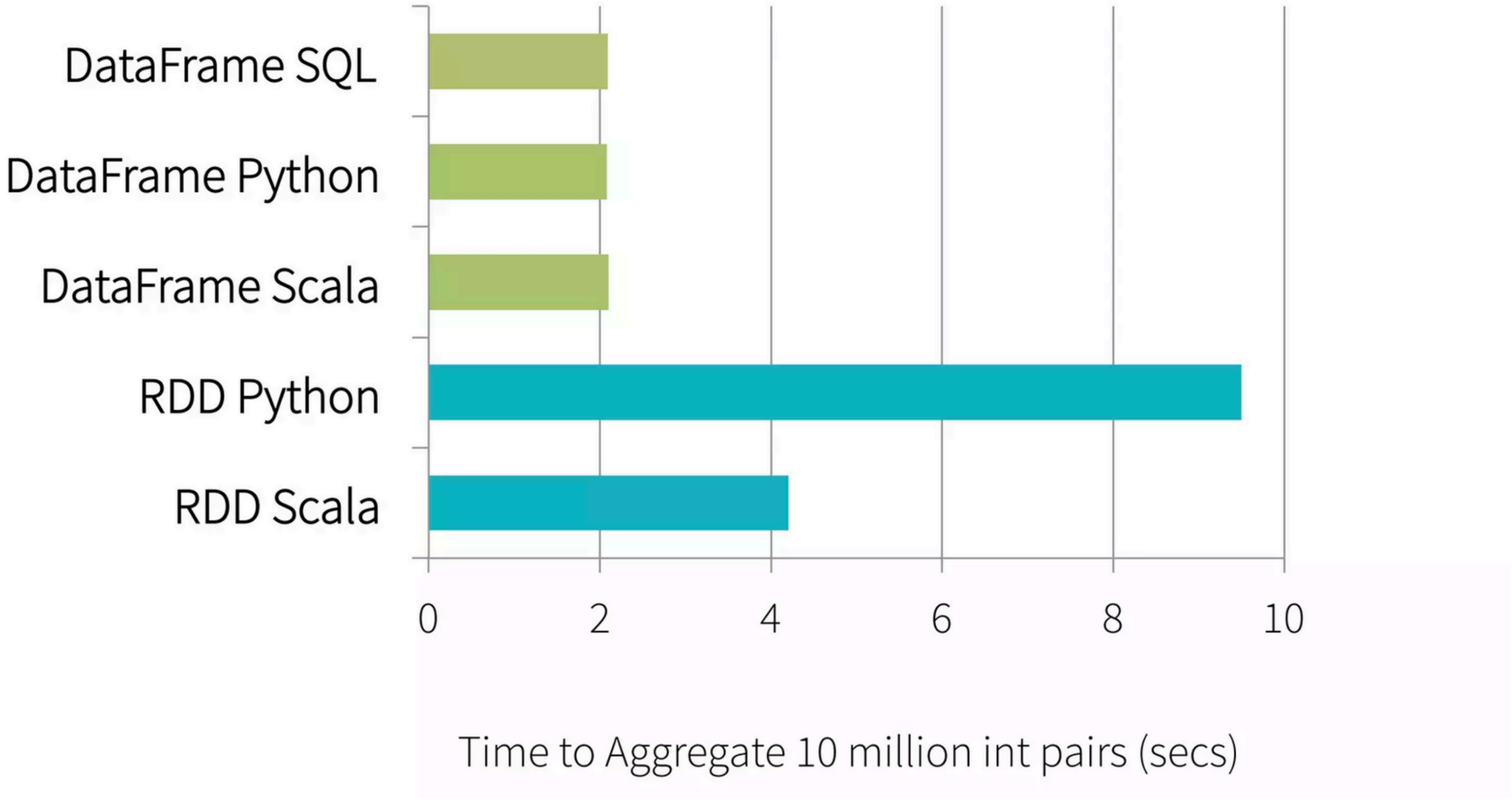
- A “unified analytics engine for large-scale data processing”
- An open-source Apache project (http://spark.apache.org)



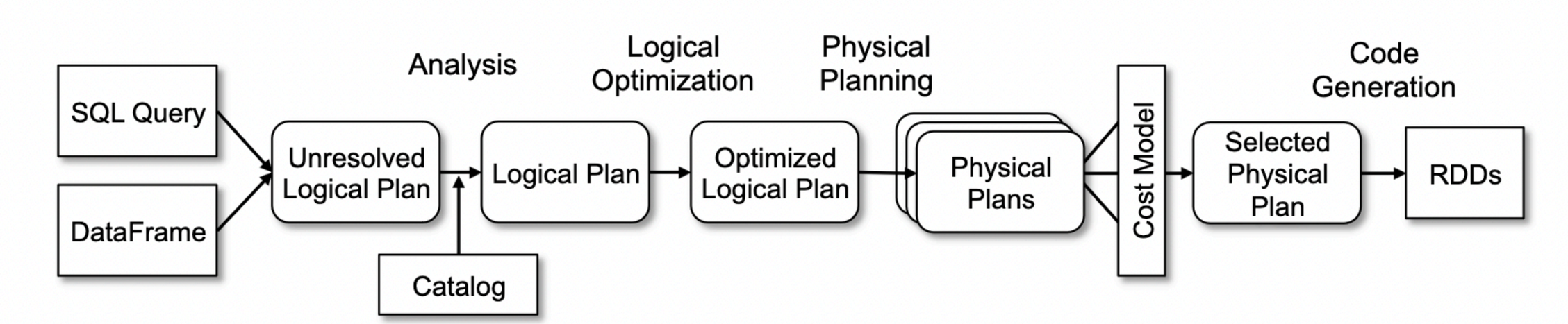
Spark SQL - library on top of Spark



Faster implementation thanks to optimization



Spark's optimization process

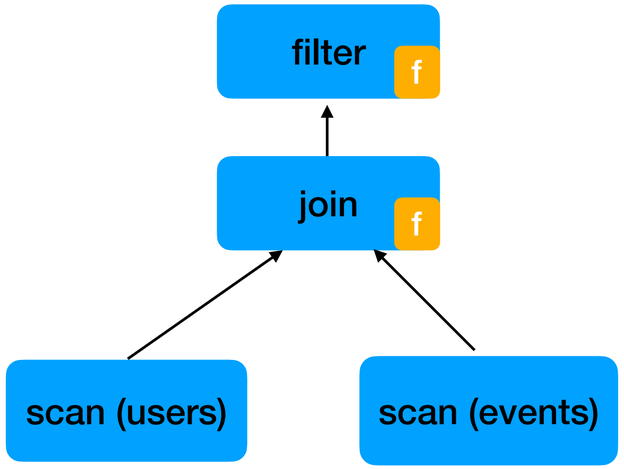


✦ Dataframes and SparkSQL share the same optimisations

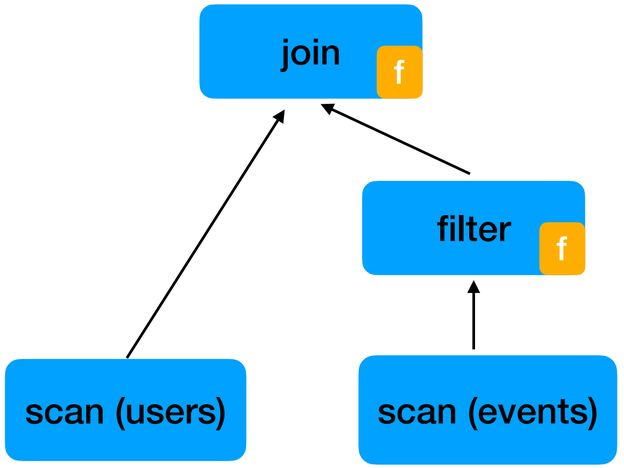
https://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

Spark's optimization process — example

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "01.01.2024")
```



Logical plan

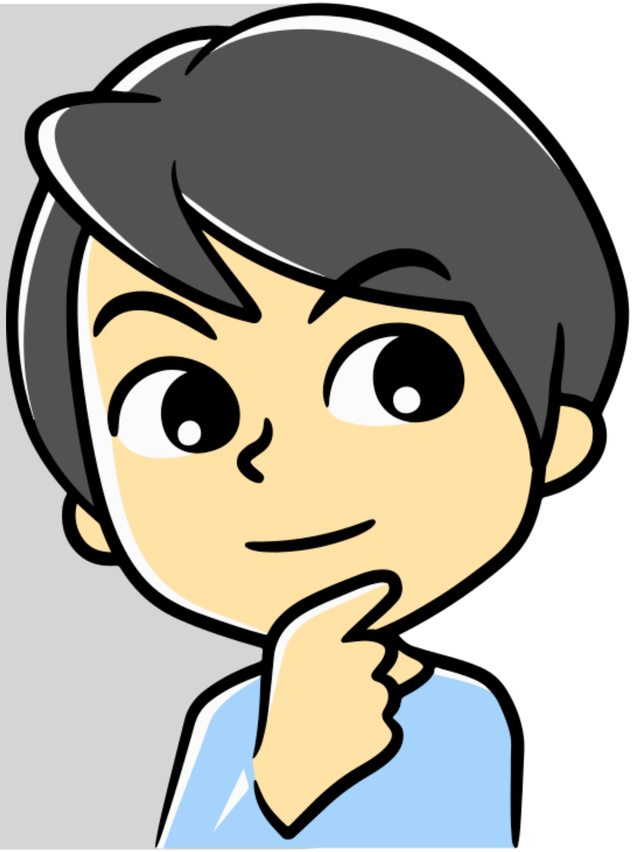


Optimized logical plan

https://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

Quiz Time

- Answer the questions found on LearnIT quiz under the Lecture 5 section



Summary

- Apache Spark under the hood
  - Jobs, Tasks, Stages
  - Caching
  - Lineage
- Apache Spark APIs
  - Opportunities for optimization

Readings

- Papers:
  - Spark: Cluster Computing with Working Sets. Hotcloud 2010
  - Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 2012
  - Spark SQL: Relational Data Processing in Spark. SIGMOD 2015 (optional)

Acknowledgements

- Some slides were taken from Matthias Boehm and Peter Bonz from the courses Large Scale Data Engineering