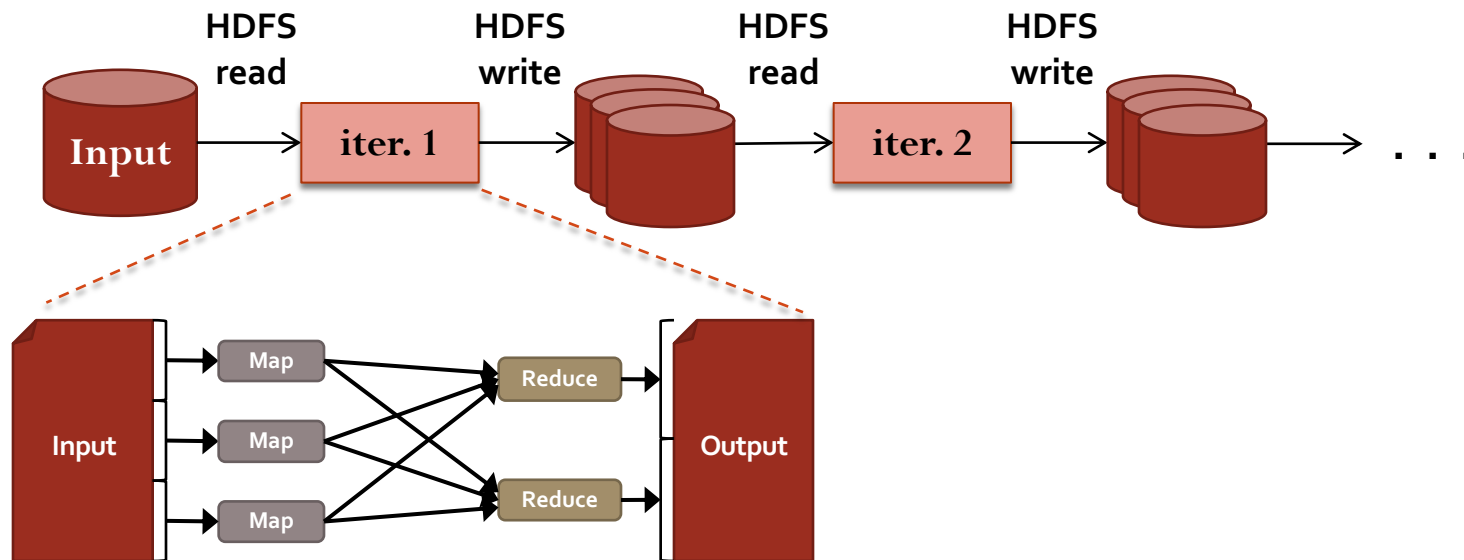


Spark

Riccardo Torlone
Università Roma Tre



Limitations of Map Reduce



- Slow due to disk IO, high communication, and serialization
Inefficient for:
 - Iterative algorithms (Machine Learning, Graphs & Network Analysis)
 - Interactive Data Mining (R, Excel-like computations, Ad hoc Reporting, Searching)

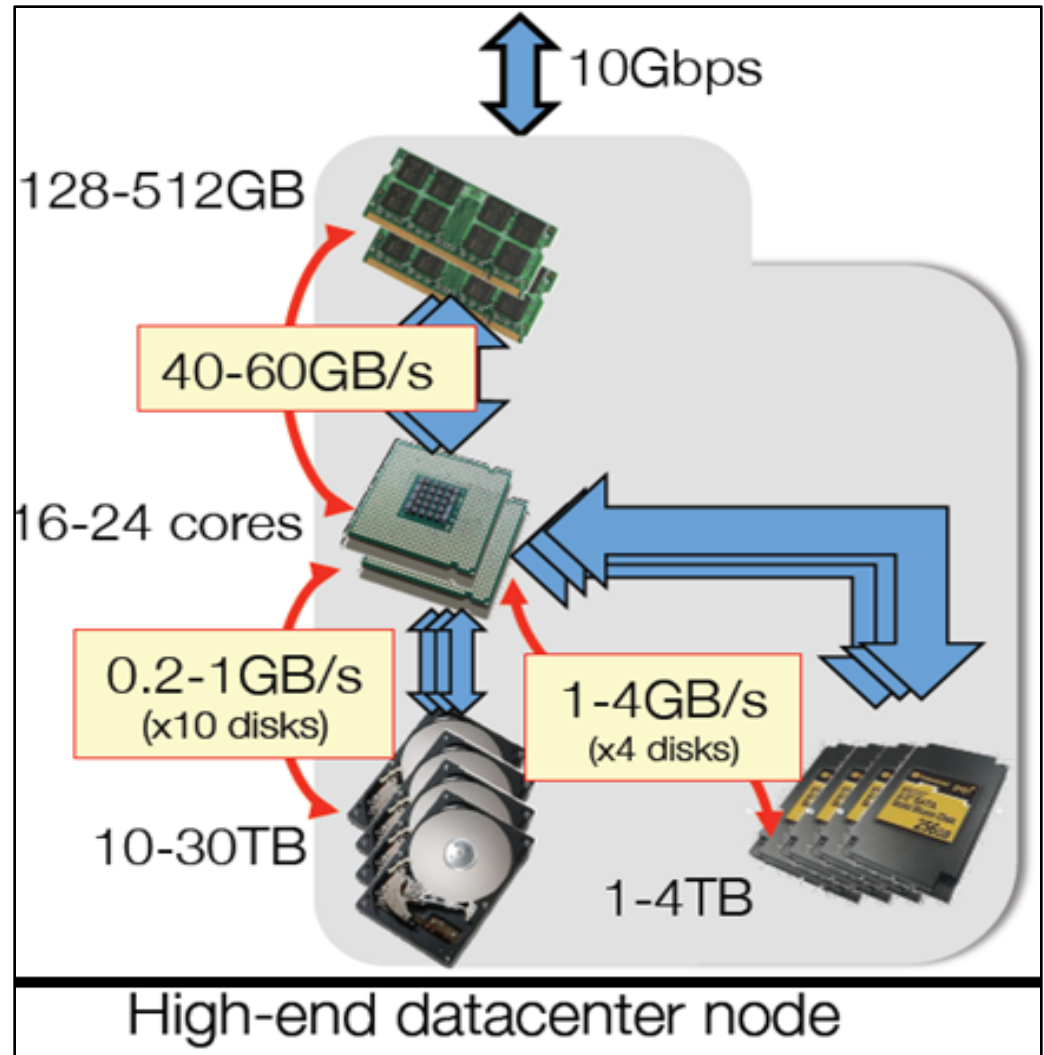
MapReduce: Weaknesses and limitations

- Efficiency
 - High communication cost
 - Frequent writing of output to disk
 - Limited exploitation of main memory
- Programming model
 - Hard to implement everything as a MR program
 - Multiple MR steps can be needed also for simple operations
 - Lack of control structures and data types
- Real-time processing
 - A MR job requires to scan the entire input
 - Stream processing and random access impossible

Solutions?

Leverage to memory:

- replace disks with SSD
- load data into memory



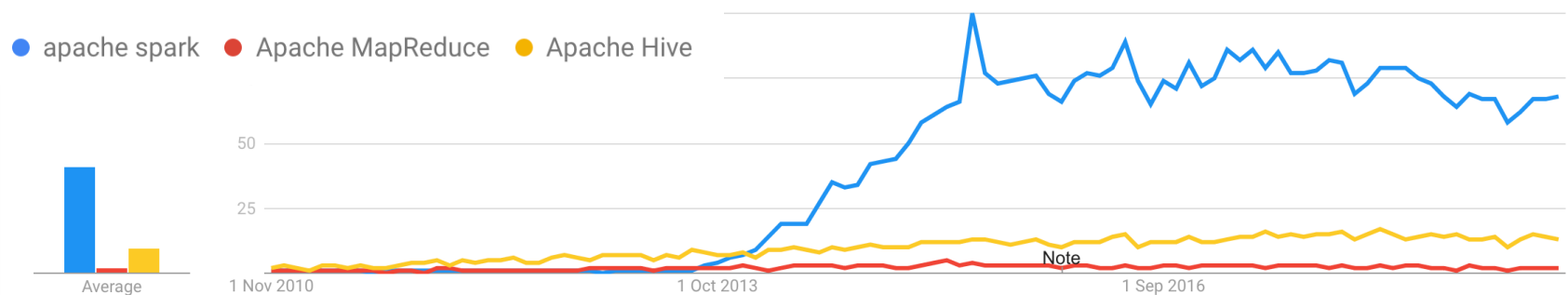


Spark

- Not a modified version of Hadoop
- Separate, fast, MapReduce-like engine
 - In-memory data storage for very fast iterative queries
 - General execution graphs and powerful optimizations
 - Up to 100x faster than Hadoop MapReduce
- Compatible with Hadoop's storage APIs
 - Can run on top of a Hadoop cluster
 - Can run on a cluster in standalone fashion or via Apache Mesos
 - Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, as well as to S3, MapR-FS, Cassandra, etc.

Project History

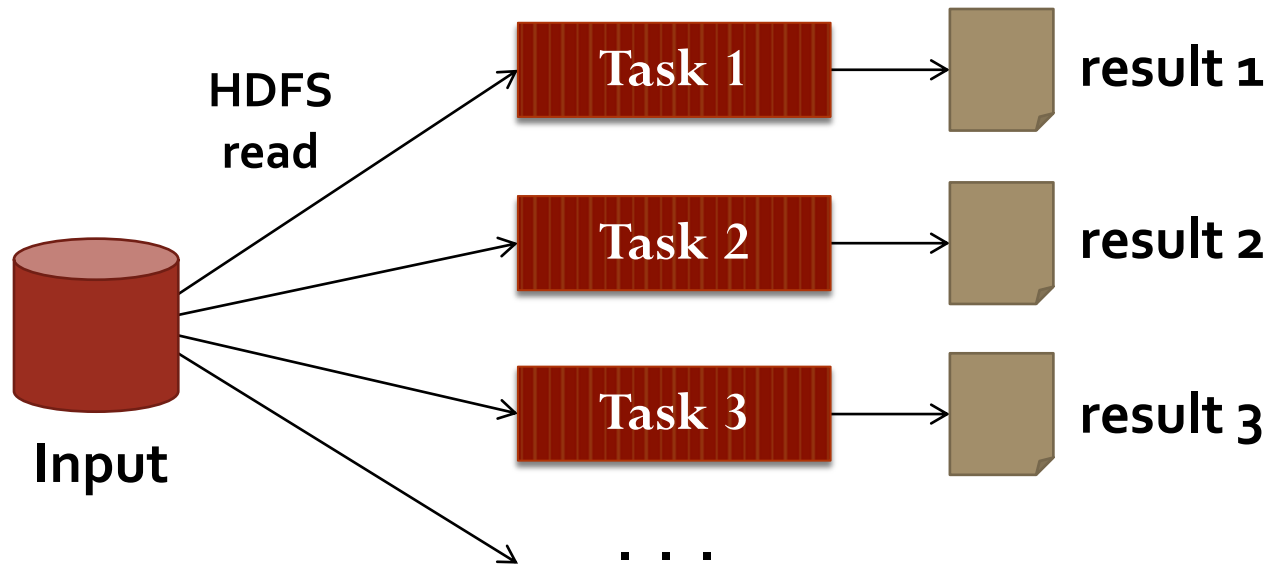
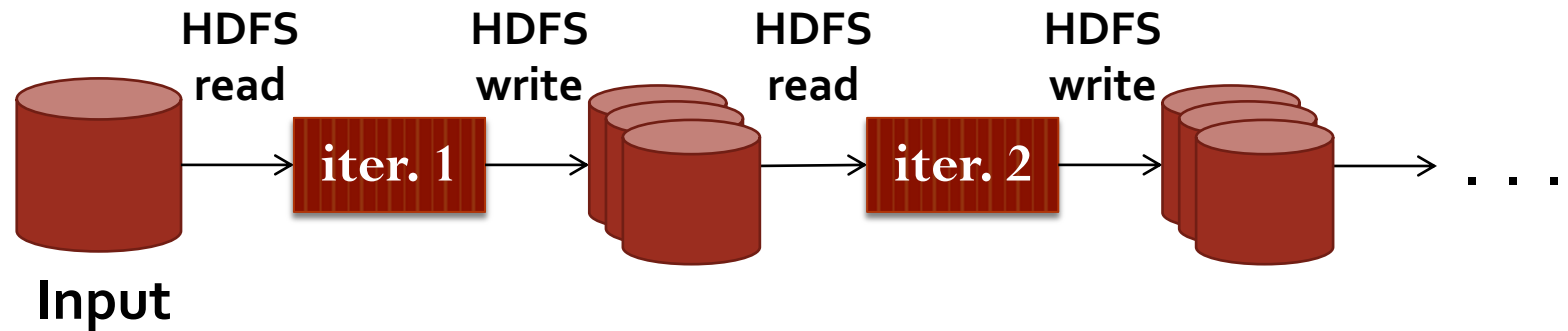
- Spark project started in 2009
- Developed originally at UC Berkeley's AMPLab by Matei Zaharia
- Open sourced 2010, Apache project from 2013
- In 2014, Zaharia founded Databricks
- It is now the most popular project for big data analysis
- Current version: v2.4.5 / Feb. 2020
- Interest over time:



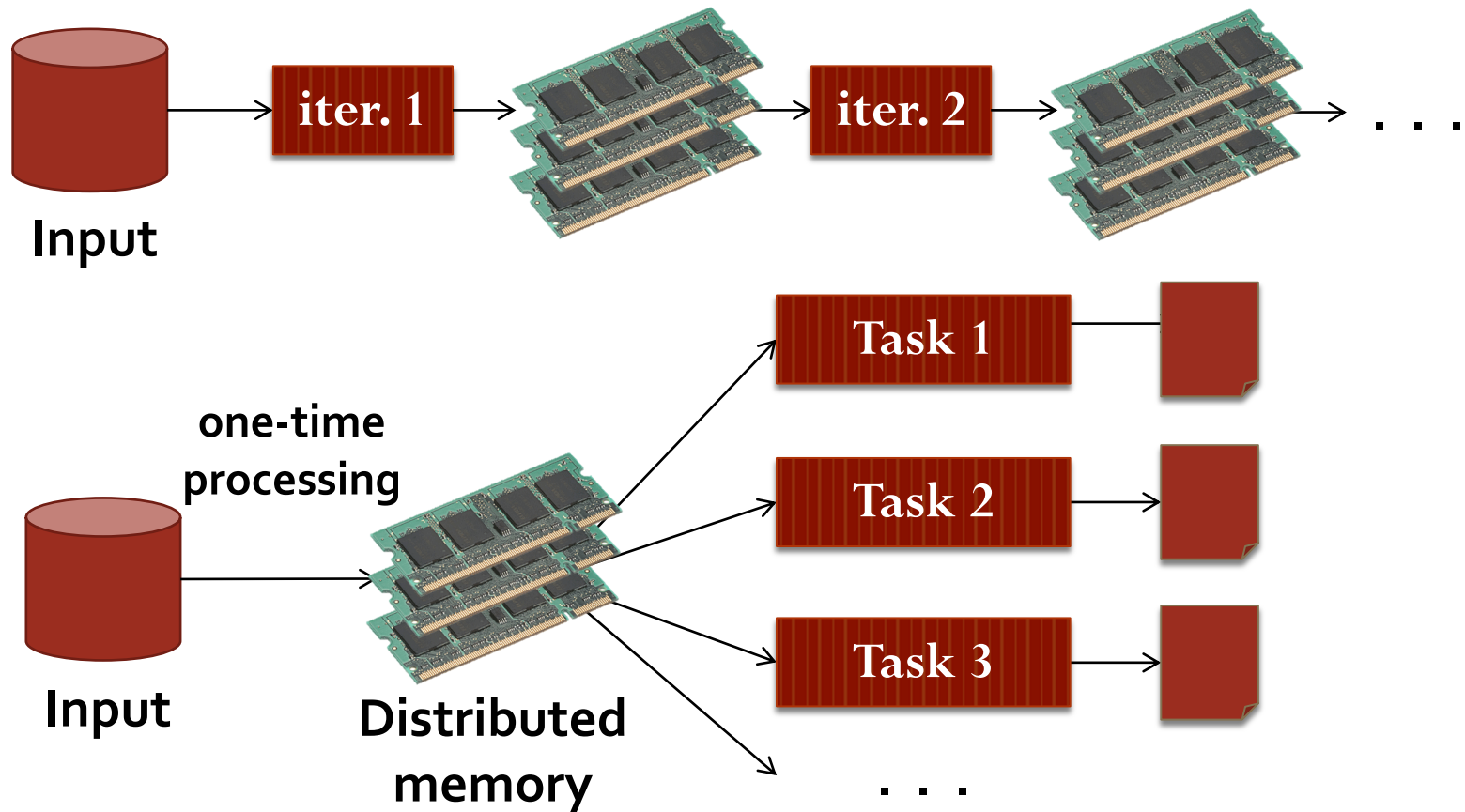
Why a New Programming Model?

- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
 - More complex, multi-stage applications (e.g. iterative graph algorithms and machine learning)
 - More efficiency
 - More interactive ad-hoc queries
- Both multi-stage and interactive apps require faster data sharing across parallel jobs

Data Sharing in MapReduce

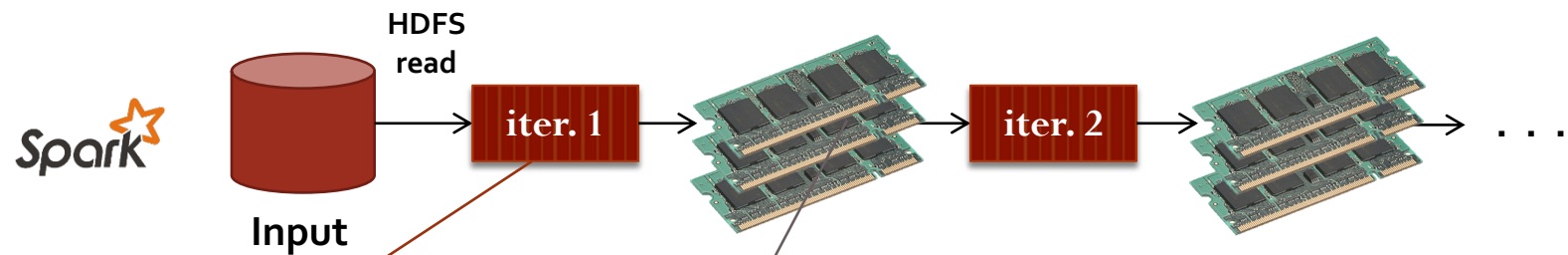


Data Sharing in Spark



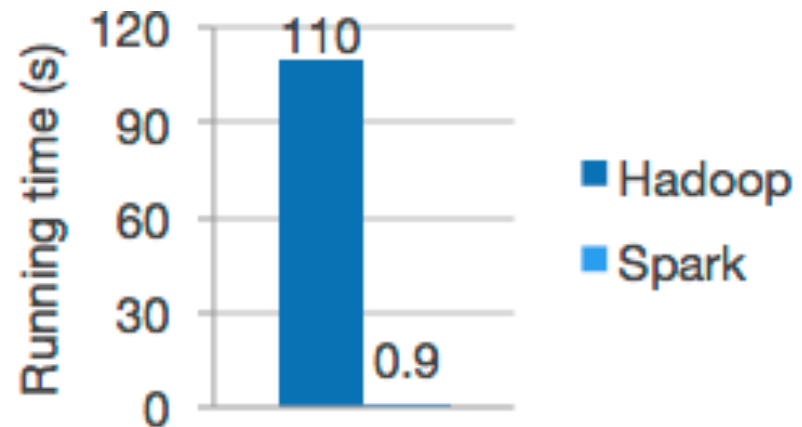
10-100× faster than network and disk

Spark Data Flow



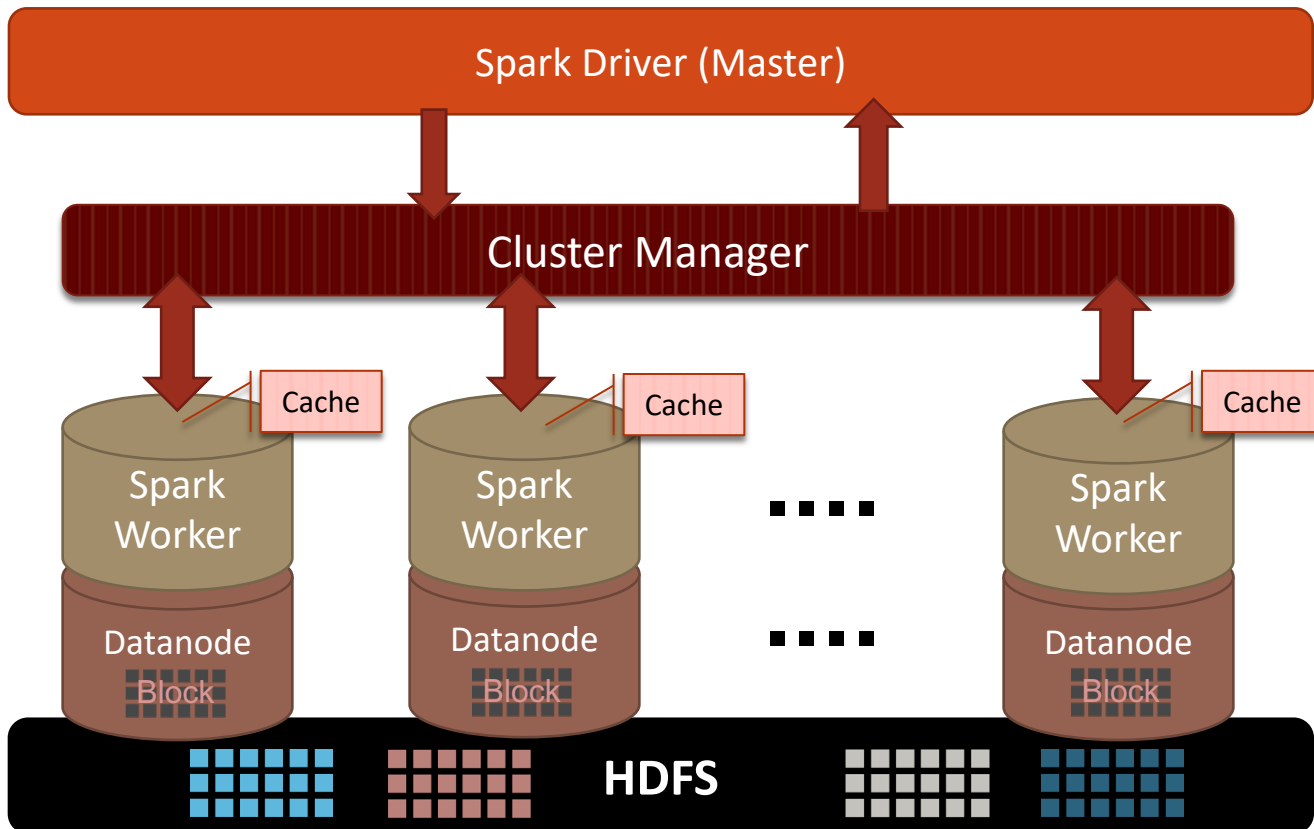
Not tied to 2 stage Map Reduce paradigm

1. Extract a working set
2. Cache it
3. Query it repeatedly

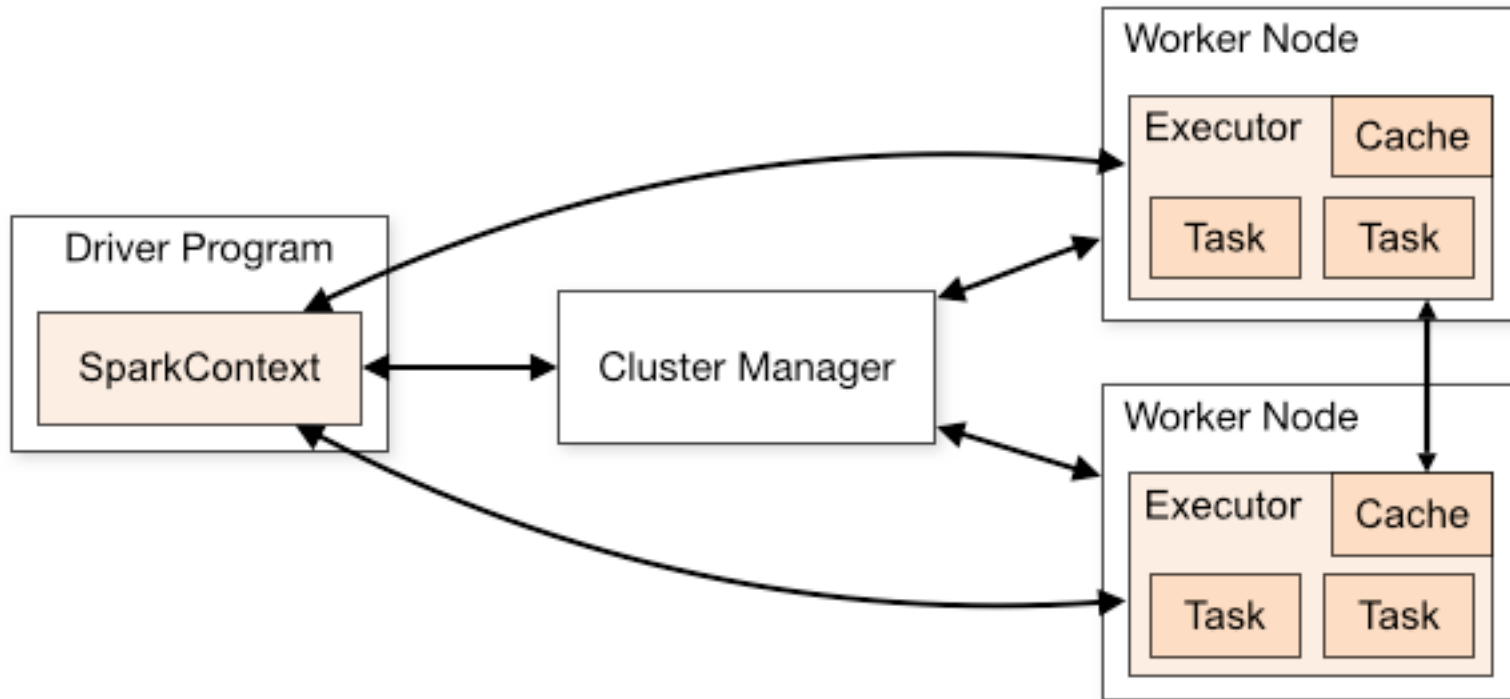


Logistic regression in Hadoop and Spark

Spark architecture



Spark architecture anatomy



Spark architecture anatomy

- Applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (the driver program)
- Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.
- To run on a cluster, the SparkContext can connect to several types of cluster managers (Spark's cluster manager, Mesos or YARN), which allocate resources across applications
- Once connected, Spark:
 - acquires executors on nodes in the cluster, which run computations and store data
 - Next, it sends your application code to the executors.
 - Finally, SparkContext sends *tasks* to the executors to run.

Spark Programming Model

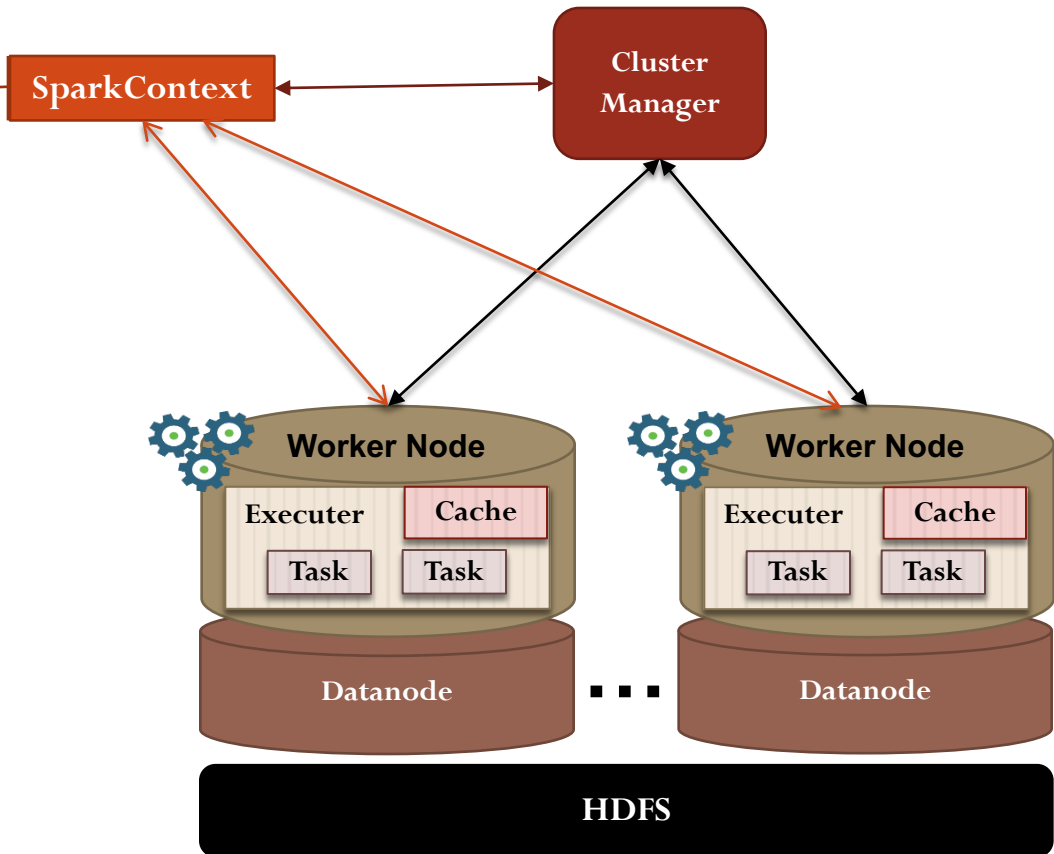
Driver Program

```
sc=new SparkContext  
rDD=sc.textfile("hdfs://...")  
rDD.filter(...)  
rDD.Cache  
rDD.Count  
rDD.map
```

Writes



User
(Developer)



Spark Programming Model

Driver Program

```
sc=new SparkContext  
rDD=sc.textfile("hdfs://...")  
rDD.filter(...)  
rDD.Cache  
rDD.Count  
rDD.map
```

RDD
(Resilient
Distributed
Dataset)

- Immutable Data structure
- In-memory (explicitly)
- Fault Tolerant
- Distributed Data Structure
- Controlled partitioning to optimize data placement
- Can be manipulated using rich set of operators.

Writes



User
(Developer)

Spark Programming Model

- Key idea: **resilient distributed datasets** (RDDs)
 - Distributed collections of objects that can be cached in memory across cluster nodes
 - Manipulated through various parallel operators
 - Automatically rebuilt on failure
 - Can persist in Memory, on Disk, or both
 - Can be partitioned to control parallel processing
- Interface
 - Clean language-integrated API for Scala, Python, Java, and R
 - Can be used interactively from Scala console

Spark's Main Abstraction: RDDs

- Resilient Distributed Datasets or RDD are the distributed memory abstractions that lets programmer perform in-memory parallel computations on large clusters in a highly fault tolerant manner.
- Currently 2 types of RDDs:
 - Parallelized collections: created by executing operators on an existing data collection. Developer can specify the number of slices to cut the dataset into. Ideally 2-3 slices per CPU.
 - Hadoop Datasets: created from any file stored on HDFS or other storage systems supported by Hadoop (S3, Hbase etc). These are created using SparkContext's **textFile** operator. Default number of slices in this case is 1 slice per file block.

Parallelized and distributed datasets

- Parallelized collections are created by calling SparkContext's `parallelize` method on an existing collection in your driver program.

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

- Once created, `distData` can be operated on in parallel.

```
distData.reduce((a, b) => a + b)
```

- Distributed datasets can be created from any storage source supported by Hadoop, (HDFS, HBase, S3, etc.) using SparkContext's **`textFile`** method. This method takes an URI for the file and reads it as a collection of lines.

```
scala> val distFile = sc.textFile("data.txt")
```

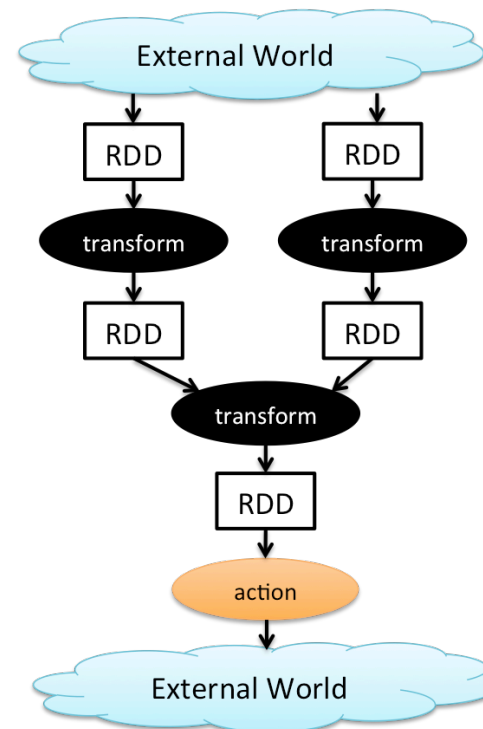
- Once created, `distFile` can be acted on by dataset operations.

```
distFile.map(s => s.length).reduce((a, b) => a + b)
```

Operators over RDDs



General DAG of operators
(e.g. map-reduce-reduce or
even more complex
combinations)



Operators over RDD

- Programmer can perform 3 types of operations

Transformations

- Create a new dataset from an existing one.
- Lazy in nature. They are executed only when some action is performed.
- Example :
 - Map(func)
 - Filter(func)
 - Distinct()

Actions

- Returns to the driver program a value or exports data to a storage system after performing a computation.
- Example:
 - Count()
 - Reduce(func)
 - Collect
 - Take()

Persistence

- For caching datasets in-memory for future operations.
- Option to store on disk or RAM or mixed (Storage Level).
- Example:
 - Persist()
 - Cache()

Transformations

- Transformations operations on a RDD that return RDD objects or collections of RDD
 - e.g. map, filter, join, cogroup, etc.
- Transformations are lazy and are not executed immediately, but only when an action requires a result to be returned to the driver program.
- This design enables Spark to run more efficiently.
 - For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.

Two kinds of transformations

- Narrow transformations
 - They are the result of map, filter and such and operate over data from a single partition only (i.e. they are self-sustained).
 - An output RDD has partitions originating from a single partition of the parent RDD.
 - Spark groups narrow transformations as a stage.
- Wide transformations
 - They are the result of groupByKey and reduceByKey. The data required to compute the records in a single partition may reside in many partitions of the parent RDD.
 - All the tuples with the same key must end up in the same partition, processed by the same task.
 - Spark must execute RDD shuffle, which transfers data across cluster and results in a new stage with a new set of partitions.

Transformations (1)

Transformation	Meaning
<code>map(func)</code>	Return a new distributed dataset formed by passing each element of the source through a function <i>func</i> .
<code>filter(func)</code>	Return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true.
<code>flatMap(func)</code>	Similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item).
<code>mapPartitions(func)</code>	Similar to map, but runs separately on each partition of the RDD, so <i>func</i> must be of type <code>Iterator<T> => Iterator<U></code> when running on an RDD of type T.
<code>mapPartitionsWithIndex(func)</code>	Similar to mapPartitions, but also provides <i>func</i> with an integer value representing the index of the partition, so <i>func</i> must be of type <code>(Int, Iterator<T>) => Iterator<U></code> when running on an RDD of type T.
<code>sample(withReplacement, fraction, seed)</code>	Sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator seed.
<code>union(otherDataset)</code>	Return a new dataset that contains the union of the elements in the source dataset and the argument.
<code>intersection(otherDataset)</code>	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
<code>distinct([numTasks])</code>	Return a new dataset that contains the distinct elements of the source dataset.
<code>groupByKey([numTasks])</code>	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using <code>reduceByKey</code> or <code>aggregateByKey</code> will yield much better performance. Note: By default, the number of reduce task depends on the number of partitions of the parent RDD. You can pass an optional <code>numTasks</code> argument to set a different number of tasks.
<code>reduceByKey(func, [numTasks])</code>	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 <i>func</i> , which must be of type <code>(V,V) => V</code> . Like in <code>groupByKey</code> , the number of reduce tasks is configurable through an optional second argument.

Transformations (2)

Transformation	Meaning
<code>aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])</code>	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in <code>groupByKey</code> , the number of reduce tasks is configurable through an optional second argument.
<code>sortByKey([ascending], [numTasks])</code>	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 <code>ascending</code> argument.
<code>join(otherDataset, [numTasks])</code>	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. Outer joins are supported through <code>leftOuterJoin</code> , <code>rightOuterJoin</code> , and <code>fullOuterJoin</code> .
<code>cogroup(otherDataset, [numTasks])</code>	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called <code>groupWith</code> .
<code>cartesian(otherDataset)</code>	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
<code>pipe(command, [envVars])</code>	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
<code>coalesce(numPartitions)</code>	Decrease the number of partitions in the RDD to <code>numPartitions</code> . Useful for running operations more efficiently after filtering down a large dataset.
<code>repartition(numPartitions)</code>	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
<code>repartitionAndSortWithinPartitions(partitioner)</code>	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling <code>repartition</code> and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Actions

- Actions are operations that return values, i.e. any RDD operation that returns a value of any type but an RDD is an action
 - e.g., Reduce, Count, Collect, Take, SaveAs, ...
- Actions are synchronous. They trigger execution of RDD transformations to return values.
- Until no action is fired, the data to be processed is not even accessed
- Only actions can materialize the entire process with real data.
- Cause data to be returned to driver or saved to output
- Cause data retrieval and execution of all transformations on RDDs

Actions

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should 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. This is 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.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
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) (Java and Scala)	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. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(path) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

Persistence

- One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations.
- By default, each RDD is recomputed each time you run an action on it, unless you persist the RDD.
- When you persist an RDD *x*, each node stores any partitions of *x* that it computes in memory and reuses them in other actions on *x*. This allows future actions to be much faster.
- You can mark an RDD to be persisted using the **`persist()`** or **`cache()`** methods. The first time it is computed in an action, it will be kept in memory on the nodes.
- Spark's cache is fault-tolerant — if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

Storage level

- Using **persist()** one can specify the Storage Level for persisting an RDD.
- **Cache()** is just a short hand for default storage level, which is **MEMORY_ONLY**.
- StorageLevel for **persist(*)**:
 - **MEMORY_ONLY**
 - **MEMORY_AND_DISK**
 - **MEMORY_ONLY_SER, MEMORY_AND_DISK_SER**
 - **DISK_ONLY**
 - **MEMORY_ONLY_2, MEMORY_AND_DISK_2**, etc.
- Which Storage level is best:
 - Few things to consider:
 - Try to keep in-memory as much as possible
 - Serialization make the objects much more space-efficient
 - Try not to spill to disk unless the functions that computed your datasets are expensive
 - Use replication only if you want fault tolerance

How Spark works at runtime

- User application create RDDs, transform them, and run actions.
- This results in a DAG (Directed Acyclic Graph) of operators.
- DAG is compiled into stages
- Each stage is executed as a series of Task (one Task for each Partition)
- Actions drive the execution

Example in Scala

```
val textFile = sc.textFile("hdfs://...")
```

```
val counts = textFile.flatMap(line => line.split(" "))  
                      .map(word => (word, 1))  
                      .reduceByKey(_ + _)
```

```
counts.saveAsTextFile("hdfs://...")
```

Same example in Python

```
text_file = sc.textFile("hdfs://...")
```

```
counts = text_file.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)
```

```
output = counts.collect()
```

```
output.saveAsTextFile("hdfs://...")
```


Same example in Java

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
```

```
JavaRDD<String> words = textFile.flatMap(new FlatMapFunction<String, String>() {  
    public Iterable<String> call(String s) { return Arrays.asList(s.split(" ")); }  
});
```

```
JavaPairRDD<String, Integer> pairs =  
    words.mapToPair(new PairFunction<String, String, Integer>() {  
        public Tuple2<String, Integer> call(String s) {  
            return new Tuple2<String, Integer>(s, 1); }  
    });
```

```
JavaPairRDD<String, Integer> counts =  
    pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {  
        public Integer call(Integer a, Integer b) { return a + b; }  
    });  
counts.saveAsTextFile("hdfs://...");
```

Complete example in Java (1)

```
package org.apache.spark.examples;

import scala.Tuple2;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.regex.Pattern;
```

Complete example in Java (2)

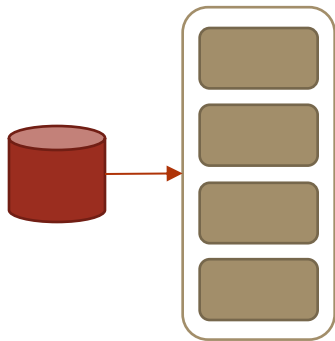
```
public final class JavaWordCount {  
    private static final Pattern SPACE = Pattern.compile(" ");  
    public static void main(String[] args) throws Exception {  
        if (args.length < 1) {  
            System.err.println("Usage: JavaWordCount <file>");  
            System.exit(1);  
        }  
        SparkConf sparkConf = new SparkConf().setAppName("JavaWordCount");  
        JavaSparkContext ctx = new JavaSparkContext(sparkConf);  
        JavaRDD<String> lines = ctx.textFile(args[0], 1);  
        JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {  
            @Override  
            public Iterator<String> call(String s) {  
                return Arrays.asList(SPACE.split(s)).iterator();  
            }  
        });  
    }  
}
```

Complete example in Java (3)

```
JavaPairRDD<String, Integer> ones = words.mapToPair(  
    new PairFunction<String, String, Integer>() {  
        @Override  
        public Tuple2<String, Integer> call(String s) {  
            return new Tuple2<>(s, 1);  
        }  
    });  
JavaPairRDD<String, Integer> counts = ones.reduceByKey(  
    new Function2<Integer, Integer, Integer>() {  
        @Override  
        public Integer call(Integer i1, Integer i2) {  
            return i1 + i2;  
        }  
    });  
List<Tuple2<String, Integer>> output = counts.collect();  
for (Tuple2<?,?> tuple : output) {  
    System.out.println(tuple._1() + ": " + tuple._2());  
}  
ctx.stop();  
}
```

Example

```
text_file = sc.textFile("hdfs://...")
```

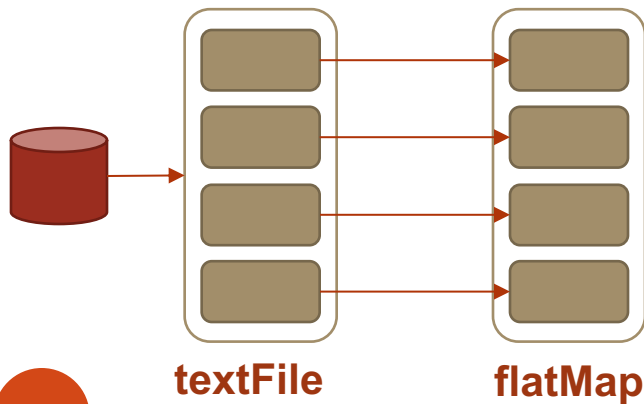


textFile

Example

```
text_file = sc.textFile("hdfs://...")
```

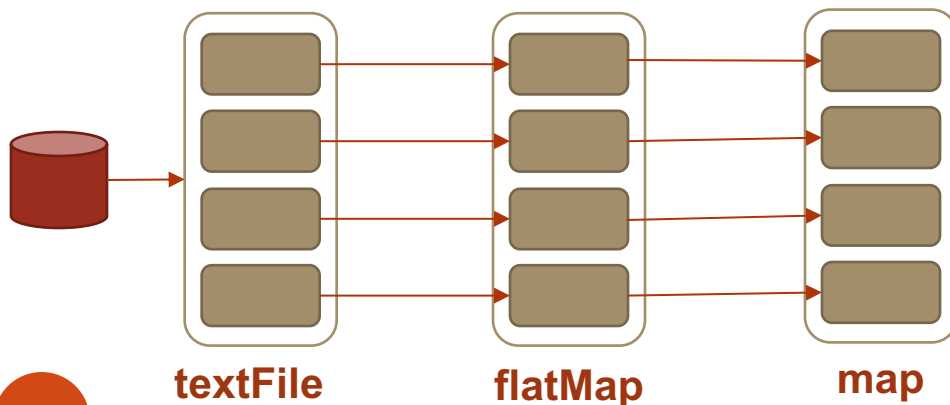
```
counts = text_file.flatMap(lambda line: line.split(" ")) \
```



Example

```
text_file = sc.textFile("hdfs://...")
```

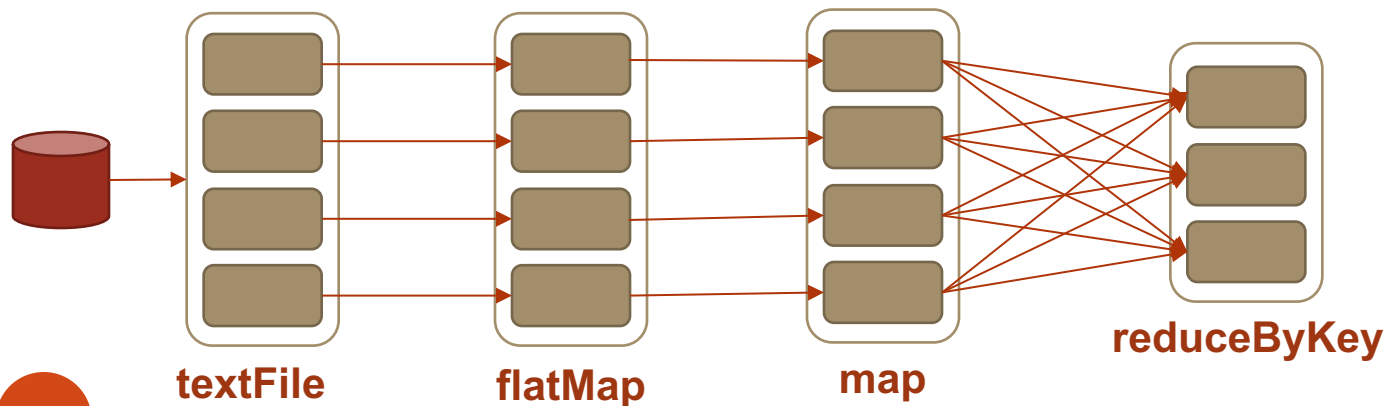
```
counts = text_file.flatMap(lambda line: line.split(" ")) \  
                  .map(lambda word: (word, 1)) \
```



Example

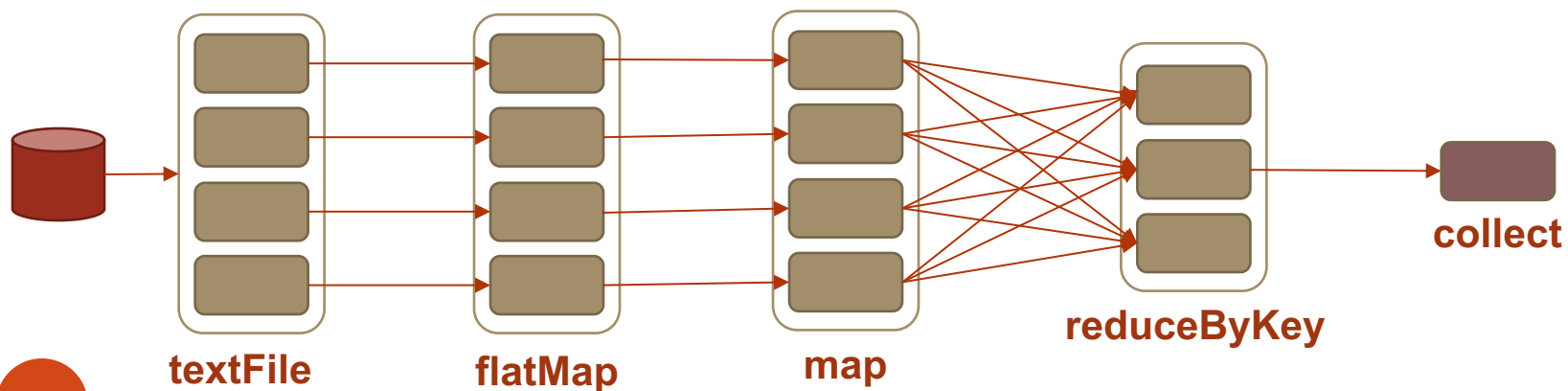
```
text_file = sc.textFile("hdfs://...")
```

```
counts = text_file.flatMap(lambda line: line.split(" ")) \  
    .map(lambda word: (word, 1)) \  
    .reduceByKey(lambda a, b: a + b)
```



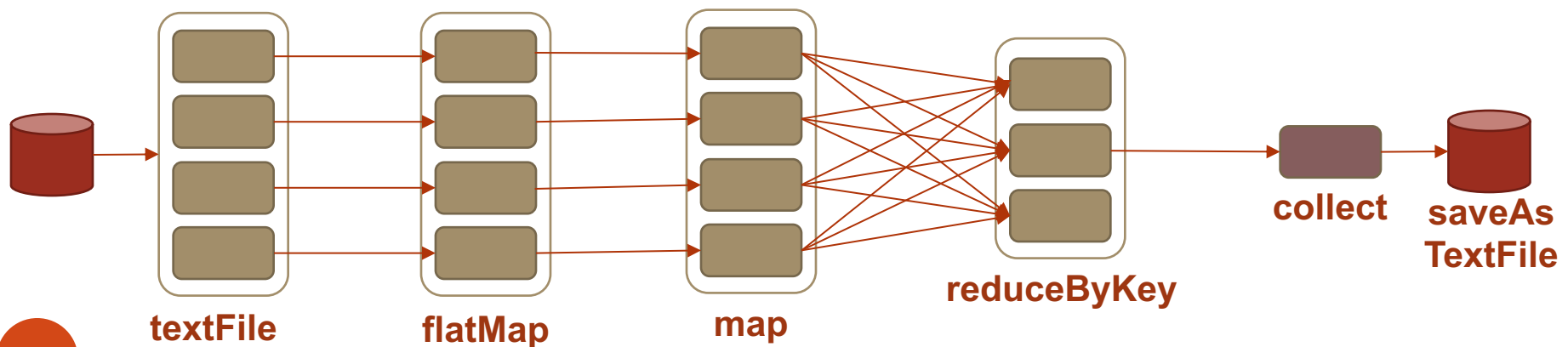
Example

```
text_file = sc.textFile("hdfs://...")  
  
counts = text_file.flatMap(lambda line: line.split(" ")) \  
    .map(lambda word: (word, 1)) \  
    .reduceByKey(lambda a, b: a + b)  
  
output = counts.collect()
```

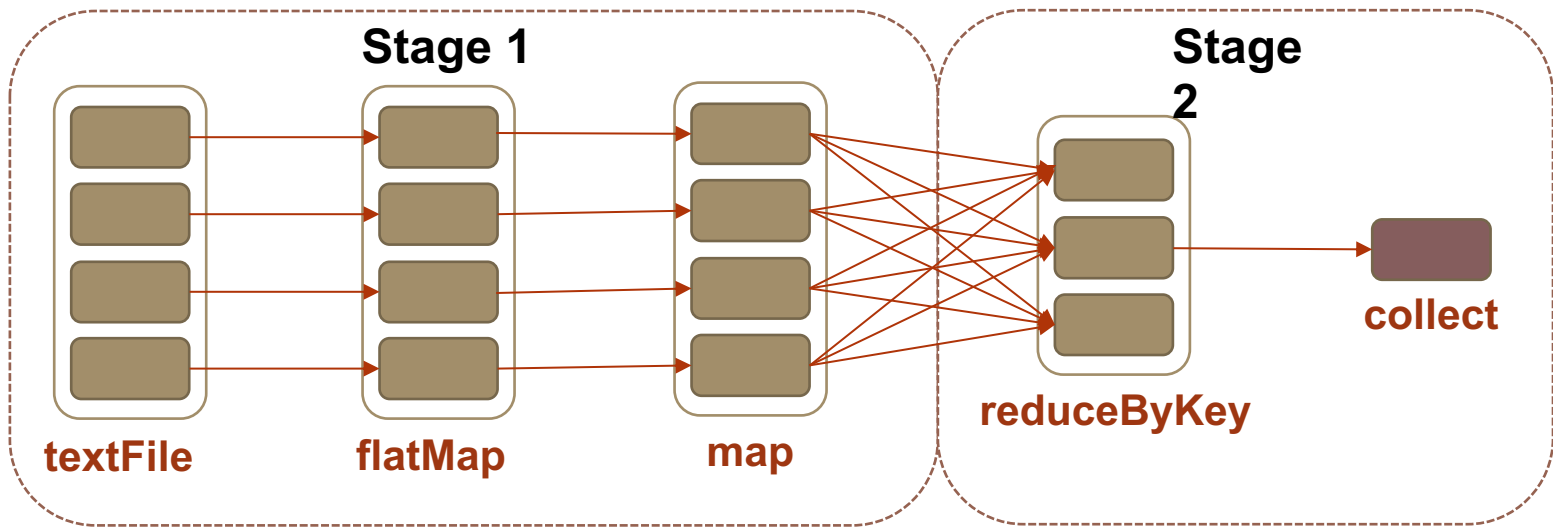


Example

```
text_file = sc.textFile("hdfs://...")  
  
counts = text_file.flatMap(lambda line: line.split(" ")) \  
                  .map(lambda word: (word, 1)) \  
                  .reduceByKey(lambda a, b: a + b)  
  
output = counts.collect()  
  
output.saveAsTextFile("hdfs://...")
```



Execution Plan



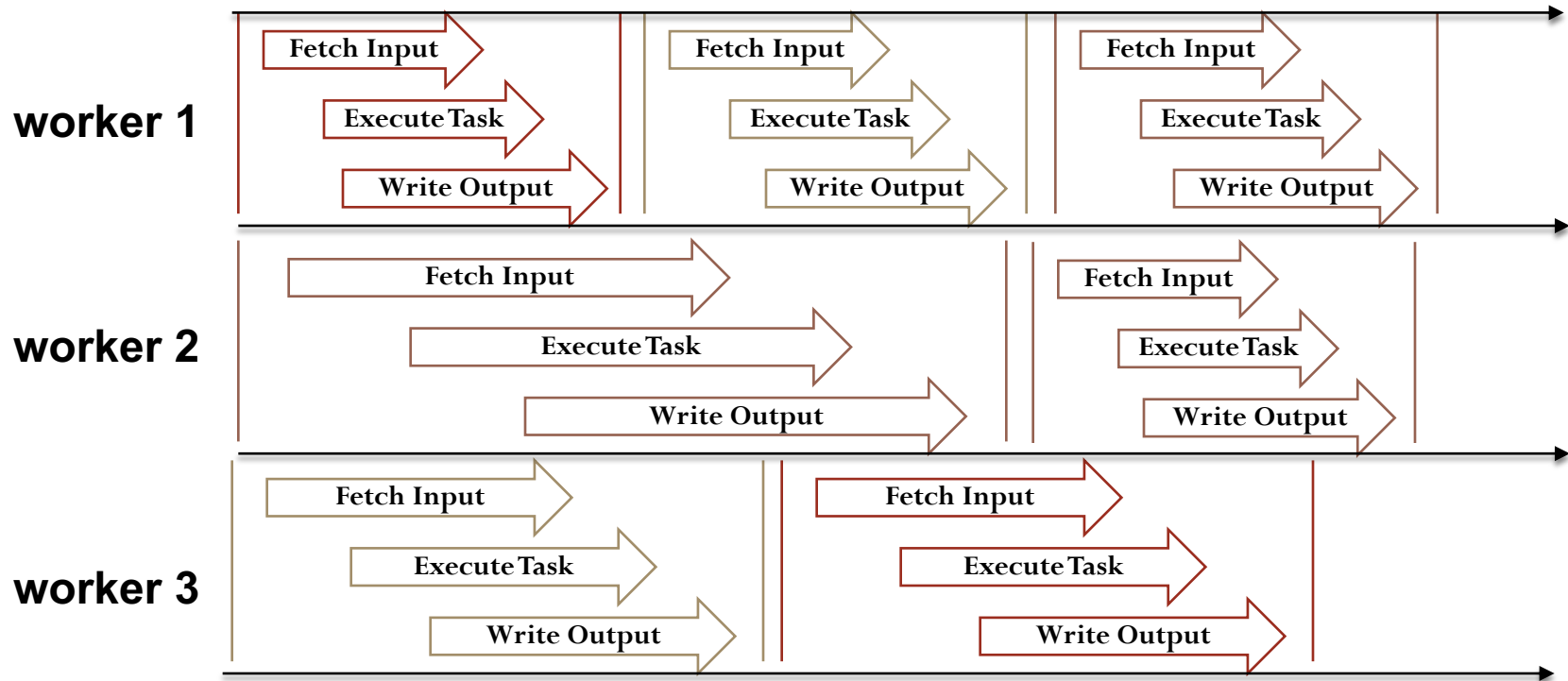
Stages are sequences of RDDs, that don't have a Shuffle in between

Stage Execution




- Spark:
 - Creates a task for each Partition in the new RDD
 - Schedules and assign tasks to slaves
- All this happens internally (you need to do anything)

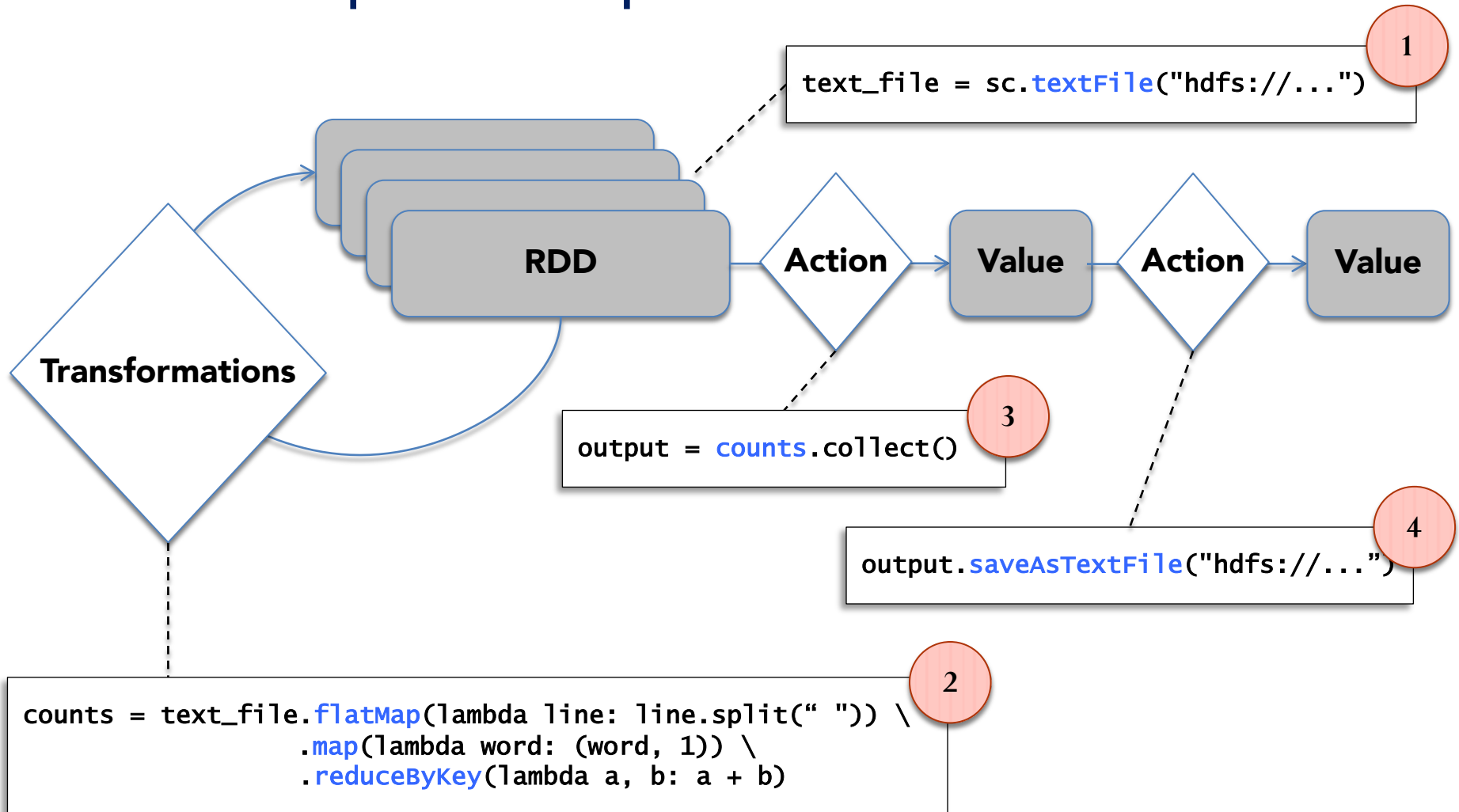
Spark Executor (Slaves)



Summary of Components

- 
- Task : The fundamental unit of execution in Spark
 - Stage: Set of Tasks that run in parallel
 - DAG : Logical Graph of RDD operations
 - RDD : Parallel dataset with partitions

Conceptual Representation



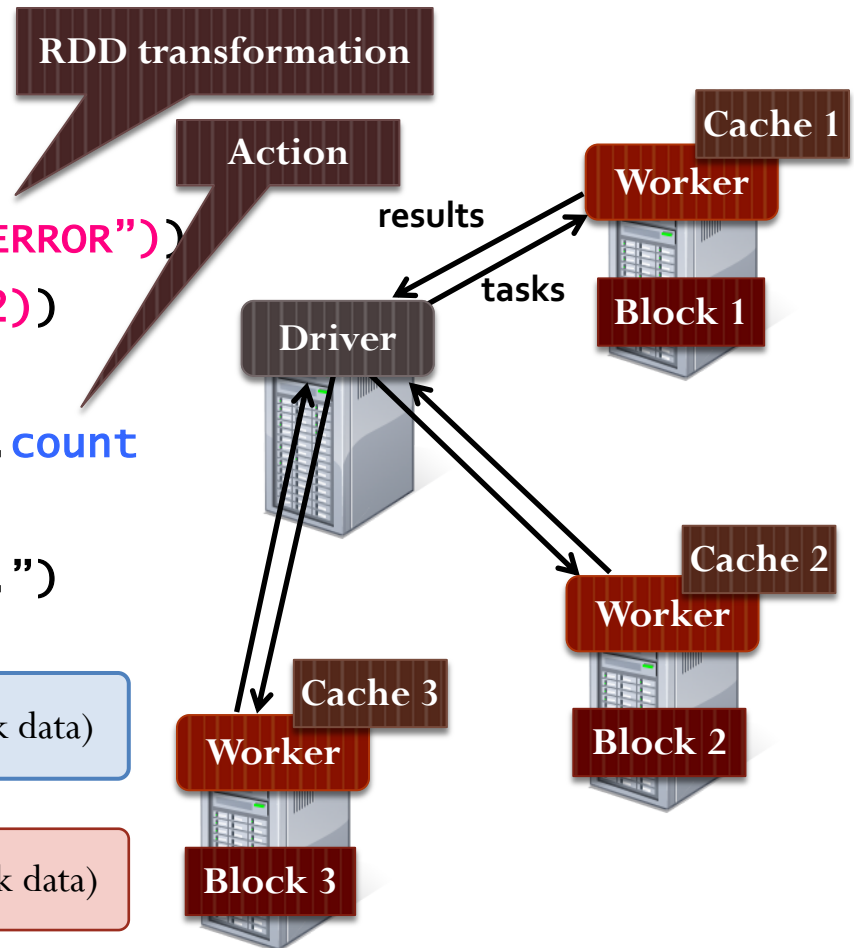
Runtime execution (Log Mining)

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
...
cachedMsgs.saveAsTextFile("hdfs://...")
```

Full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

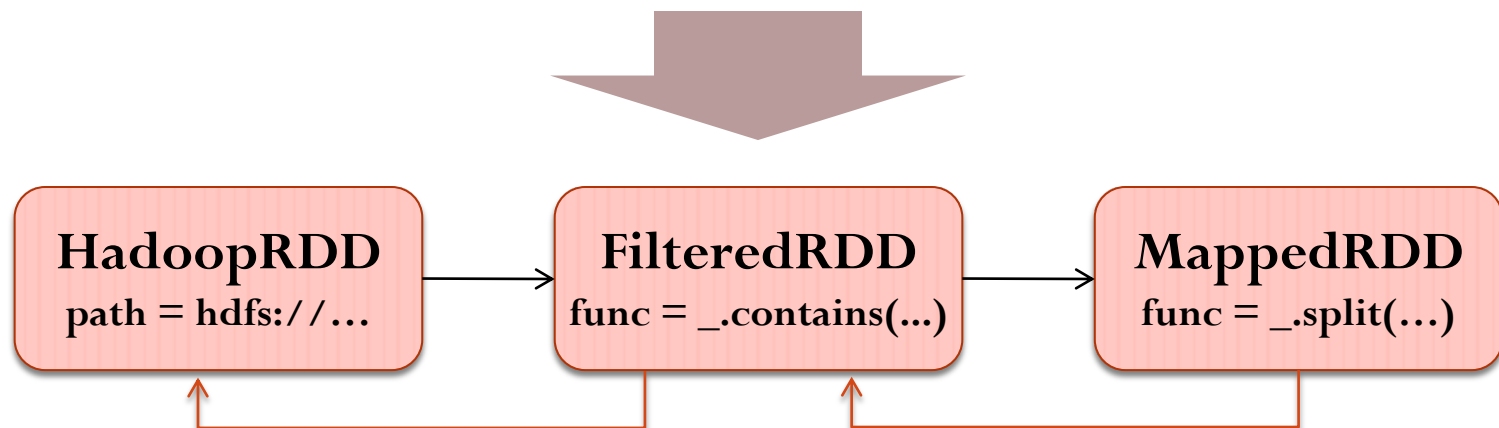
Scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)



Fault Tolerance

- RDDs track the series of transformations used to build them (their lineage)
- Lineage information is used to recompute lost data
- E.g:

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



Another example: Logistic Regression

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

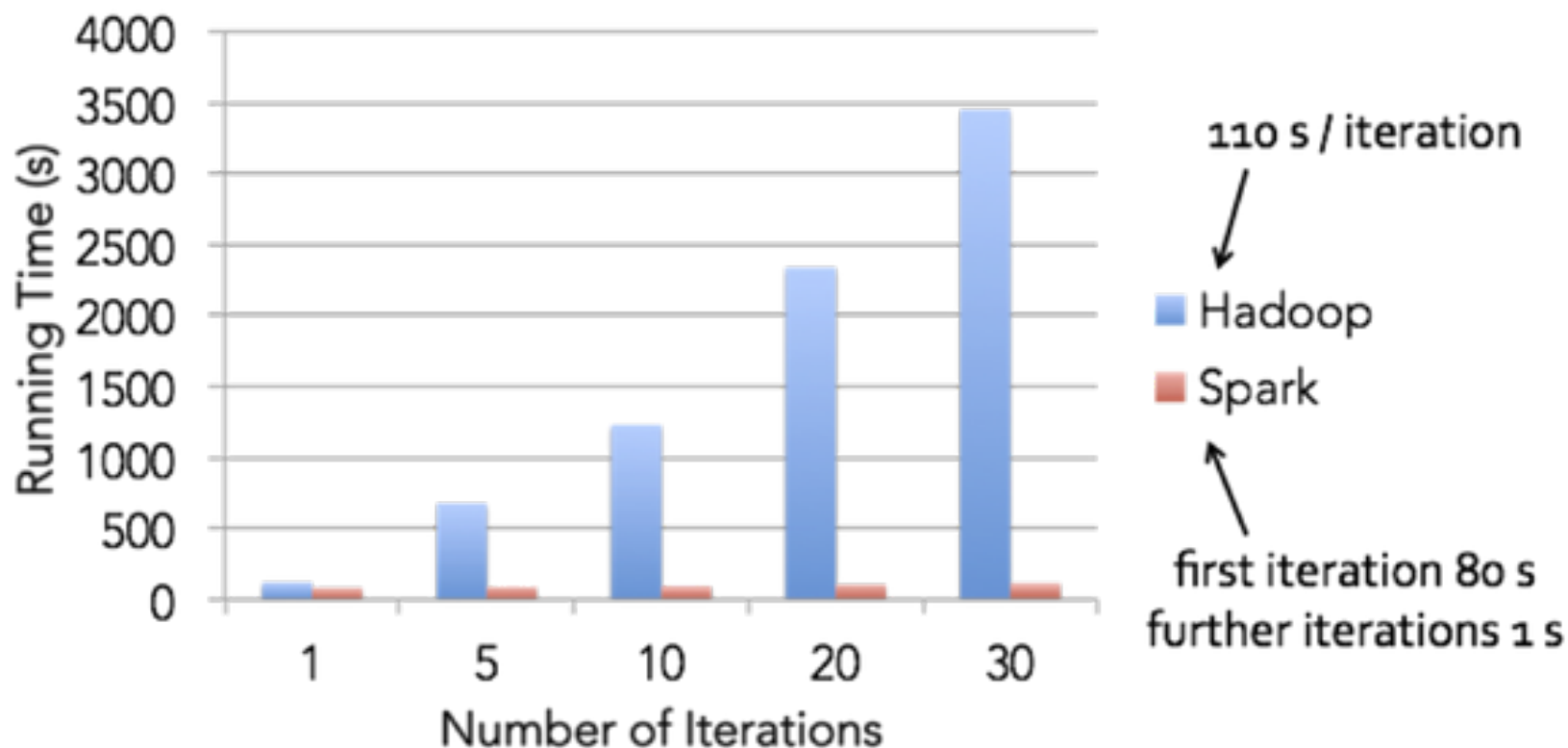
println("Final w: " + w)
```

Load data in memory once

Initial parameter vector

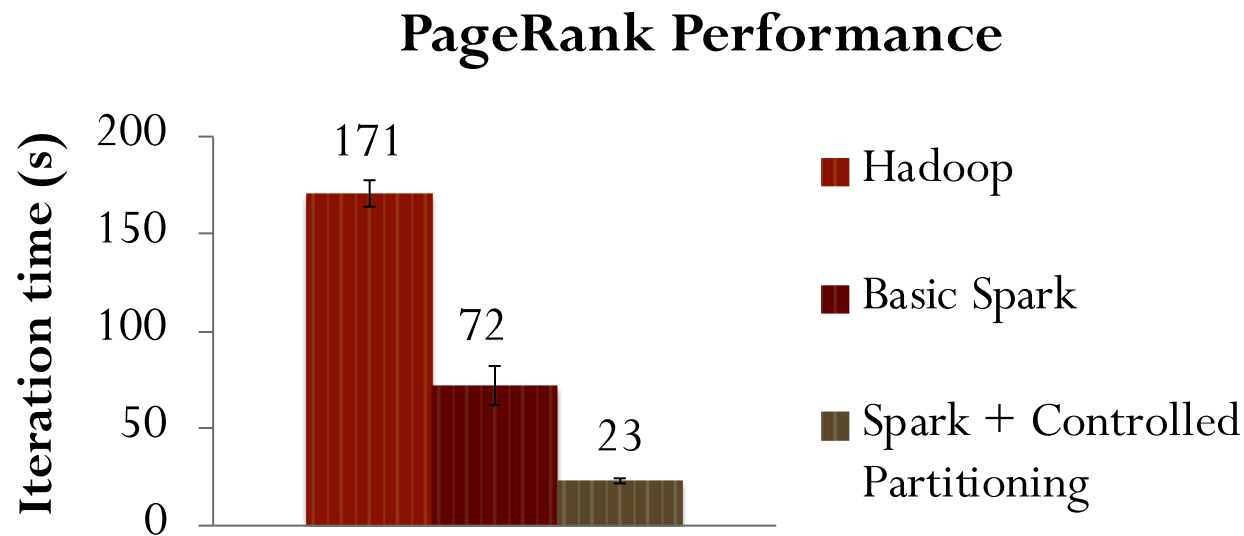
Repeated MapReduce steps
to do gradient descent

Logistic Regression Performance



Other Spark Features

- Hash-based reduces (faster than MapReduce sort)
- Controlled data partitioning to lower communication



User Applications

CONVIVA®

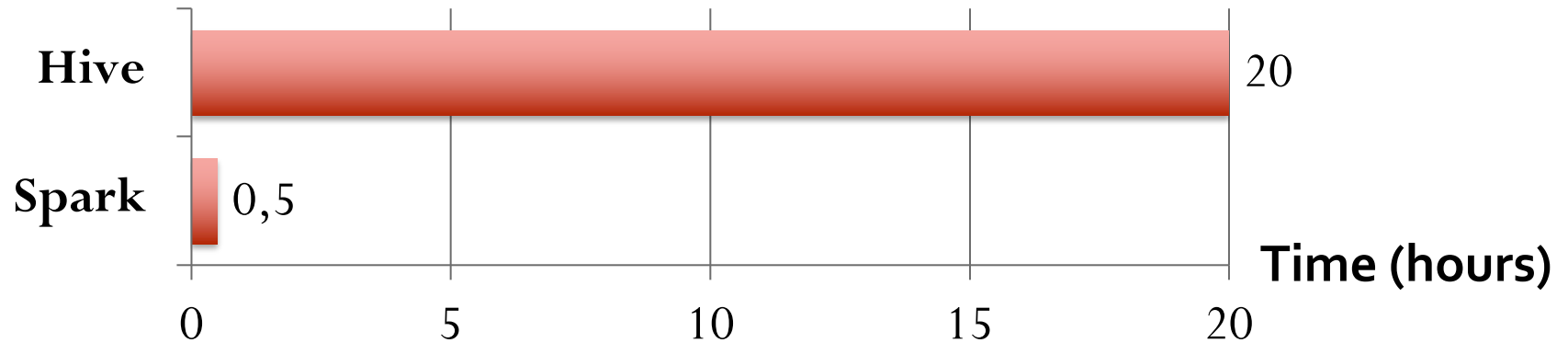
- In-memory analytics & anomaly detection (Conviva)
- Interactive queries on data streams (Quantifind)
- Exploratory log analysis (Foursquare)
- Traffic estimation w/ GPS data (Mobile Millennium)
- Twitter spam classification (Monarch)
- . . .

quantifind

foursquare

KLOUT

Conviva GeoReport



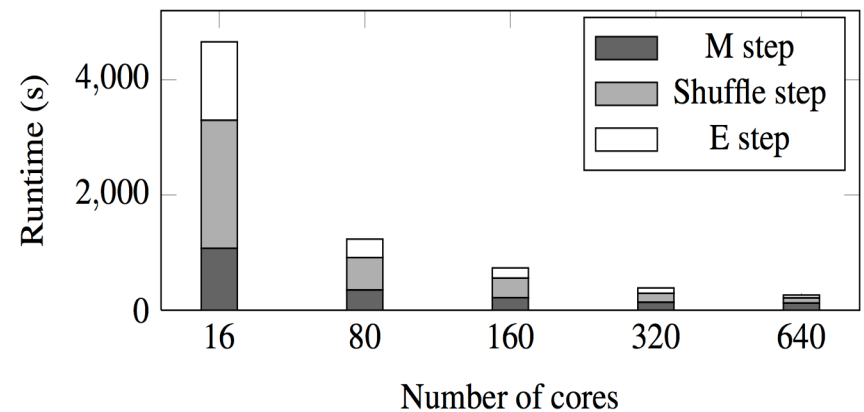
- Group aggregations on many keys w/ same filter
- 40× gain over Hive from avoiding repeated reading, deserialization and filtering

Mobile Millennium Project

- Estimate city traffic from crowdsourced GPS data



**Iterative EM algorithm
scaling to 160 nodes**



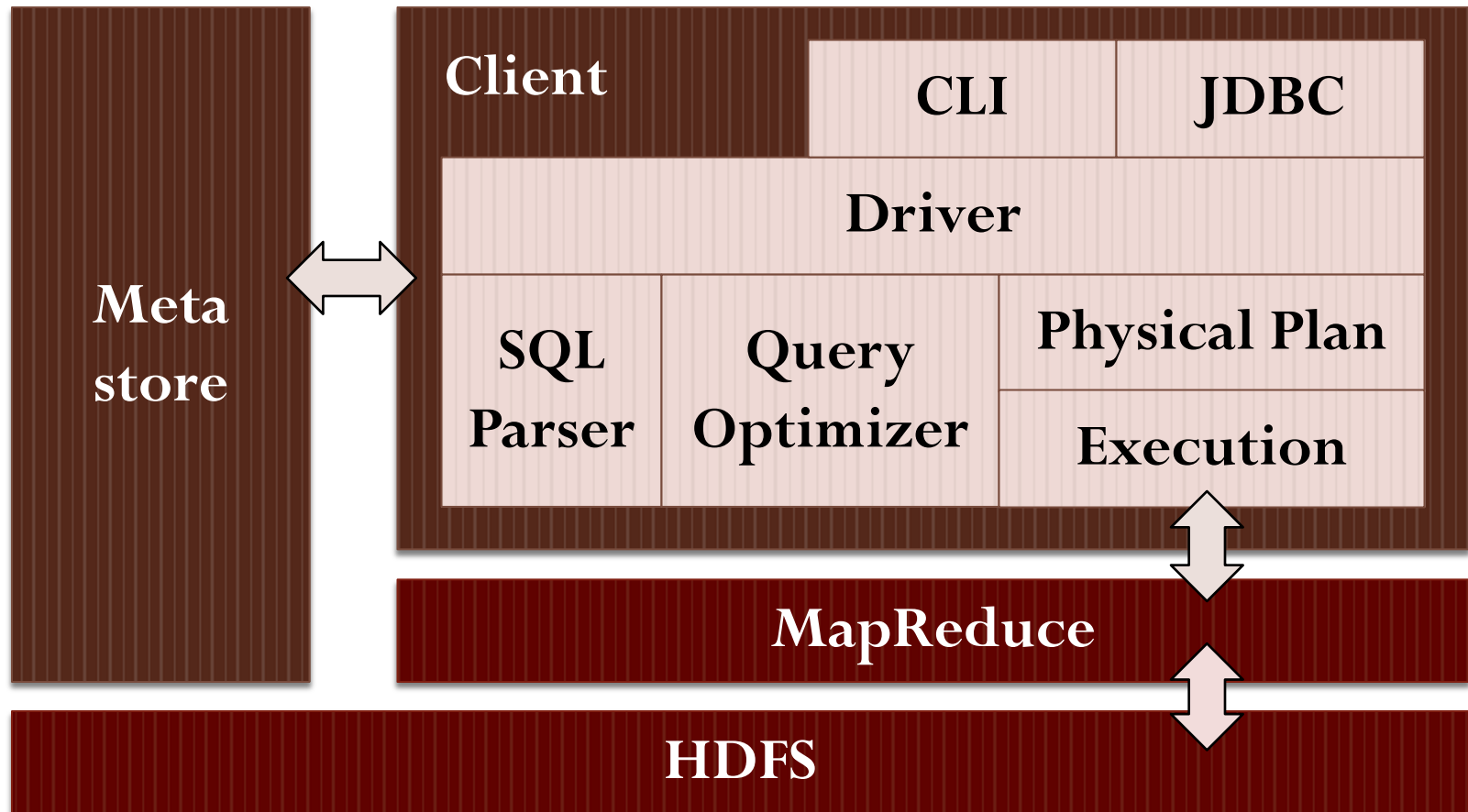
What is Spark SQL?



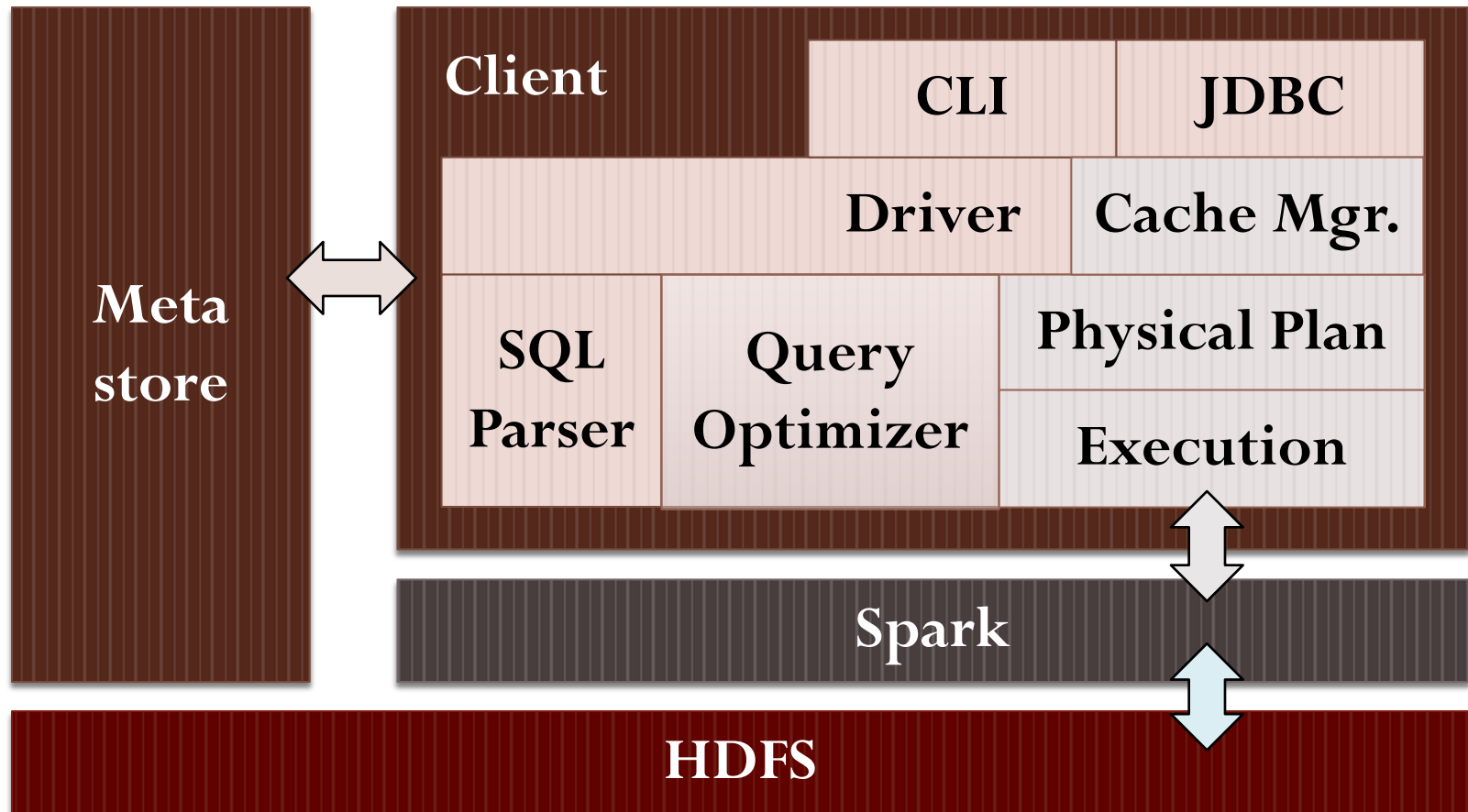
- Spark module for structured data processing
- Port of Apache Hive to run on Spark
- Compatible with existing Hive data (you can run unmodified Hive queries on existing data)
- Speedup of up to 40x
- Motivation:
 - Hive is great, but Hadoop's execution engine makes even the smallest queries take minutes
 - Many data users know SQL
 - Can we extend Hive to run on Spark?
- Initially “Shark”, now “Spark SQL”



Hive Architecture



Spark SQL Architecture



Efficient In-Memory Storage

- Simply caching records as Java objects is inefficient due to high per-object overhead
- Instead, Spark SQL employs column-oriented storage using arrays of primitive types.
- This format is called **Parquet**

**Benefit: similarly compact size to serialized data,
but >5x faster to access**

Row Storage

1	john	4.1
2	mike	3.5
3	sally	6.4

Column Storage

1	2	3
john	mike	sally
4.1	3.5	6.4

Datasets and DataFrames

- A (RD)Dataset is a distributed collection of data, usually organized in records.
 - A new interface added in Spark 1.6.
 - The Dataset API is available in Scala and Java. Python does not but many of the benefits of the Dataset API are already available.
- A DataFrame is a Dataset organized into named columns.
 - It is conceptually equivalent to a table in a relational database.
 - DataFrames can be constructed from: structured data files, tables in Hive, external databases, or existing RDDs.
 - The DataFrame API is available in Scala, Java, Python, and R.
 - In Scala and Java, a DataFrame is represented by a Dataset of Rows.

Starting Point: SparkSession

- The entry point into all functionality in Spark SQL is the `SparkSession` class.
- To create a basic `SparkSession`, just use `SparkSession.builder()`

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Creating DataFrames

- Within a `SparkSession`, applications can create `DataFrames` from an existing `RDD`, from a Hive table, or from Spark data sources (json, parquet, csv, text, jdbc, orc, libsvm).
- Spark SQL can automatically infer the schema of a JSON dataset and load it as a `DataFrame`.

```
df=spark.read.json("examples/src/main/Resources/people.json")
df.show()
```

```
# +-----+-----+
# |  age|    name|
# +-----+-----+
# |null|Michael|
# |  30|    Andy|
# |  19|  Justin|
# +-----+-----+
```

DataFrame Operations

DataFrames provide a domain-specific language for structured data manipulation in Scala, Java, Python and R.

```
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)
```

```
df.select("name").show()
# +-----+
# |    name|
# +-----+
# |Michael|
# |    Andy|
# +-----+
```

```
df.filter(df['age']>21).show()
# +---+-----+
# |age|name|
# +---+-----+
# | 30|Andy|
# +---+-----+
```

```
df.groupBy("age").count().show()
# +---+-----+
# | age|count|
# +---+-----+
# |  19|    1|
# |  30|    1|
# +---+-----+
```

Running SQL Queries Programmatically

The `sql` function on a `SparkSession` enables applications to run SQL queries programmatically and returns the result as a `DataFrame`.

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")
sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
# +-----+-----+
# |  age|   name|
# +-----+-----+
# |null|Michael|
# |  30|   Andy|
# |  19|  Justin|
# +-----+-----+
```


Load/Save Functions

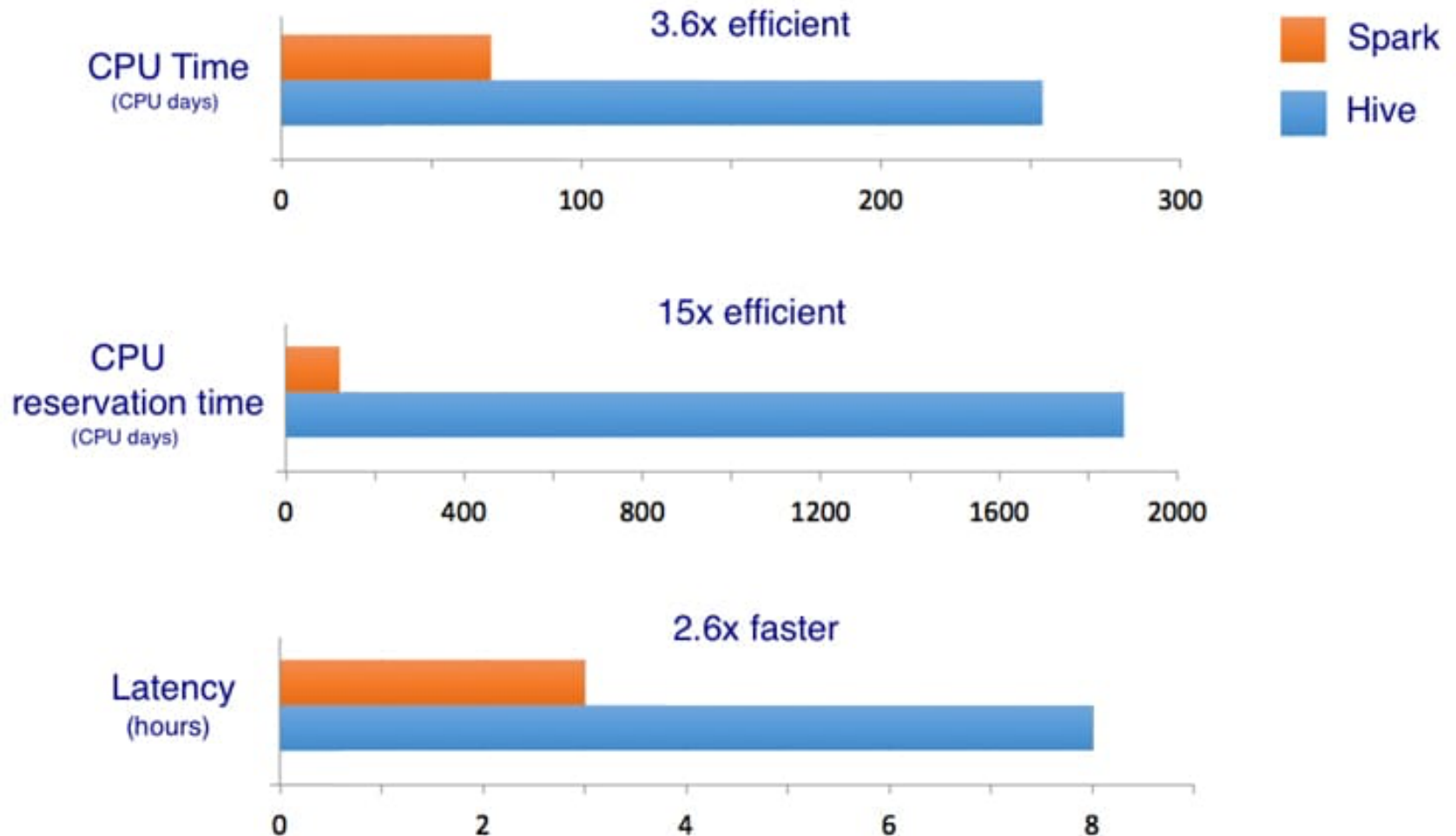
- In the simplest form, the default data source (parquet unless otherwise configured by `spark.sql.sources.default`) will be used for all operations.

```
df=spark.read.load("examples/src/main/resources/
                    users.parquet")
df.select("name","age").write.save("out.parquet")
```

- You can also manually specify the data source (json, parquet, csv, text , jdbc, orc, libsvm). DataFrames loaded from any data source type can be converted into other types using this syntax.

```
df=spark.read.load("examples/src/main/resources/people.json",
                    format="json")
df.select("name","age").write.save("out.parquet",
                                    format="parquet")
```

Spark vs Hive



What's Next?

- Spark's model was motivated by two emerging uses (interactive and multi-stage applications)
- Another emerging use case that needs fast data sharing is stream processing
 - Track and update state in memory as events arrive
 - Large-scale reporting, click analysis, spam filtering, etc

Spark streaming

- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Spark Streaming receives live input data streams and divides the data into (micro)batches, which are then processed by the Spark engine to generate the final stream of results in batches.

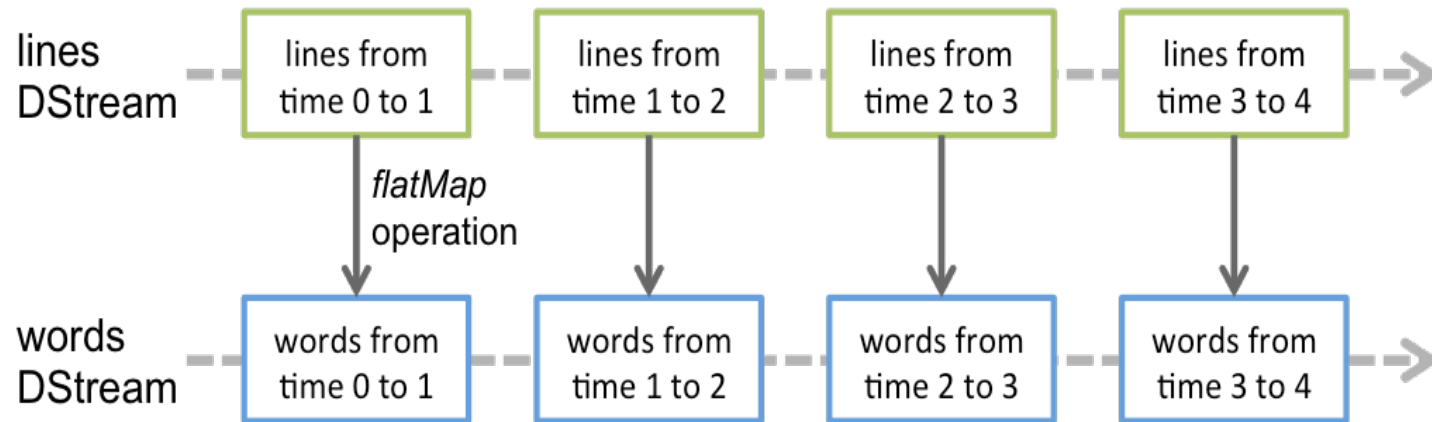


Data ingestion

- Many possible sources
- Usually a "publish & subscribe" system like Kafka

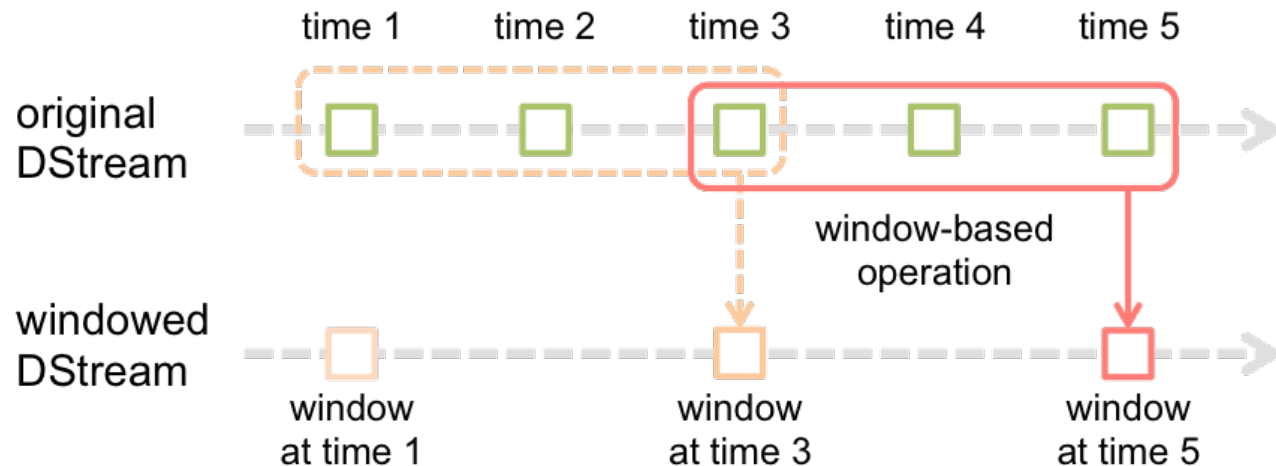


Spark operations on batches



Window operations

- Window operations are applied only to those RDDs that fall into a 'sliding' time window



- Fault-tolerance is supported by means of data replication among multiple Spark worker nodes.

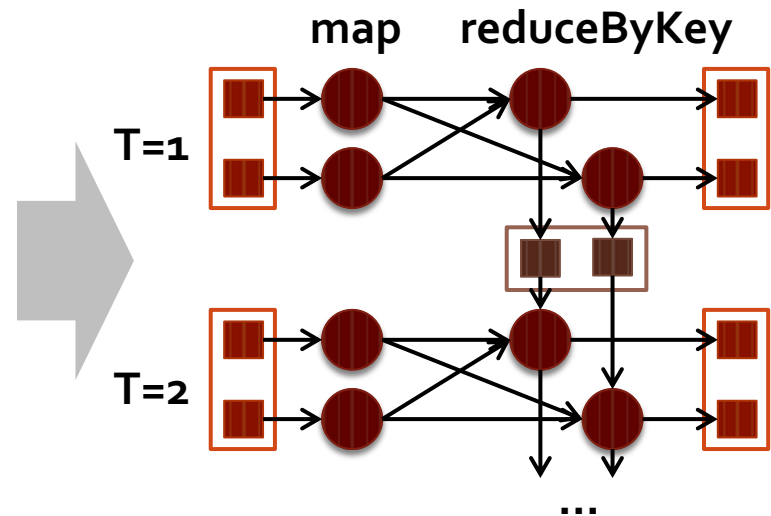
An example of Spark streaming

```
#StreamingContext is the main entry point streaming
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
# two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
# Create a lines DStream connected to a hostname:port
lines = ssc.socketTextStream("localhost", 9999)
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
# Print the first ten elements of each RDD generated in the stream
wordCounts.pprint()
ssc.start()                # Start the computation
ssc.awaitTermination()    # Wait for the computation to terminate
```


Run-time model

- Runs as a series of small (~ 1 s) batch jobs, keeping state in memory as fault-tolerant RDDs
- Intermix seamlessly with batch and ad-hoc queries

```
wordCounts =  
  scc.flatMap(lambda line:  
              line.split(" ")) \  
  .map(lambda word: (word, 1)) \  
  .reduceByKey(lambda a,b: a+b)
```

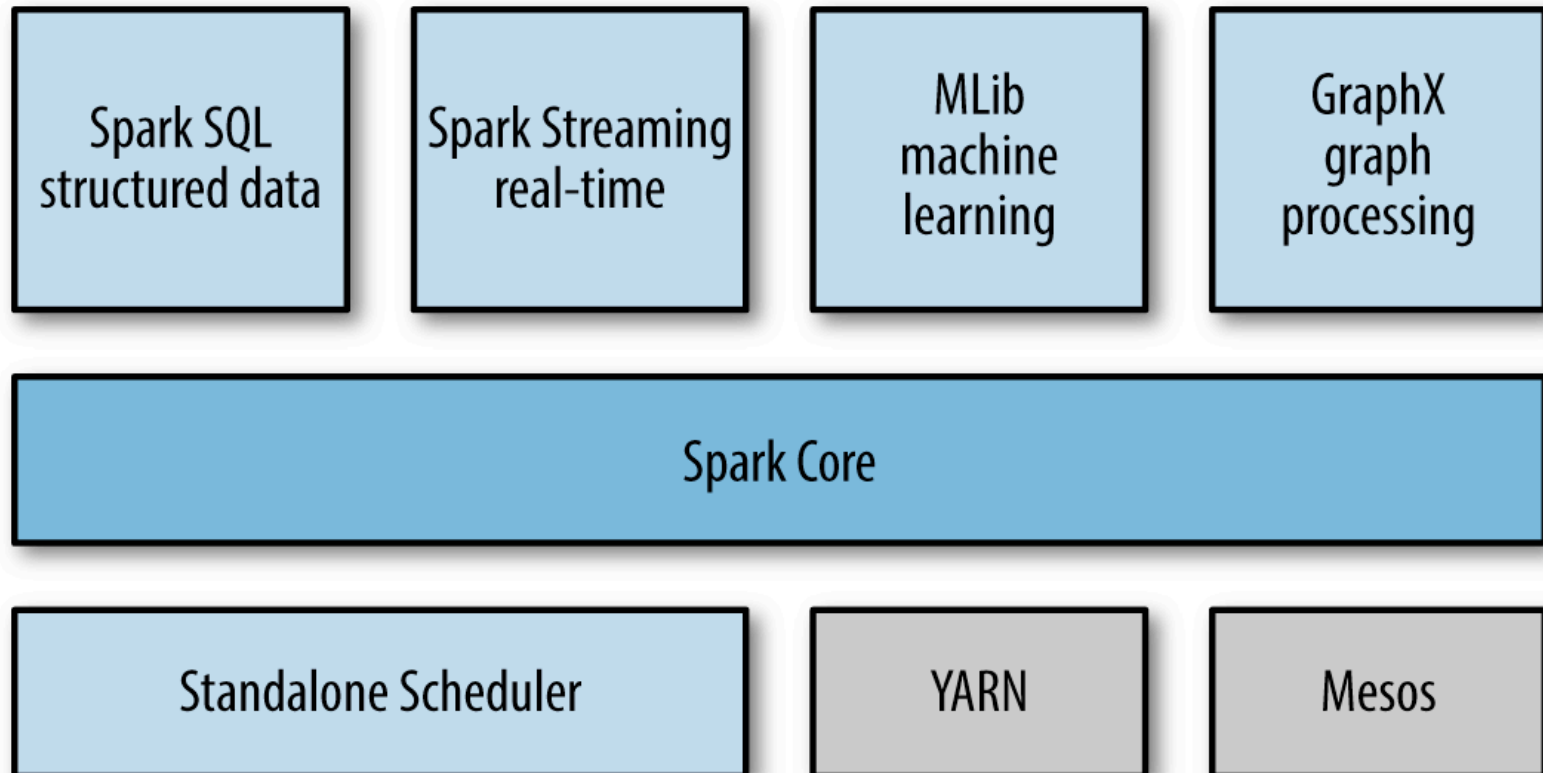


Can process 42 million records/second (4 GB/s)
on 100 nodes at sub-second latency

Spark MLlib & GraphX

- Spark MLlib: Machine Learning libraries included into the Spark API.
 - Machine Learning algorithms up to 100 times faster than implementations in MapReduce
 - Many algorithms and utilities: linear regression, clustering, frequent itemset mining, matrix decomposition,...
- Spark GraphX: API for graph processing and parallel graph computations integrated into Spark.
 - Including, amongst others, PageRank, strongly connected components, triangle count, etc.

Spark Software Stack



References

