BIG DATA TRAINING

Intro to Spark

Thoại Nam

Content

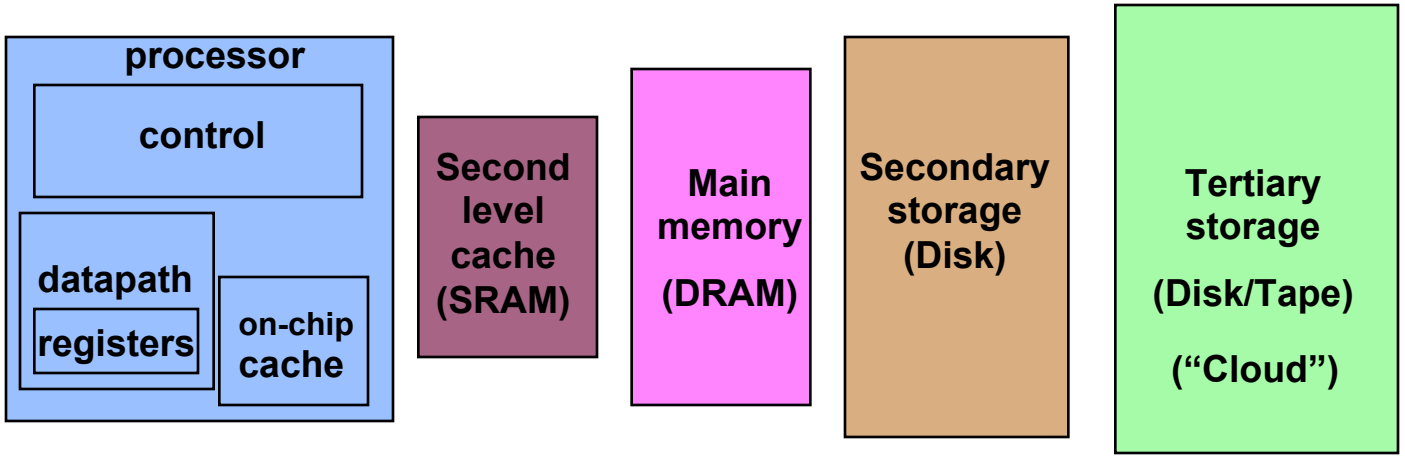
- Shared/Distributed memory
- MapReduce drawbacks
- Spark

Multiprocessor

- Consists of many fully programmable processors each capable of executing its own program
- Shared address space architecture
- Classified into 2 types
  - Uniform Memory Access (UMA) Multiprocessors
  - Non-Uniform Memory Access (NUMA) Multiprocessors

Memory hierarchy

- Most programs have a high degree of locality in their accesses
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality to improve average



Speed	1ns	10ns	100ns	10ms	10sec
Size	KB	MB	GB	TB	PB

Traditional Network Programming

- Message-passing between nodes (MPI, RPC, etc.)
- Really hard to do at scale:
  - How to **split problem across nodes**?
    - Important to consider network and data locality
  - How **to deal with failures**?
    - **If a typical server fails every 3 years, a 10,000-node cluster sees 10 faults/day!**

Data-Parallel Models

- Restrict the programming interface so that the system can do more automatically

"Here's an operation, run it on all of the data"

  - I don't care *where* it runs (you schedule that)
  - In fact, feel free to run it *twice* on different nodes

MapReduce Programming Model

- MapReduce turned out to be an incredibly useful and widely-deployed framework for processing large amounts of data. However, its design forces programs to comply with its computation model, which is:
  - Map: create a <key, value> pairs
  - Shuffle: combine common keys together and partition them to reduce workers
  - Reduce: process each unique key and all of its associated values

MapReduce drawbacks

- Many applications had to run MapReduce over multiple passes to process their data
- All intermediate data had to **be stored back in the file system** (GFS at Google, HDFS elsewhere), which tended to be **slow** since stored data was not just written to disks but also replicated
- The **next MapReduce phase could not start until the previous MapReduce job completed fully**
- MapReduce was also designed to read its data from a distributed file system (GFS/HDFS). In many cases, however, data resides within an SQL database or is streaming in (e.g., activity logs, remote monitoring).

MapReduce programmability

- Most real applications require multiple MR steps
  - Google indexing pipeline: 21 steps
  - Analytics queries (e.g. count clicks &top K): 2 – 5 steps
  - Iterative algorithms (e.g. PageRank): 10's of steps

- Multi-step jobs create spaghetti code
  - 21 MR steps -> 21 mapper and reducer classes

Problems with MapReduce

- MapReduce use cases showed two major limitations:

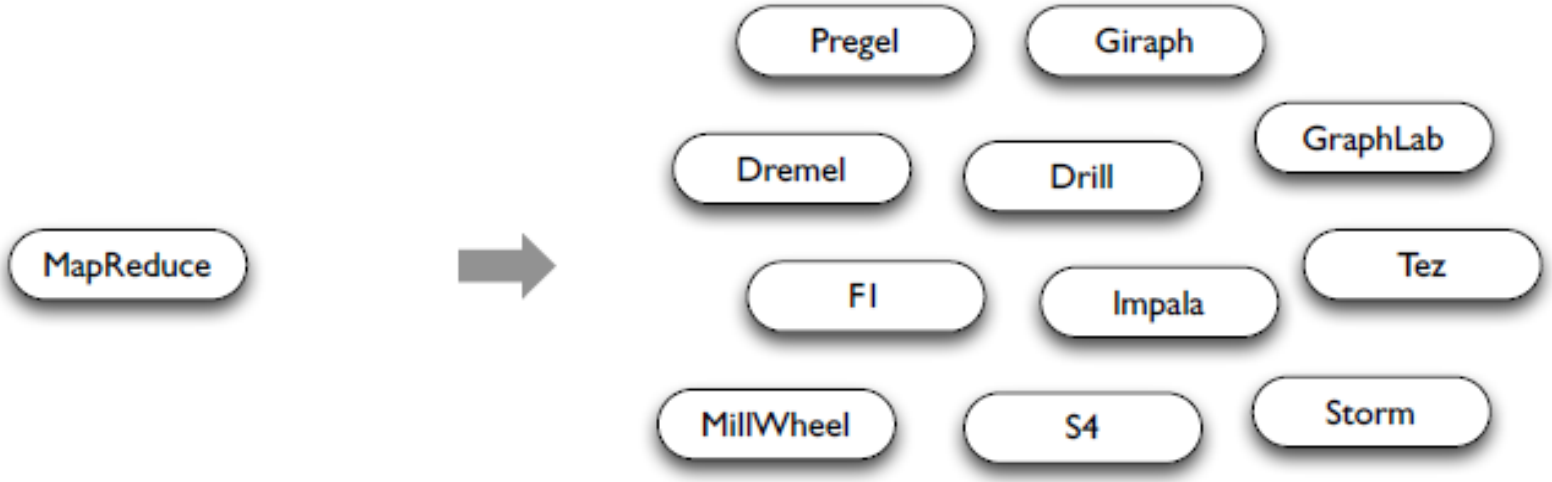
  (1) Difficulty of programming directly in MR

  (2) Performance bottlenecks

- In short, MapReduce doesn't compose well for large-scale applications

- Therefore, people built high level frameworks and specialized systems.

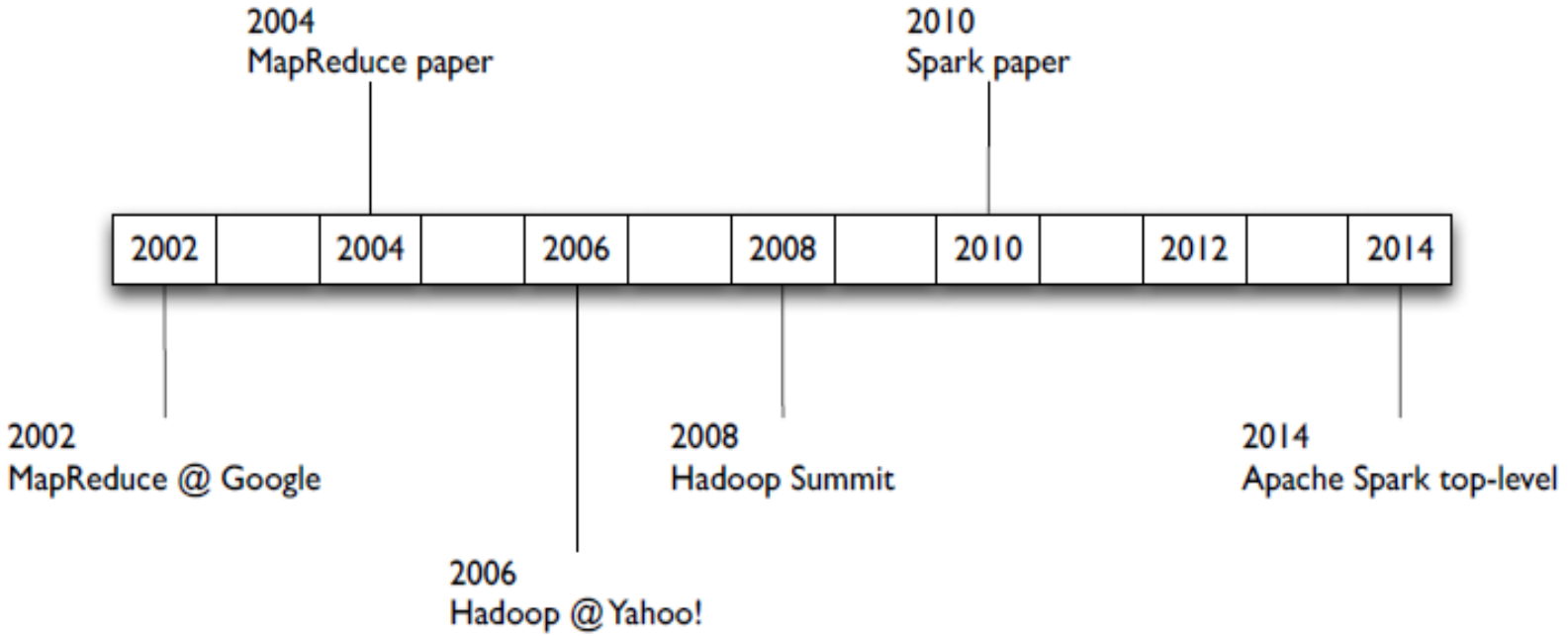
Specialized Systems



Spark


HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
BK
TP. HCM
CENTER OF COMPUTER ENGINEERING

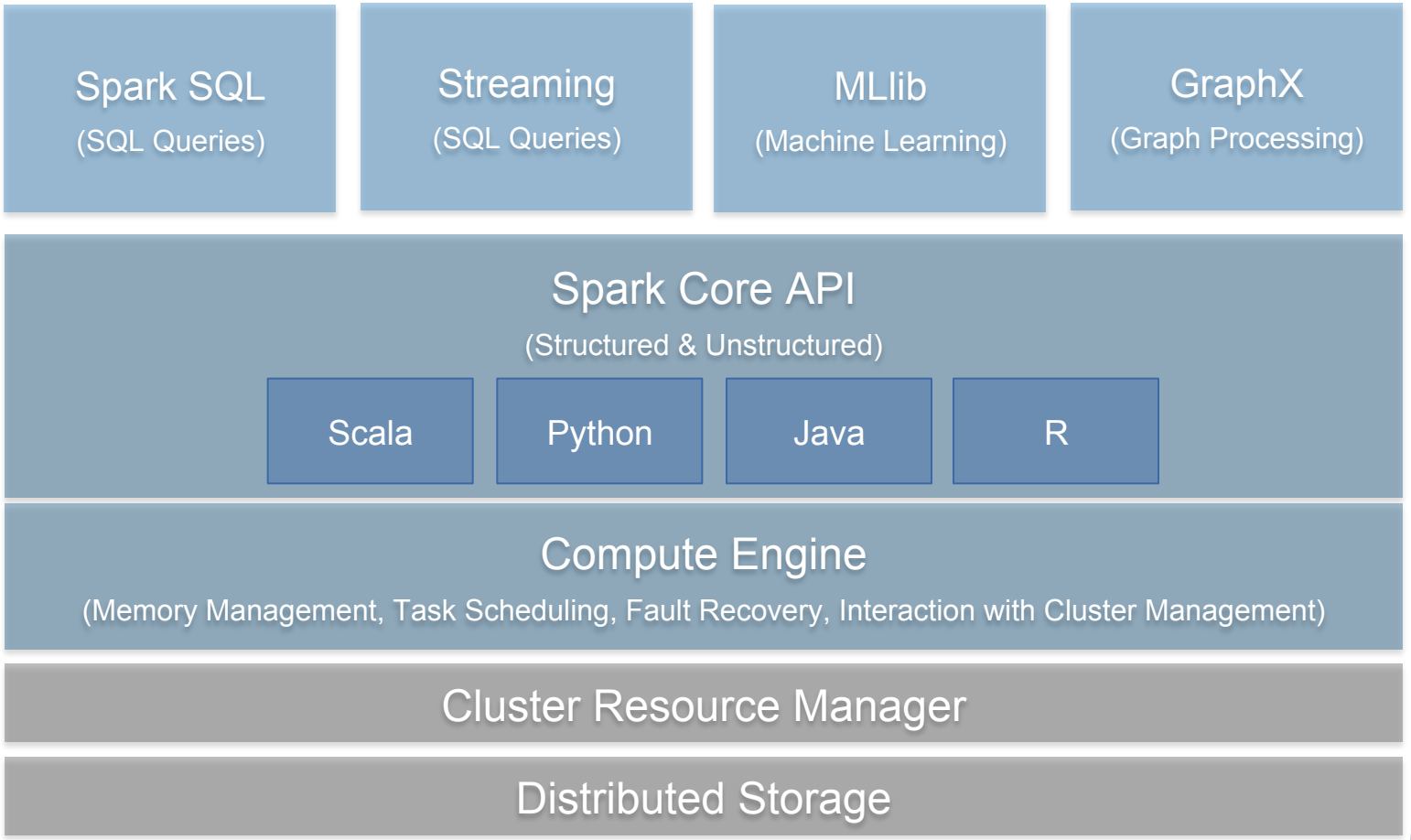
Spark: A Brief History



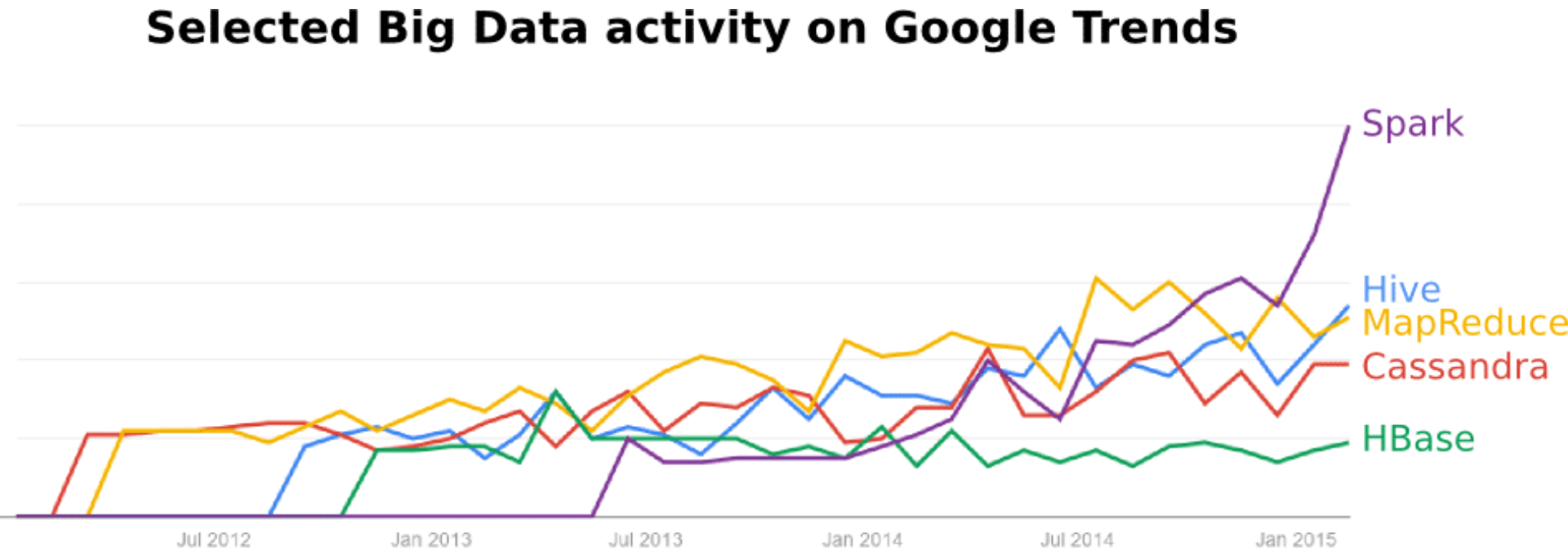
Spark Summary

- Highly flexible and general-purpose way of dealing with big data processing needs
- Does not impose a rigid computation model, and supports a variety of input types
- Deal with text files, graph data, database queries, and streaming sources and not be confined to a two-stage processing model
- Programmers can develop arbitrarily-complex, multi-step data pipelines arranged in an arbitrary directed acyclic graph **(DAG) pattern.**
- Programming in Spark involves defining a sequence of **transformations and actions**
- Spark has support for a map action and a reduce operation, so it **can implement traditional MapReduce** operations but it also **supports SQL queries, graph processing, and machine learning**
- Stores its **intermediate results in memory**, providing for dramatically higher performance.

Spark ecosystem



Spark


Selected Big Data activity on Google Trends
Spark
Hive
MapReduce
Cassandra
HBase
Jul 2012
Jan 2013
Jul 2013
Jan 2014
Jul 2014
Jan 2015

Programmability

```java
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

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

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in 50+ lines of Java MR

```scala
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

Performance

Time to sort 100TB

2013 Record: Hadoop

2100 machines

72 minutes

2014 Record: Spark

207 machines

23 minutes

RDD: Core Abstraction

Write programs in terms of **distributed datasets**

and operations on them

Resilient Distributed Datasets

- **Collections of objects spread across a cluster, stored in RAM or on Disk**
- Built through **parallel transformations**
- **Automatically rebuilt on failure**

Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)

RDD

Resilient Distributed Datasets are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel

Two types:

- *parallelized collections* – take an existing single-node collection and parallel it
- *Hadoop datasets: files on HDFS or other compatible storage*

RDD: Core Abstraction

- An application that uses Spark identifies data sources and the operations on that data. The main application, called the **driver program** is linked with the Spark API, which creates a SparkContext (heart of the Spark system and coordinates all processing activity.) This SparkContext in the driver program connects to a Spark **cluster manager**. The cluster manager responsible for **allocating worker nodes**, **launching executors on them**, and **keeping track of their status**
- Each worker node runs one or more executors. An executor is a process that runs an instance of a Java Virtual Machine (JVM)
- When each executor is launched by the manager, it establishes a connection back to the driver program
- The executor runs tasks on behalf of a specific SparkContext (application) and keeps related data in memory or disk storage
- A task is a transformation or action; the executor remains running for the duration of the driver program.

RDD: Core Abstraction

- **Each worker node runs one or more executors.** An executor is a process that runs an instance of a Java Virtual Machine (JVM)
- When each executor is launched by the manager, it establishes a connection back to the driver program
- The executor runs tasks on behalf of a specific SparkContext (application) and keeps related data in memory or disk storage
- **A task is a transformation or action.** The executor remains running for the duration of the application. This provides a performance advantage over the MapReduce approach since new tasks can be started very quickly
- **The executor also maintains a cache**, which stores frequently-used data in memory instead of having to store it to a disk-based file as the MapReduce framework does
- **The driver goes through the user's program**, which consists of actions and transformations on data and converts that into a series of tasks. The driver then sends tasks to the executors that registered with it
- A task is application code that runs in the executor on a Java Virtual Machine (JVM) and can be written in languages such as Scala, Java, Python, Clojure, and R. It is transmitted as a jar file to an executor, which then runs it.

RDD

- **Data in Spark is a collection of Resilient Distributed Datasets (RDDs).** This is often a huge collection of stuff. Think of an individual RDD as a table in a database or a structured file.
- Input data is organized into RDDs, which will often be partitioned across many computers. RDDs can be created in three ways:

(1) They can be present as any file stored in HDFS or any other storage system supported in Hadoop. This includes Amazon S3 (a key-value server, similar in design to Dynamo), HBase (Hadoop's version of Bigtable), and Cassandra (a no-SQL eventually-consistent database). This data is created by other services, such as *event streams, text logs, or a database*. For instance, the results of a specific query can be treated as an RDD. A list of files in a specific directory can also be an RDD.

RDD

(2) RDDs can be streaming sources using the Spark Streaming extension.

This could be a stream of events from remote sensors, for example.

For fault tolerance, a sliding window is used, where the contents of the stream are buffered in memory for a predefined time interval.

RDD

(3) An RDD can be the output of a transformation function. This allows one task to create data that can be consumed by another task and is the way tasks pass data around.

For example, one task can filter out unwanted data and generate a set of key-value pairs, writing them to an RDD.

This RDD will be cached in memory (overflowing to disk if needed) and will be read by a task that reads the output of the task that created the key/value data.

RDD properties

- **They are immutable.** That means their contents cannot be changed. A task can read from an RDD and create a new RDD but it cannot modify an RDD. The framework magically garbage collects unneeded intermediate RDDs.
- **They are typed.** An RDD will have some kind of structure within in, such as a key-value pair or a set of fields. Tasks need to be able to parse RDD streams.
- **They are ordered.** An RDD contains a set of elements that can be sorted. In the case of key-value lists, the elements will be sorted by a key. The sorting function can be defined by the programmer but sorting enables one to implement things like Reduce operations
- **They are partitioned**. Parts of an RDD may be sent to different servers. The default partitioning function is to send a row of data to the server corresponding to hash(key) mod server count.

RDD operations

- Spark allows two types of operations on RDDs: **transformations and actions**
  - **Transformations** read an RDD and return a new RDD. Example transformations are map, filter, groupByKey, and reduceByKey. Transformations are evaluated lazily, which means they are computed only when some task wants their data (the RDD that they generate). At that point, the driver schedules them for execution
  - **Actions** are operations that evaluate and return a new value. When an action is requested on an RDD object, the necessary transformations are computed and the result is returned. Actions tend to be the things that generate the final output needed by a program. Example actions are reduce, grab samples, and write to file

Spark Essentials: *Transformations*

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)

Spark Essentials: *Actions*

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

Data storage

Spark does not care how data is stored. The appropriate **RDD connector determines how to read data.**

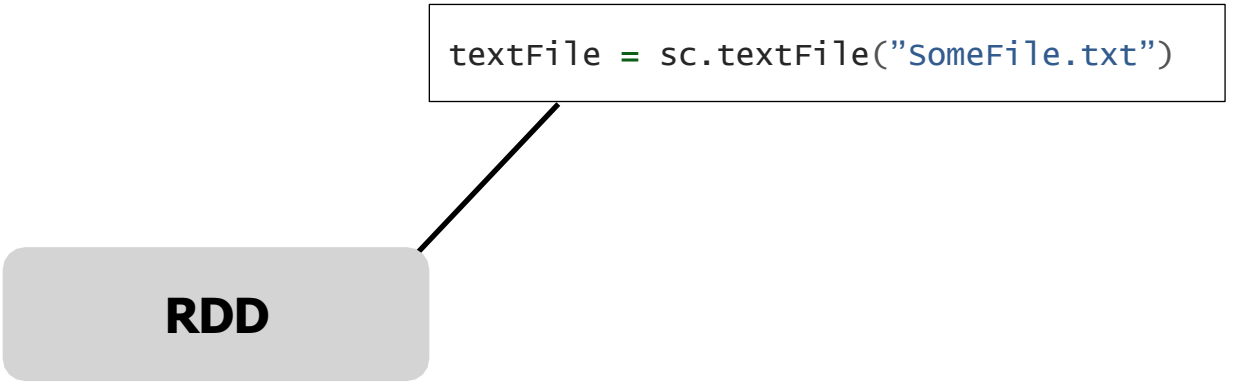
For example, RDDs can be the result of a query in a Cassandra database and new RDDs can be written to Cassandra tables.

Alternatively, RDDs can be read from HDFS files or written to an HBASE table.

Fault tolerance

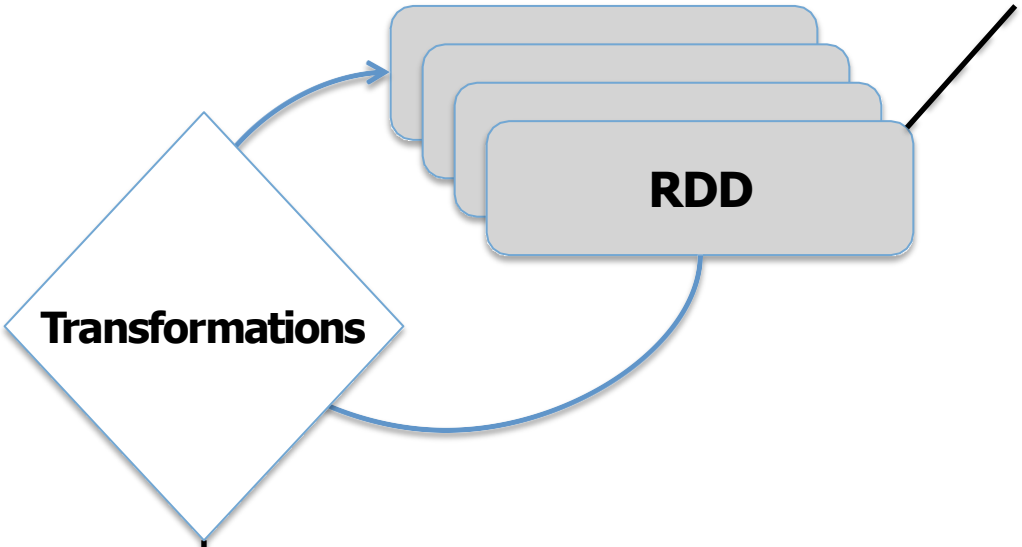
- For each RDD, the driver tracks the sequence of transformations used to create it
- That means every RDD knows which task needed to create it. If any RDD is lost (e.g., a task that creates one died), the driver can ask the task that generated it to recreate it
- The driver maintains the entire dependency graph, so this recreation may end up being a chain of transformation tasks going back to the original data.

Working with RDDs



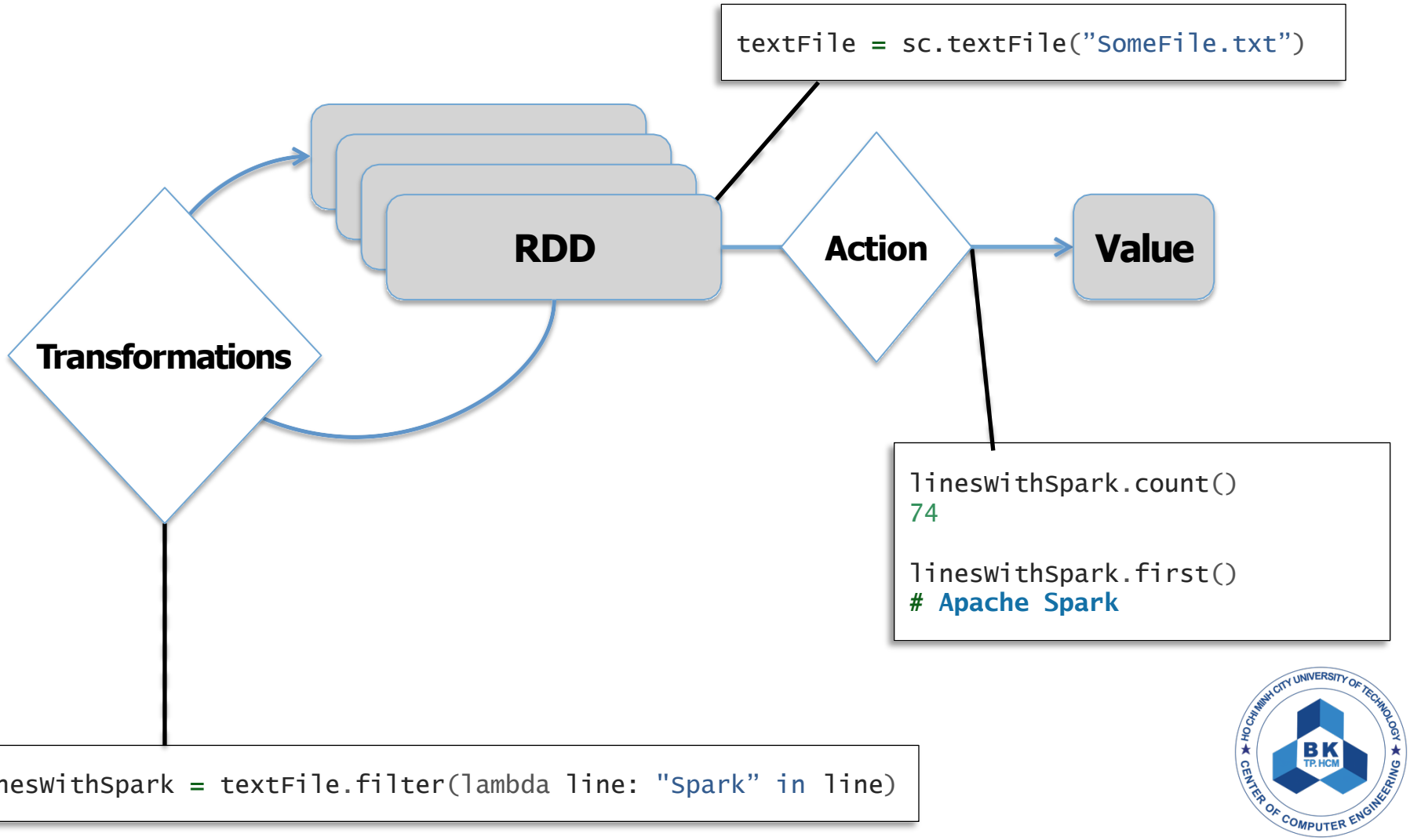
Working with RDDs

```
textFile = sc.textFile("SomeFile.txt")
```




```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

Working with RDDs

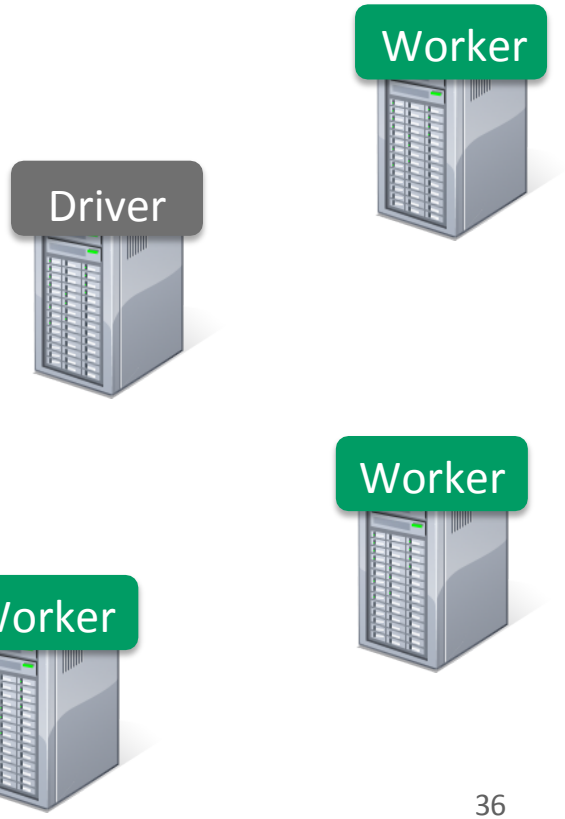


Example: Log Mining

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

Example: Log Mining

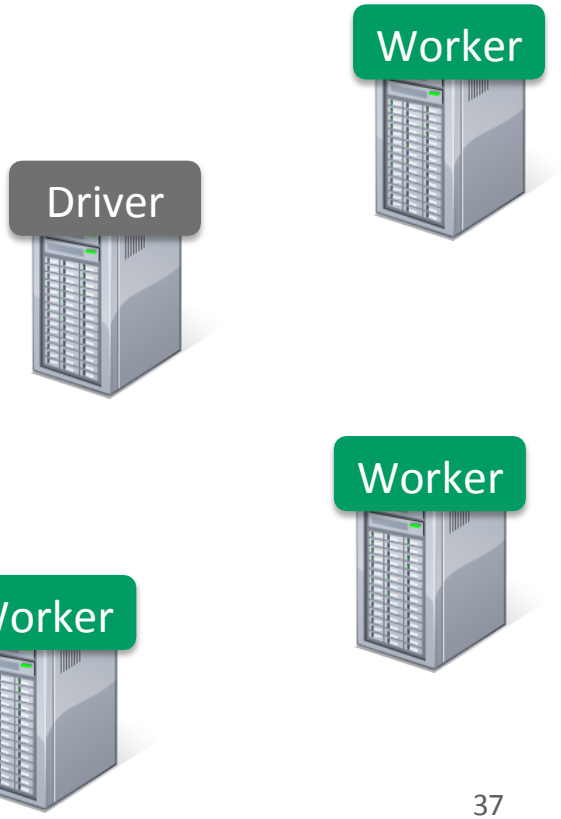
Load error messages from a log into memory, then interactively search for various patterns



Example: Log Mining

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

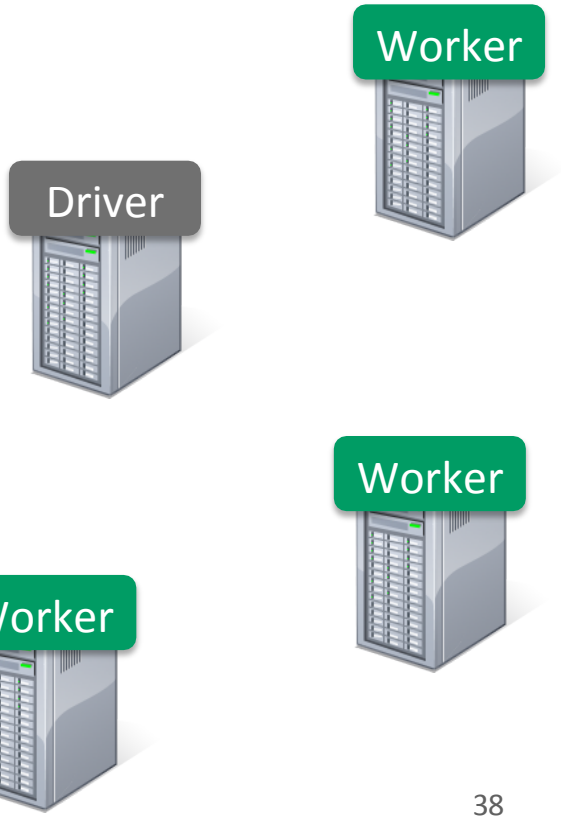
```
lines = spark.textFile(“hdfs://...”)
```



Example: Log Mining

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

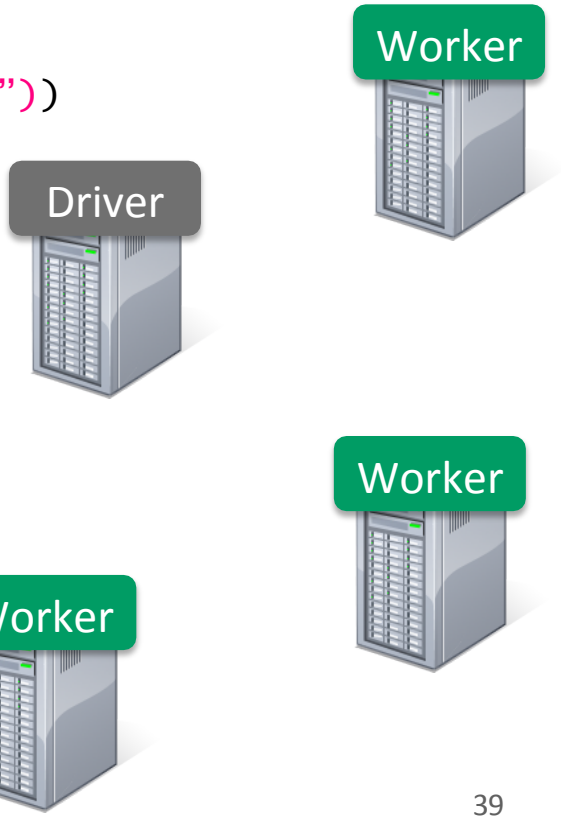
```
lines = spark.textFile(“hdfs://...”)
```



Example: Log Mining

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

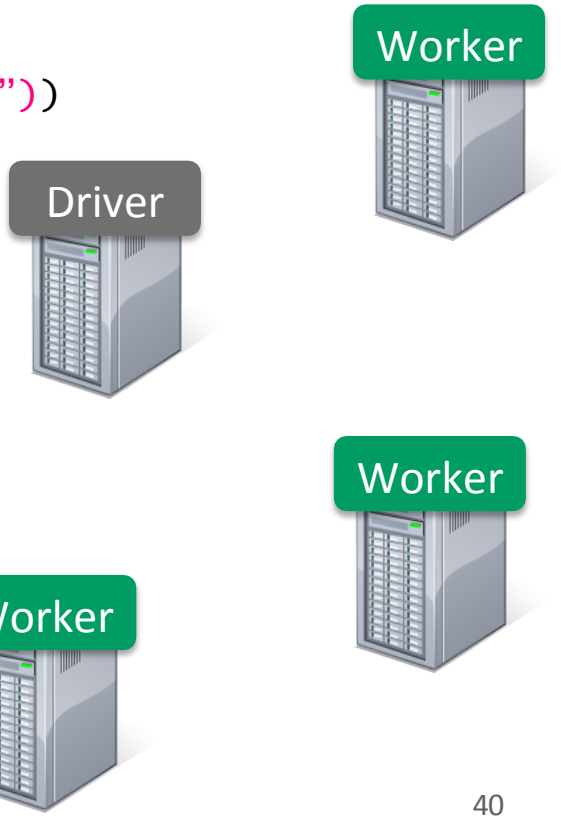
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
```



Example: Log Mining

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

```
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()
```

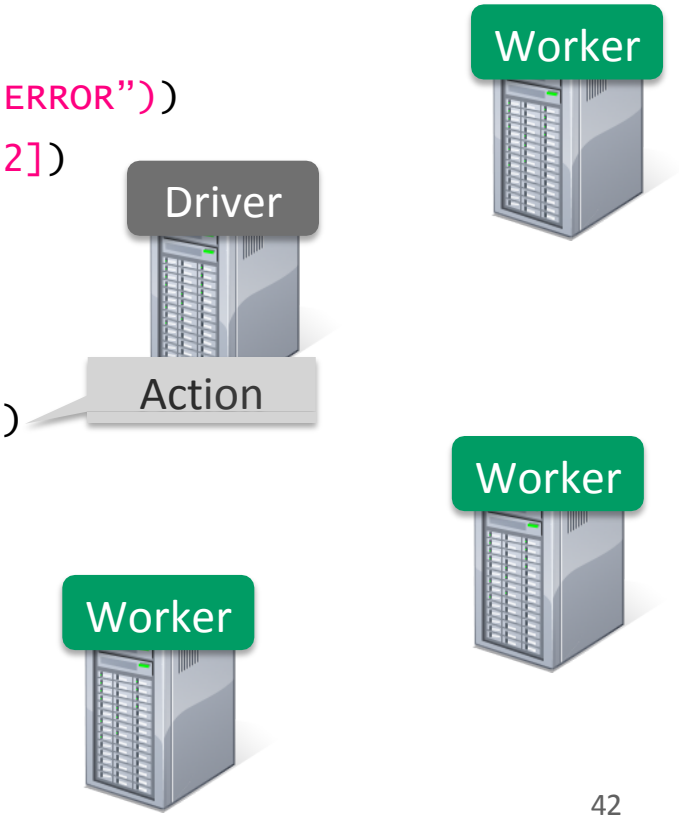
```
messages.filter(lambda s: “mysql” in s).count()
```

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()
```

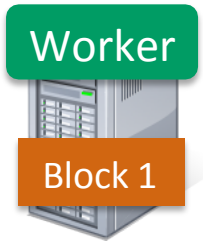


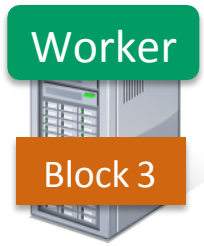
Example: Log Mining

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()
```



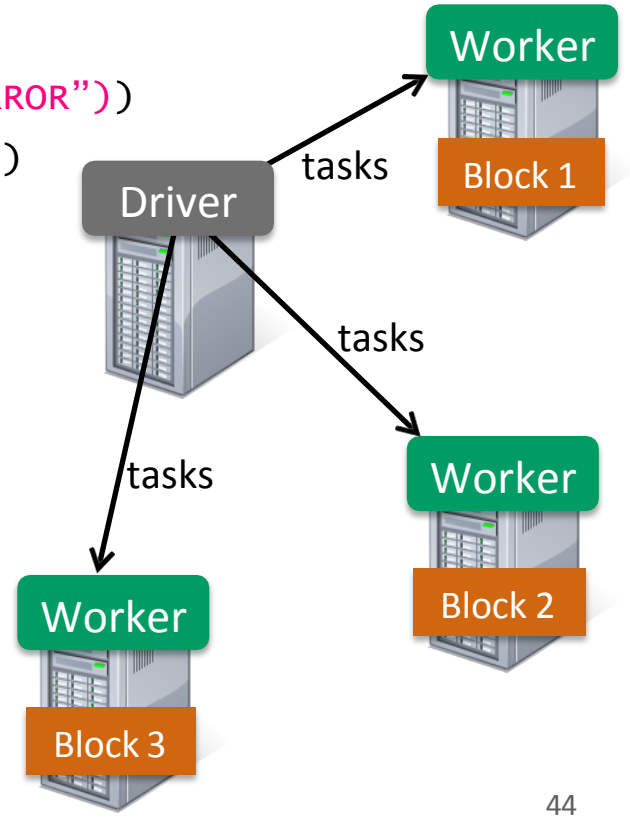


Example: Log Mining

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()
```

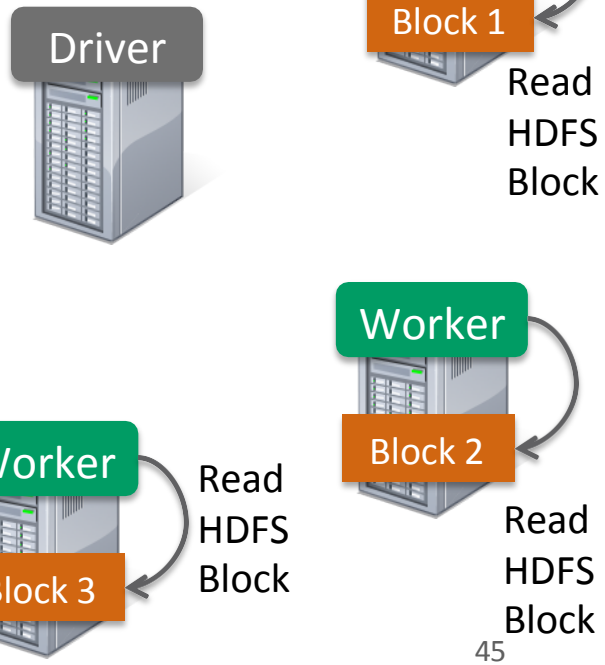


Example: Log Mining

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()
```

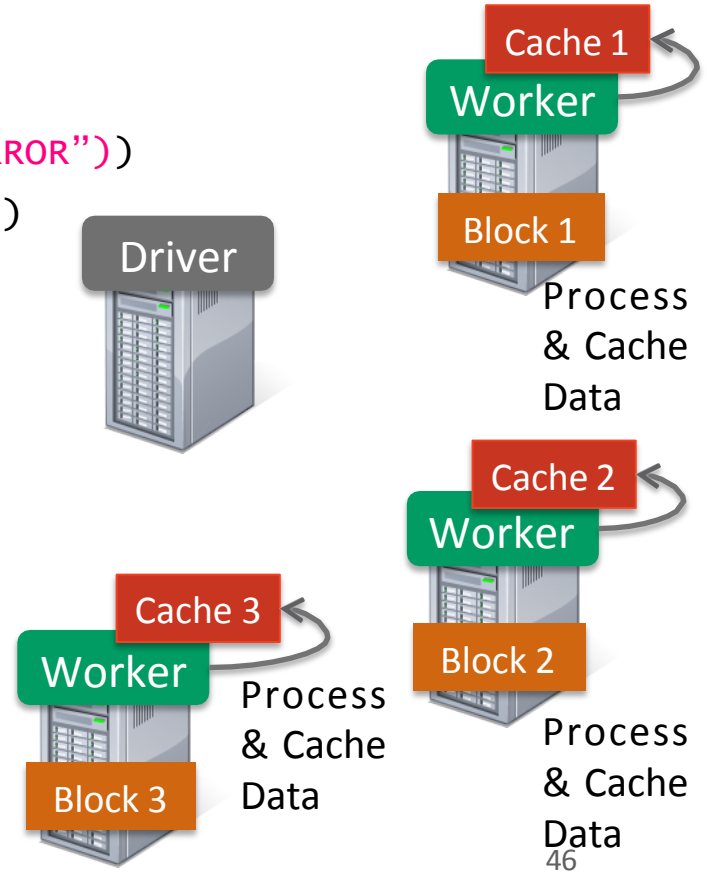


Example: Log Mining

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()
```

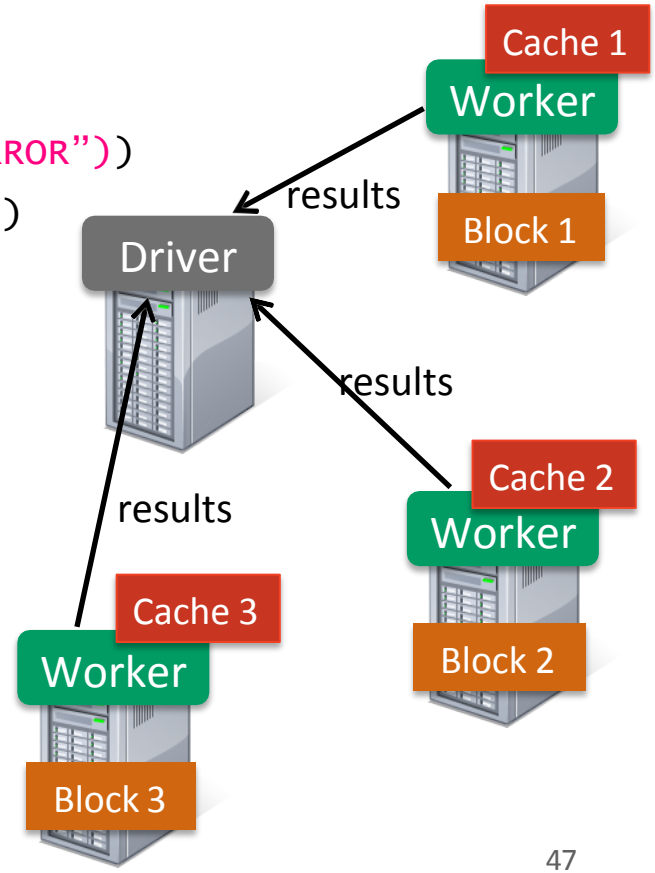


Example: Log Mining

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()
```

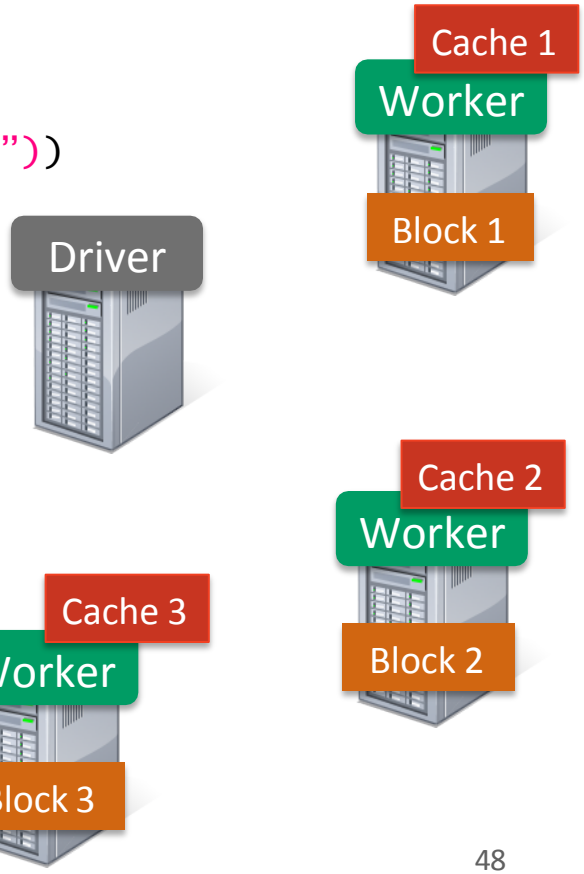


Example: Log Mining

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()
```

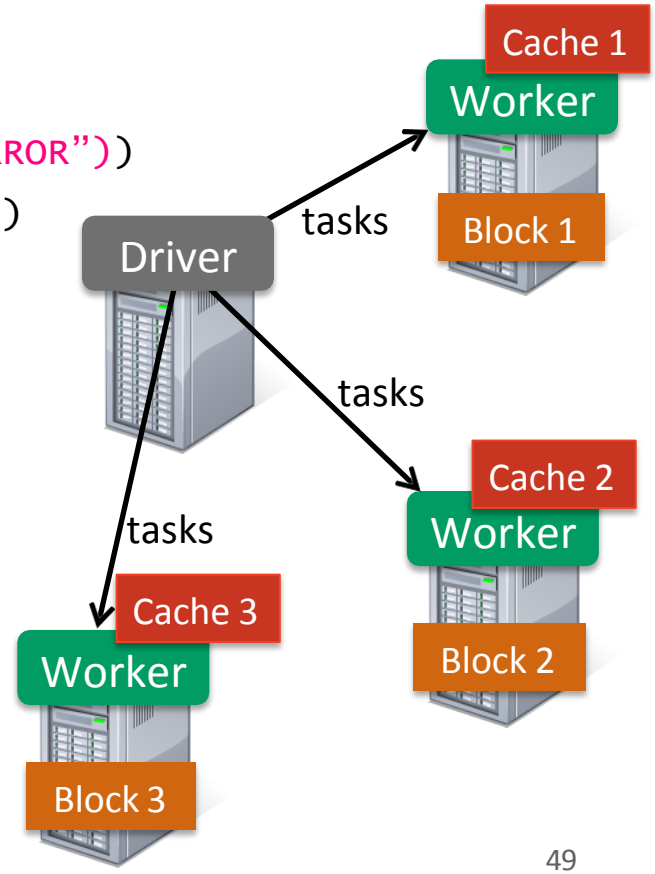


Example: Log Mining

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()
```

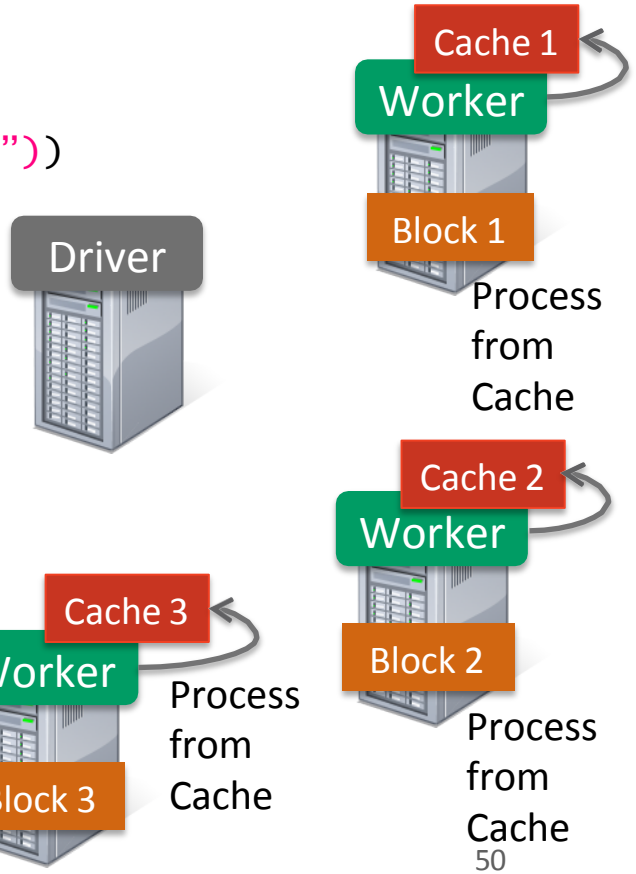


Example: Log Mining

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()
```

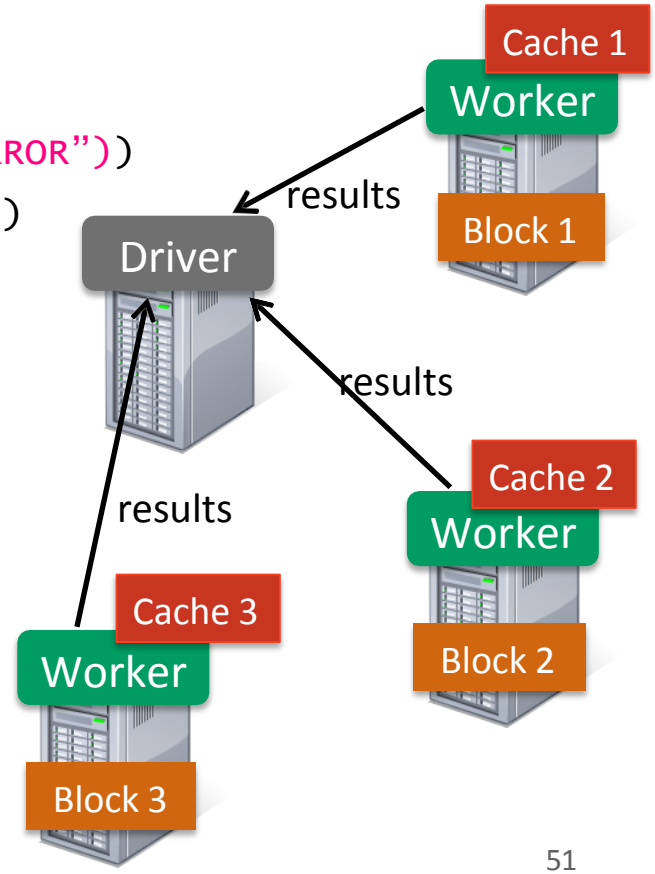


Example: Log Mining

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()
```

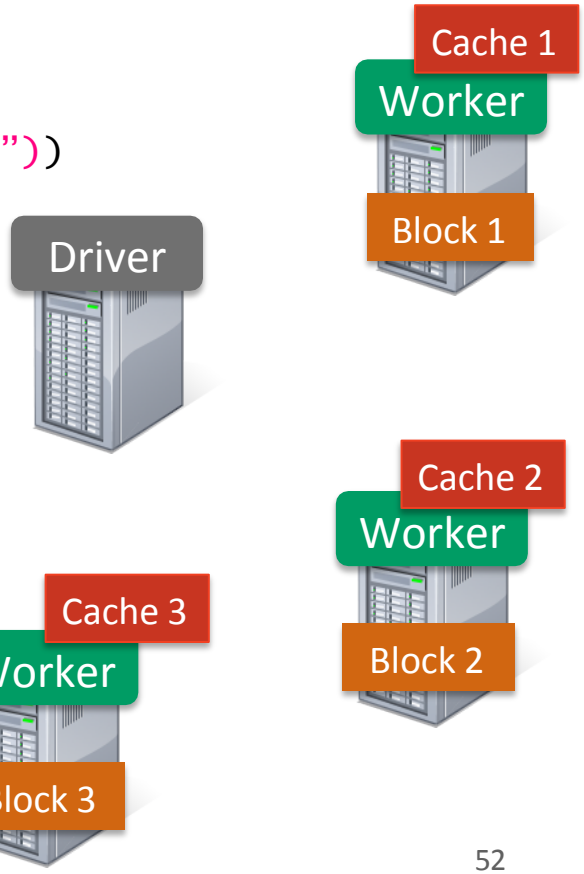


Example: Log Mining

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()
```



Cache your data ➔ Faster Results

Full-text search of Wikipedia

- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk

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

Expressive API

`map` `reduce`

Expressive API

```
map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin
```

```
reduce
count
fold
reduceByKey
groupByKey
cogroup
cross
zip
```

```
sample
take
first
partitionBy
mapWith
pipe
save
```

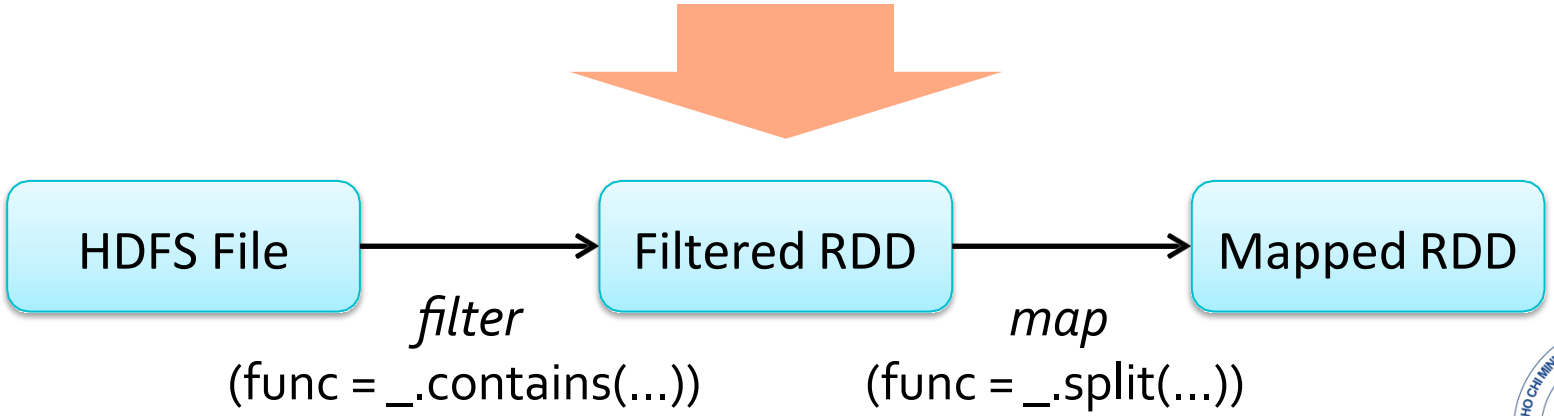
...

Fault Recovery

RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions

Ex:

```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



Fault Recovery Results