CS4225/CS5425 Big Data Systems for Data Science

Introduction to Spark

Bryan Hooi
School of Computing
National University of Singapore
bhooi@comp.nus.edu.sg

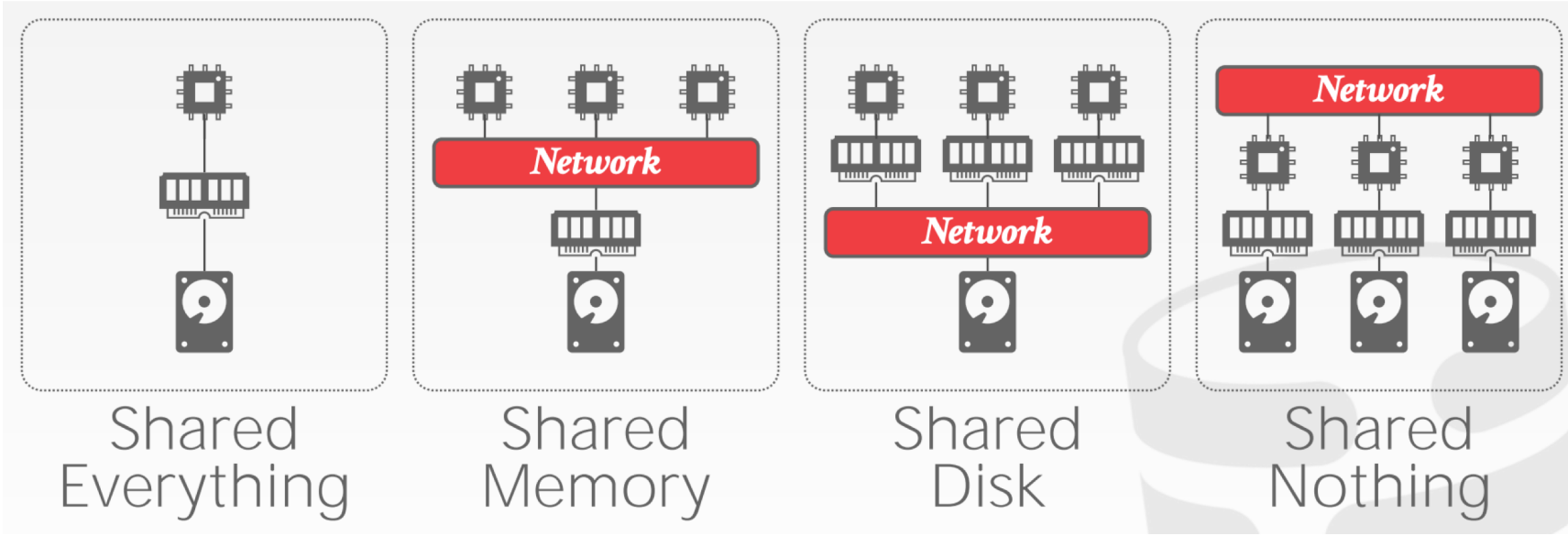
School of Computing

Announcements

- HW1 will be due 10 Oct 11.59pm.
  - For any issues, feel free to email me / post on LumiNUS / visit office hours
- HW2 will be released this weekend and will be due 4 weeks after that (on 31 Oct 11.59pm)

Week	Date	Topics	Tutorial	Due Dates
1	12 Aug	Overview and Introduction		
2	19 Aug	MapReduce - Introduction		
3	26 Aug	MapReduce and Relational Databases		
4	2 Sep	MapReduce and Data Mining	Tutorial: Hadoop	
5	9 Sep	NoSQL Overview 1		Assignment 1 released
6	16 Sep	NoSQL Overview 2		
Recess				
7	30 Sep	Apache Spark 1	Tutorial: NoSQL & Spark	Assignment 2 released
8	7 Oct	Apache Spark 2		Assignment 1 due (10 Oct)
9	14 Oct	Large Graph Processing 1	Tutorial: Graph Processing	
10	21 Oct	Large Graph Processing 2		
11	28 Oct	Stream Processing	Tutorial: Stream Processing	Assignment 2 due (31 Oct)
12	4 Nov	Deepavali – No Class		
13	11 Nov	**Test**		

Recap: Distributed Database Architectures



Examples:

Single node DBMS	Supercomputers	Cloud databases	NoSQL (mostly)

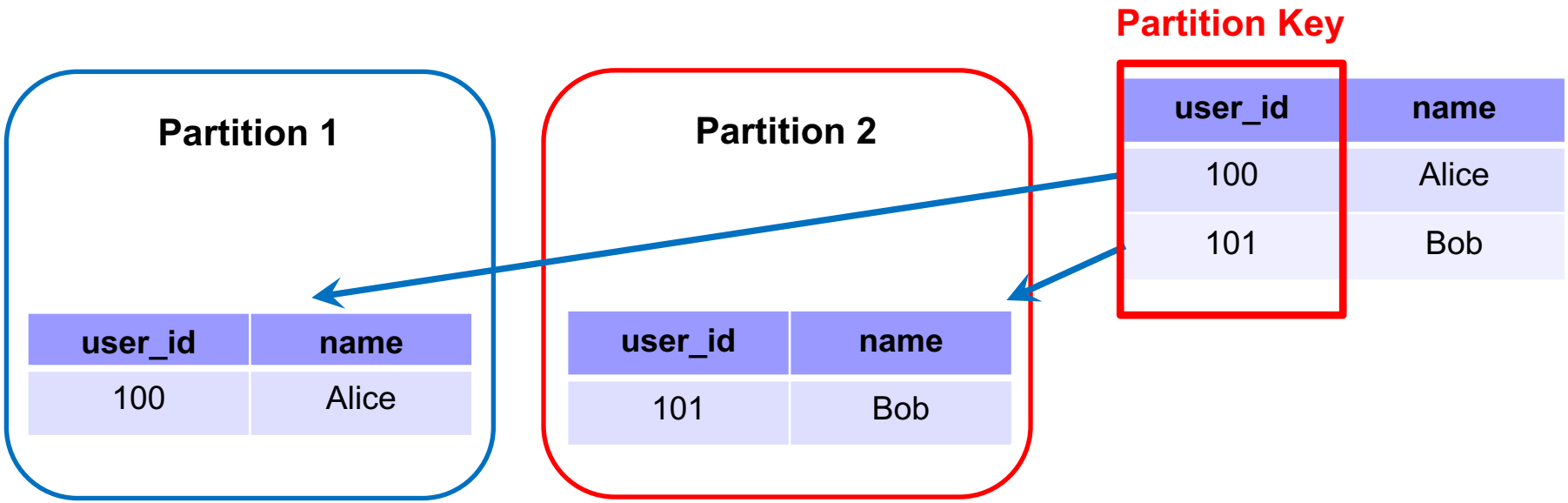
ORACLE RAC presto snowflake

Cockroach LABS H-Store redis cassandra mongoDB CouchDB relax MySQL Cluster

Credit: Andy Pavlo (CMU CS15-446)

Recap: Horizontal Partitioning

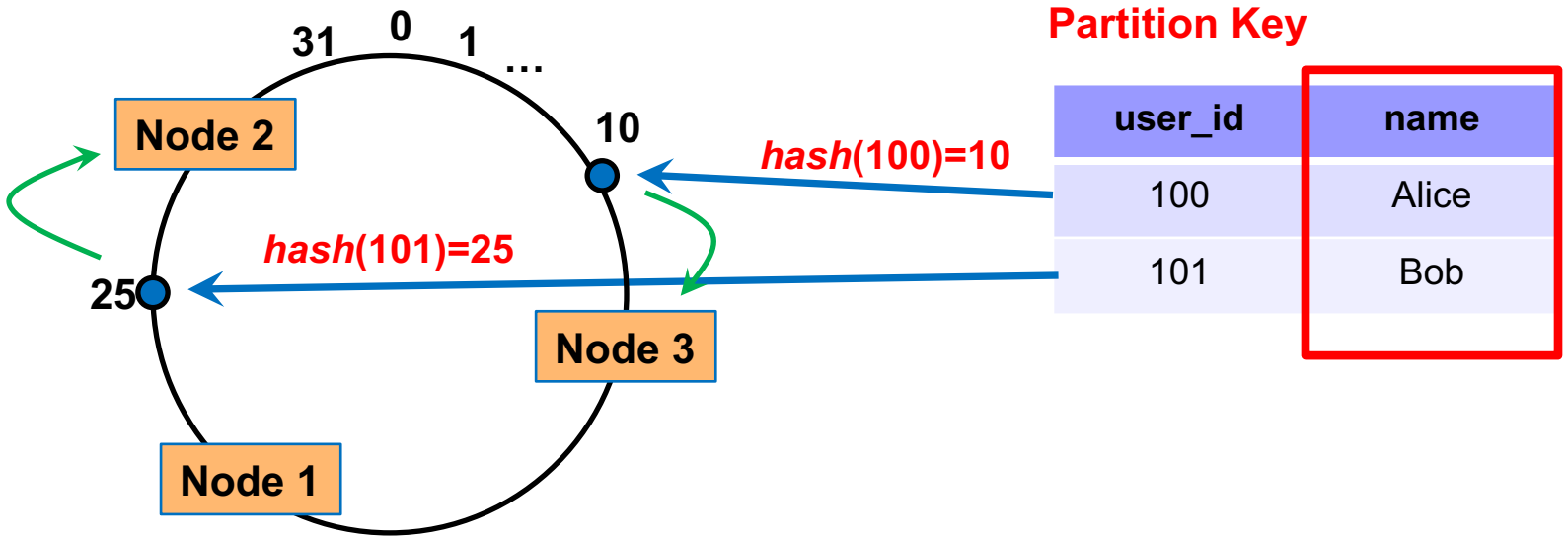
- Different tuples are stored in different nodes



- Can be done using **range partition** and **hash partition**
- **Partition Key** (or "shard key") is the variable used to decide which node each tuple will be stored on
  - **How to choose partition key?** If we often need to filter tuples based on a column, or "group by" a column, then that column is a suitable partition key
  - Example: if we filter tuples by user_id=100, and user_id is the partition key, then all the user_id=100 data will be on the same partition, improving locality and reducing network I/O.

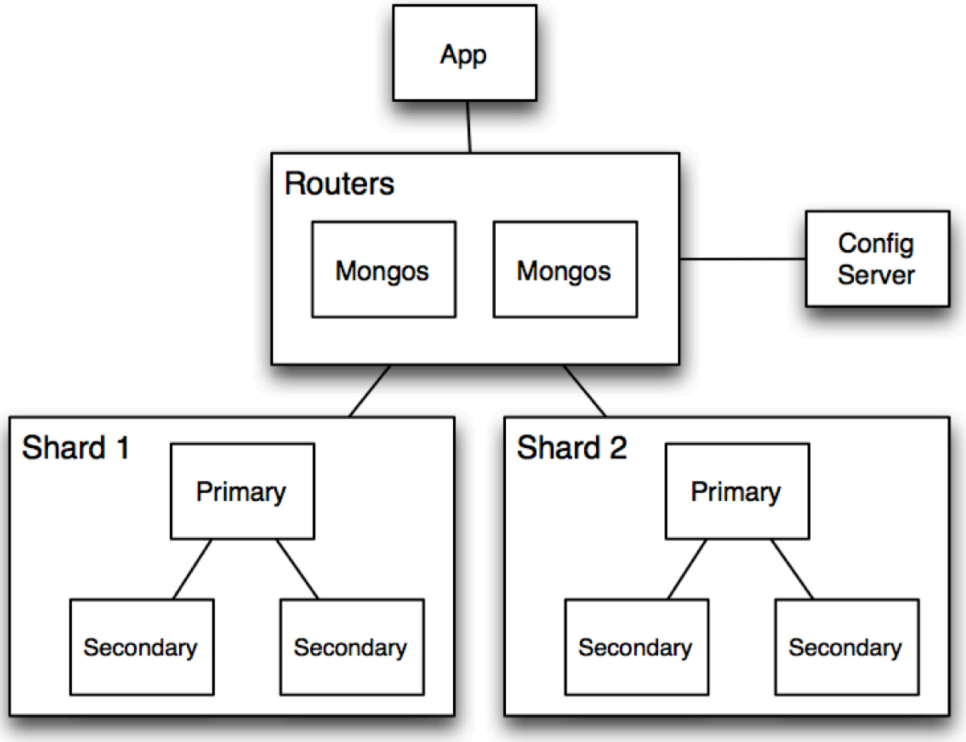
Recap: Consistent Hashing

- Think of the output of the hash function as lying on a circle:



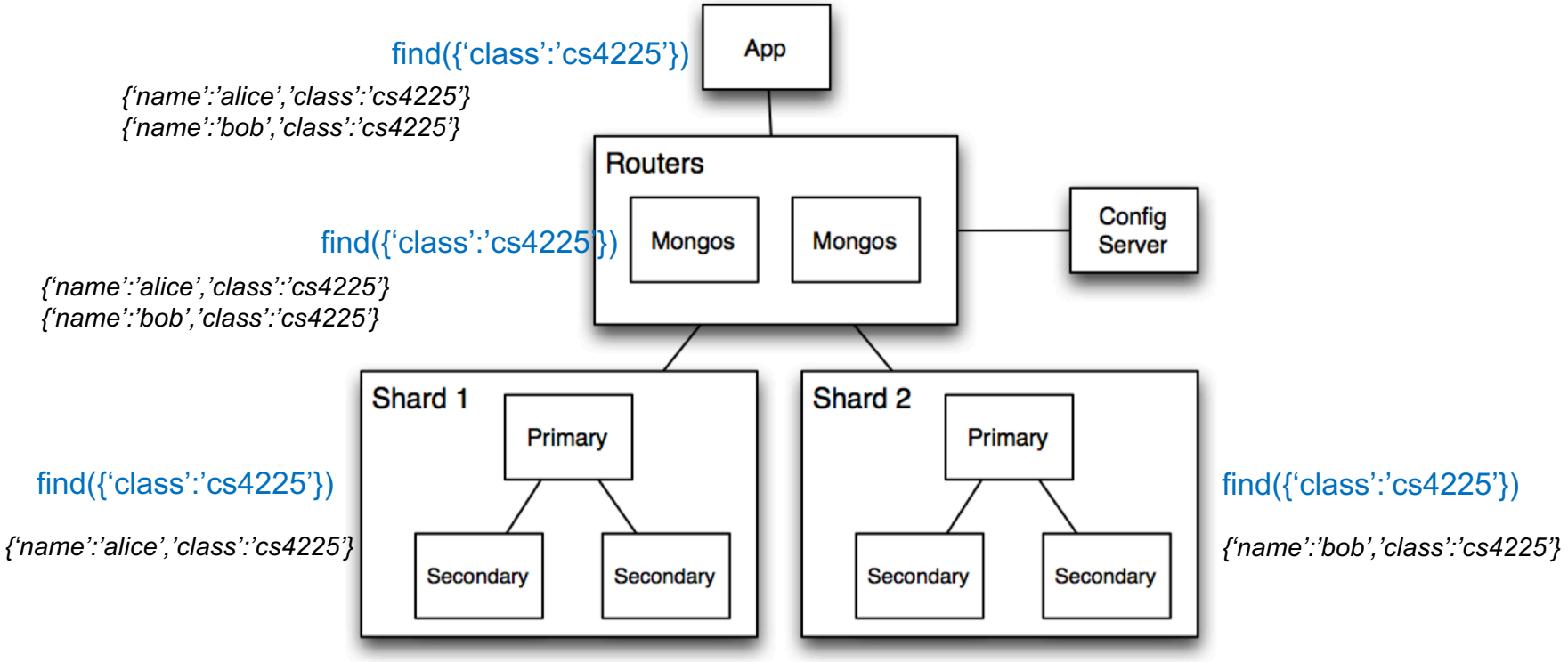
- **How to partition**: each node has a 'marker' (rectangles)
- Each tuple is placed on the circle, and assigned to the node that comes clockwise-after it
- To delete a node, we simply re-assign all its tuples to the node clock-wise after this one
- Similarly, nodes can be added by splitting the largest current node into two

Recap: Architecture of MongoDB



- **Routers (mongos)**: handle requests from application and route the queries to correct shards
- **Config Server**: stores metadata about which data is on which shard
- **Shards**: store data, and run queries on their data

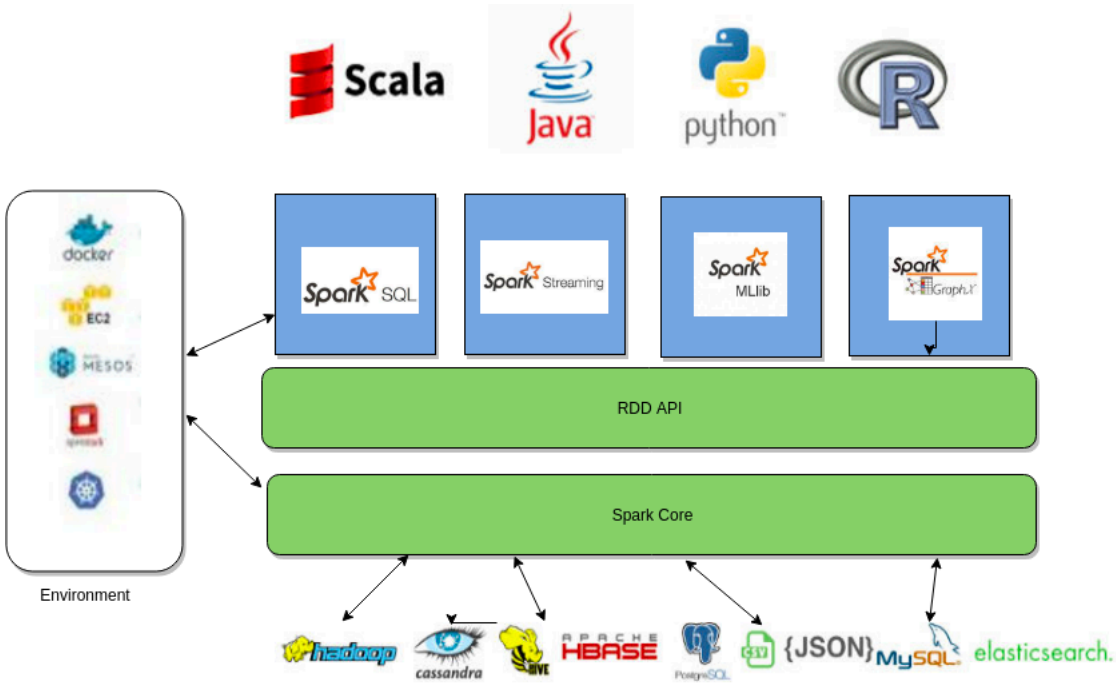
Recap: Example of Read or Write Query



1. Query is issued to a router (mongos) instance
2. With help of config server, mongos determines which shards should be queried
3. Query is sent to relevant shards
4. Shards run query on their data, and send results back to mongos
5. mongos merges the query results and returns the merged results

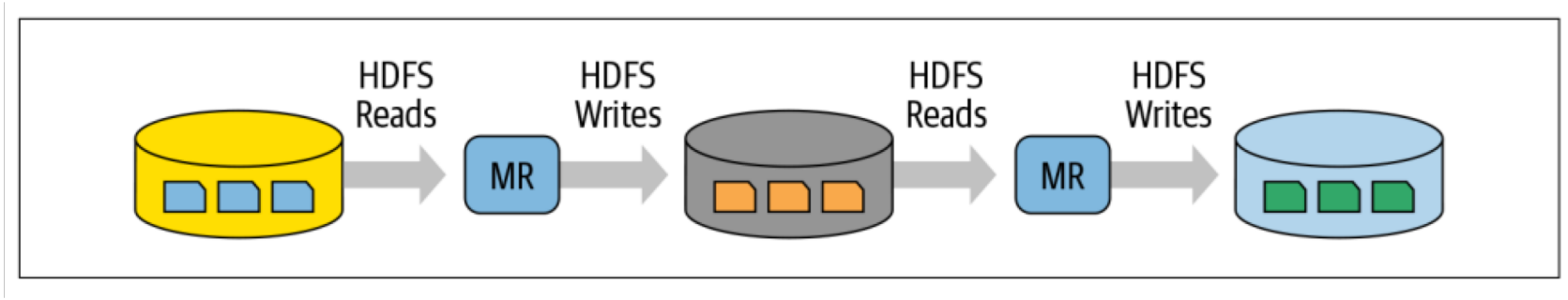
Today's Plan

- **Introduction and Basics**
- Working with RDDs
- Caching and DAGs
- DataFrames and Datasets



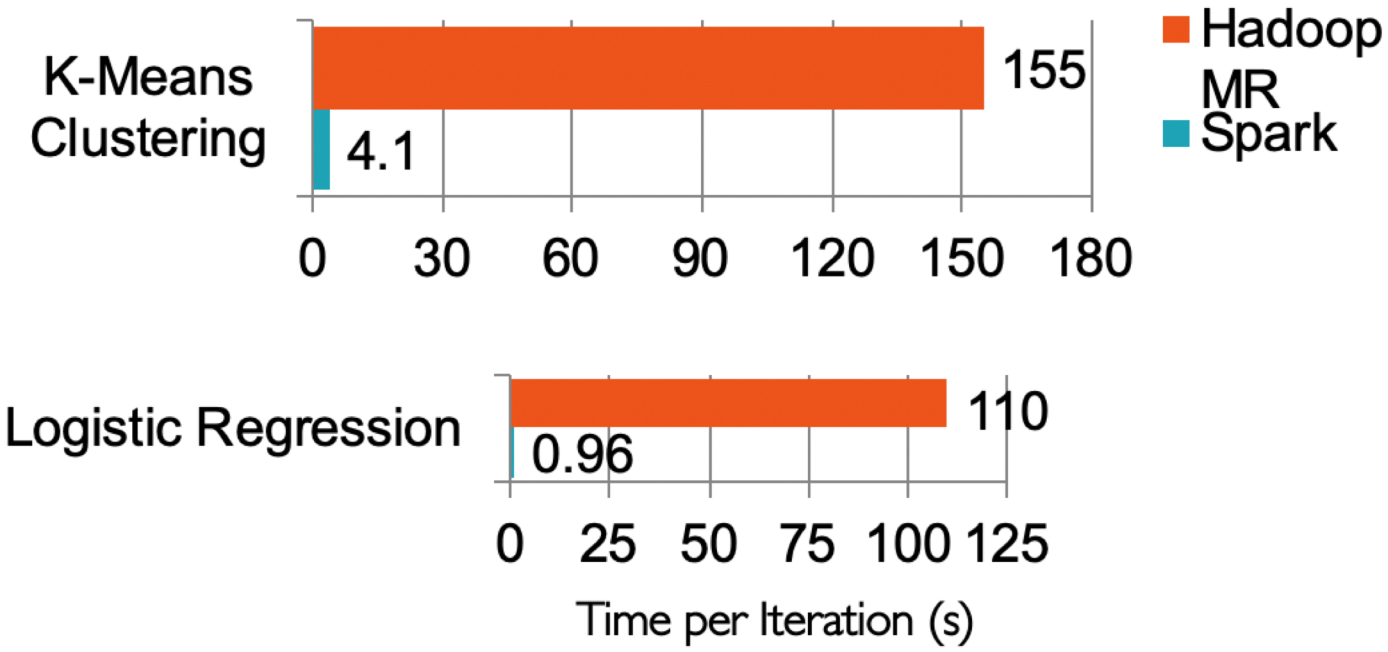
Source

Motivation: Hadoop vs Spark



- Issues with Hadoop Mapreduce:
  - **Network and disk I/O costs**: intermediate data has to be written to local disks and shuffled across machines, which is slow
  - Not suitable for **iterative** (i.e. modifying small amounts of data repeatedly) processing, such as interactive workflows, as each individual step has to be modelled as a MapReduce job.
- Spark stores most of its intermediate results in memory, making it much faster, especially for iterative processing
  - When memory is insufficient, Spark **spills to disk** which requires disk I/O

Performance Comparison



Ease of Programmability

```java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

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();
    Job job = Job.getInstance(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);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

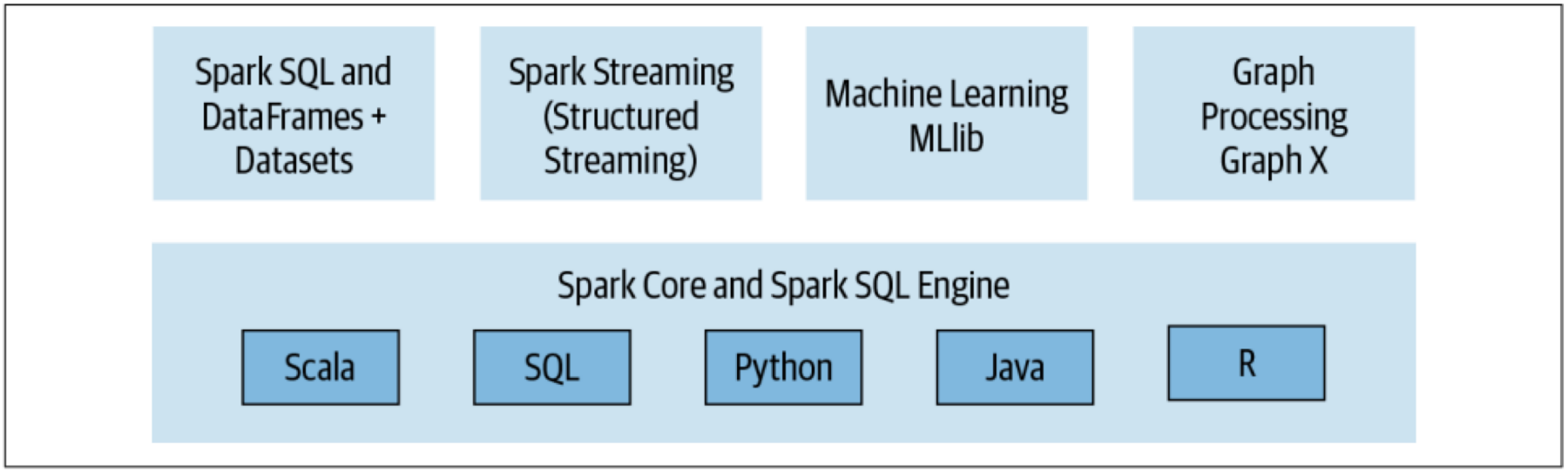
WordCount (Hadoop MapReduce)

Ease of Programmability

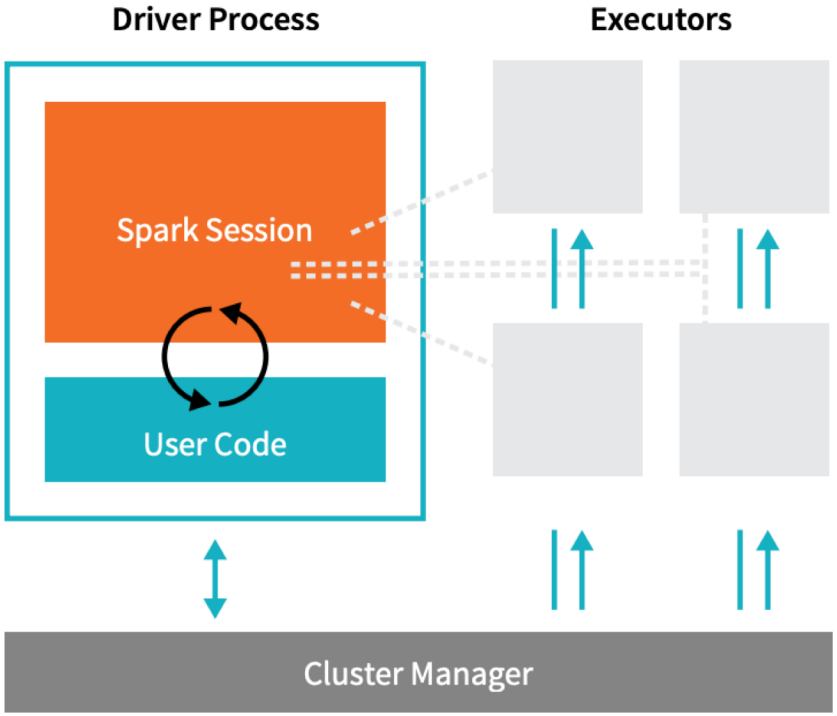
```
val file = spark.textFile("hdfs://...")

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

WordCount (Spark)

Spark Components and API Stack

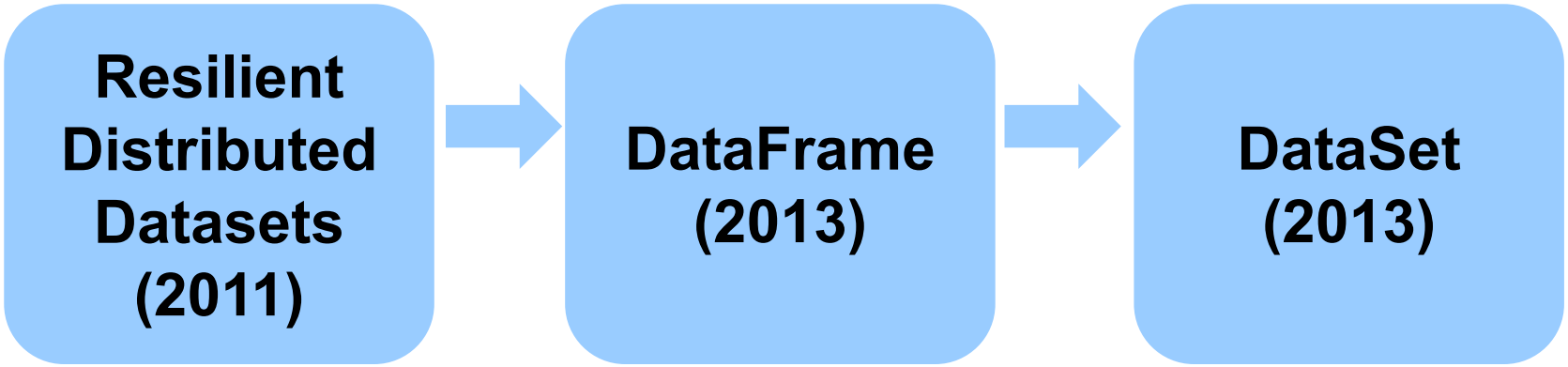


Spark Architecture



- **Driver Process** responds to user input, manages the Spark application etc., and distributes work to **Executors**, which run the code assigned to them and send the results back to the driver
- **Cluster Manager** (can be Spark's standalone cluster manager, YARN, Mesos or Kubernetes) allocates resources when the application requests it
- In **local mode,** all these processes run on the same machine

Evolution of Spark APIs



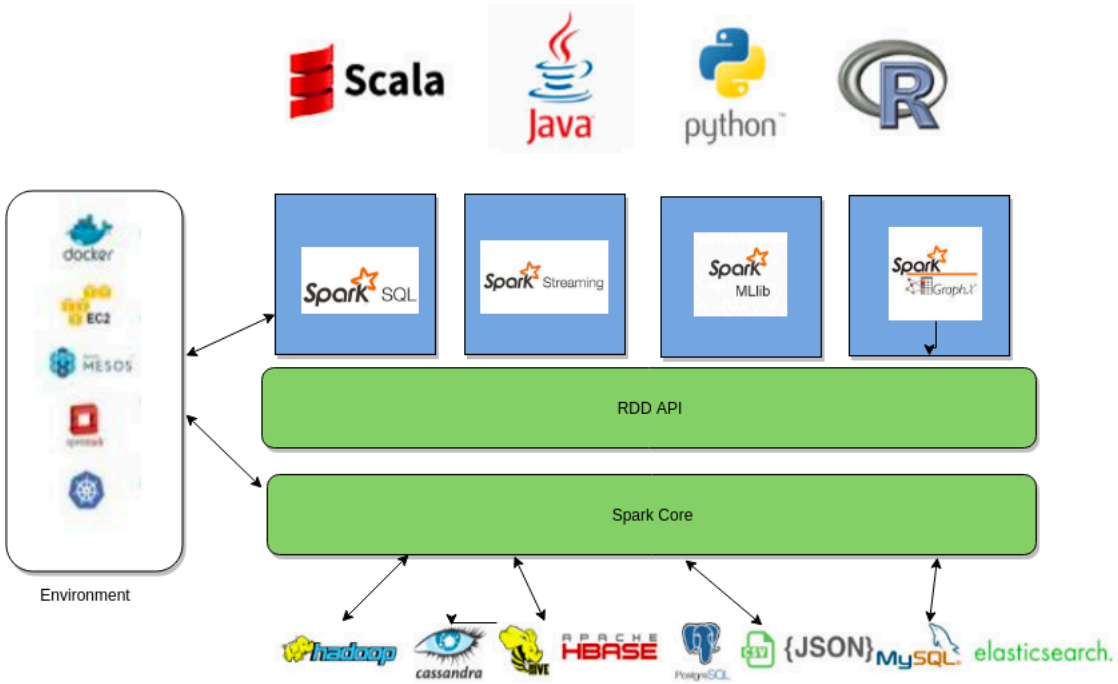
- A collection of JVM objects
- Functional operators (map, filter, etc)

- A collection of Row objects
- Expression-based operations
- Logical plans and optimizer

- Internally rows, externally JVM objects.
- Almost the "Best of both worlds": type safe + fast

Today's Plan

- Introduction and Basics
- **Working with RDDs**
- Caching and DAGs
- DataFrames and Datasets



Source

Achieve fault tolerance through **lineages**

Represent a collection of objects that is **distributed over machines**

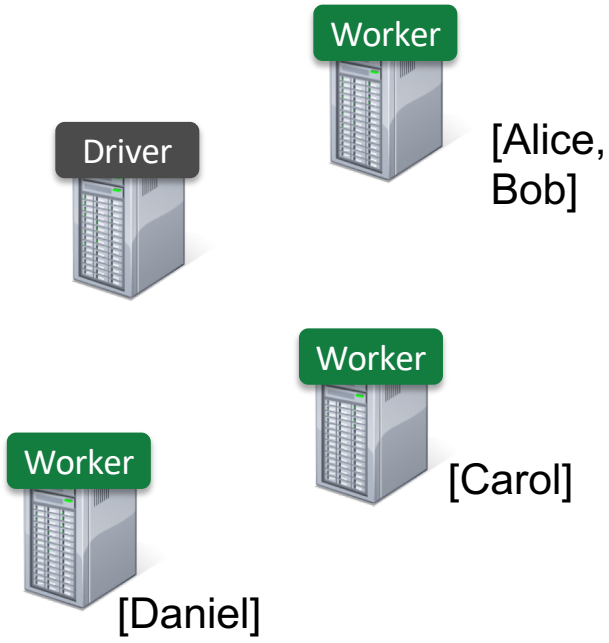
Resilient Distributed Datasets (RDDs)

RDD: Distributed Data

```
# Create an RDD of names, distributed over 3 partitions
dataRDD = sc.parallelize(["Alice", "Bob", "Carol", "Daniel"], 3)
```

Partition data into 3 parts

- RDDs are **immutable**, i.e. they cannot be changed once created.
- This is an RDD with 4 strings. In actual hardware, it will be partitioned into the 3 workers.

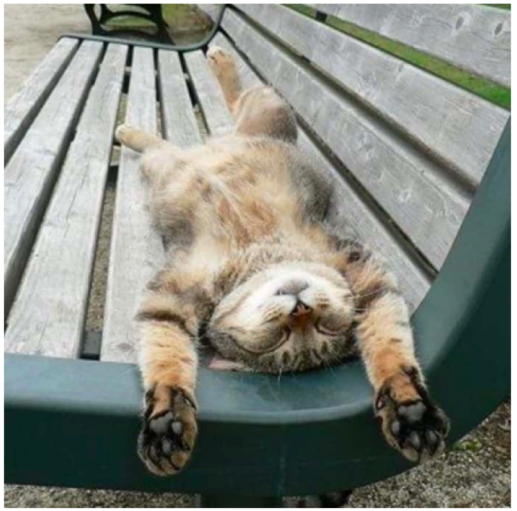


Transformations

- **Transformations** are a way of transforming RDDs into RDDs.

```
# Create an RDD of tuples (name, age)

dataRDD = sc.parallelize(["Alice", "Bob", "Carol",
"Daniel"], 3)

nameLen = dataRDD.map(lambda s: len(s))
```

- This represents the transformation that maps each string to its length, creating a new RDD.
- However, transformations are **lazy**. This means the transformation will not be executed yet, until an **action** is called on it
  - Q: what are the advantages of being lazy?
  - A: Spark can optimize the query plan to improve speed (e.g. removing unneeded operations)
- Examples of transformations: `map, order, groupBy, filter, join, select`



Actions

- **Actions** trigger Spark to compute a result from a series of transformations.

```
dataRDD = sc.parallelize(["Alice", "Bob", "Carol",
"Daniel"], 3)

nameLen = dataRDD.map(lambda s: len(s))

nameLen.collect()
```

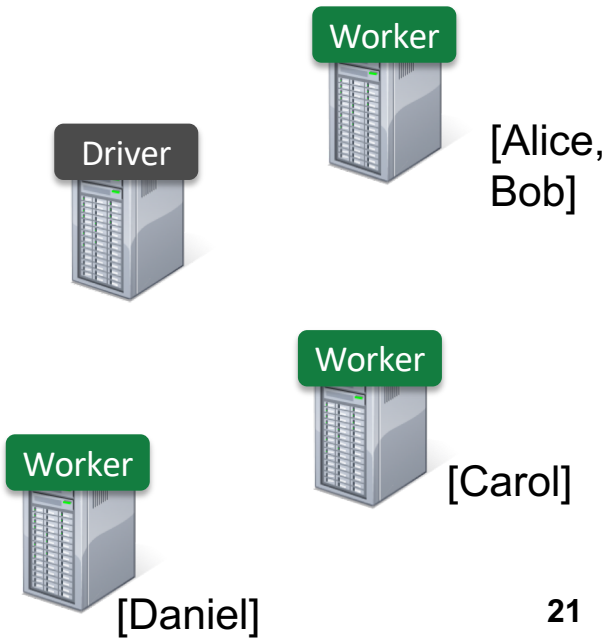
```
[5, 3, 5, 6]
```

- collect() here is an action.
  - It is the action that asks Spark to retrieve all elements of the RDD to the driver node.
- Examples of actions: `show`, `count`, `save`, `collect`

Distributed Processing

```
# Create an RDD of tuples (name, age)

dataRDD = sc.parallelize(["Alice", "Bob", "Carol",
"Daniel"], 3)

nameLen = dataRDD.map(lambda s: len(s))

nameLen.collect()
```

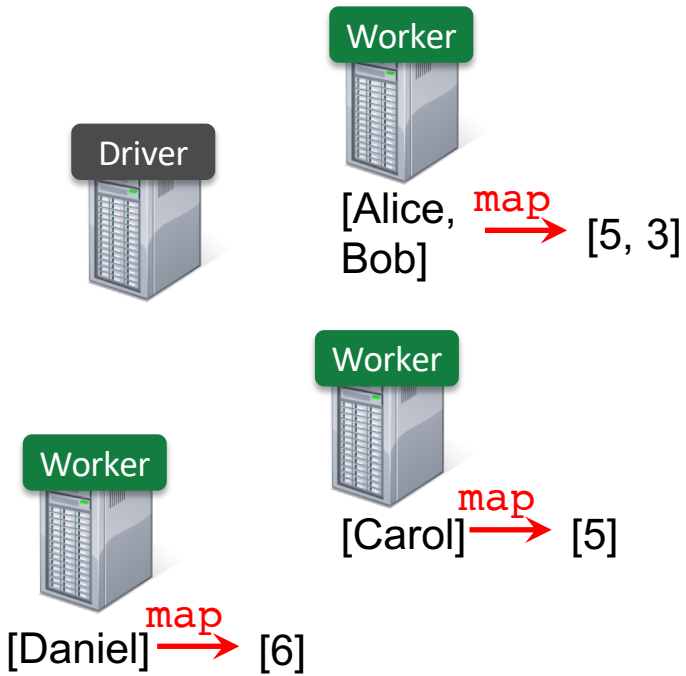
- As we previously said, RDDs are actually distributed across machines.
- Thus, the transformations and actions are executed in parallel. The results are only sent to the driver in the final step.



Distributed Processing

```
# Create an RDD of tuples (name, age)

dataRDD = sc.parallelize(["Alice", "Bob", "Carol",
"Daniel"], 3)

nameLen = dataRDD.map(lambda s: len(s))

nameLen.collect()
```

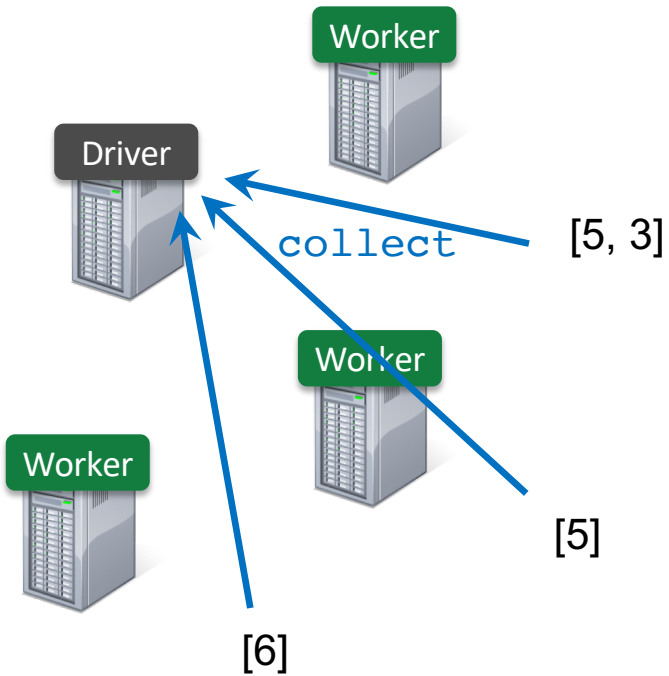
- As we previously said, RDDs are actually distributed across machines.
- Thus, the transformations and actions are executed in parallel. The results are only sent to the driver in the final step.



Distributed Processing

```
# Create an RDD of tuples (name, age)

dataRDD = sc.parallelize(["Alice", "Bob", "Carol",
"Daniel"], 3)

nameLen = dataRDD.map(lambda s: len(s))

nameLen.collect()
```

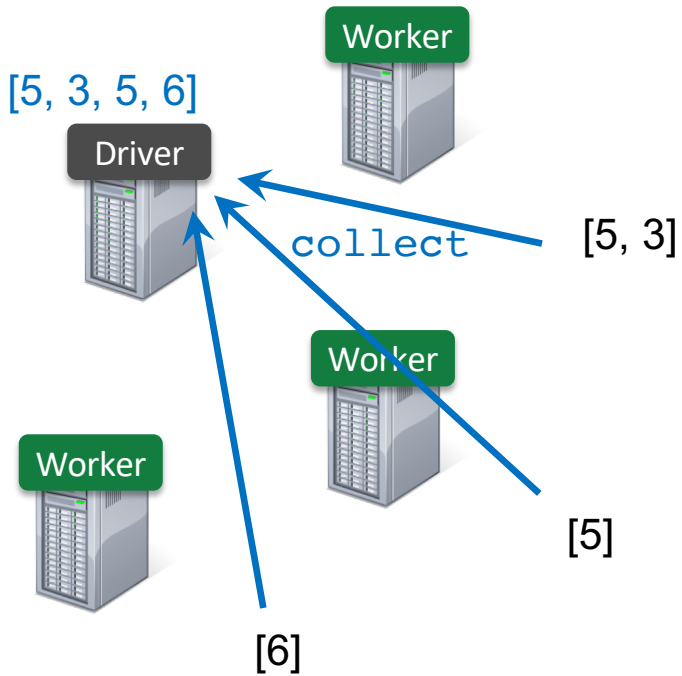
- As we previously said, RDDs are actually distributed across machines.
- Thus, the transformations and actions are executed in parallel. The results are only sent to the driver in the final step.



Distributed Processing

```
# Create an RDD of tuples (name, age)

dataRDD = sc.parallelize(["Alice", "Bob", "Carol",
"Daniel"], 3)

nameLen = dataRDD.map(lambda s: len(s))

nameLen.collect()
```

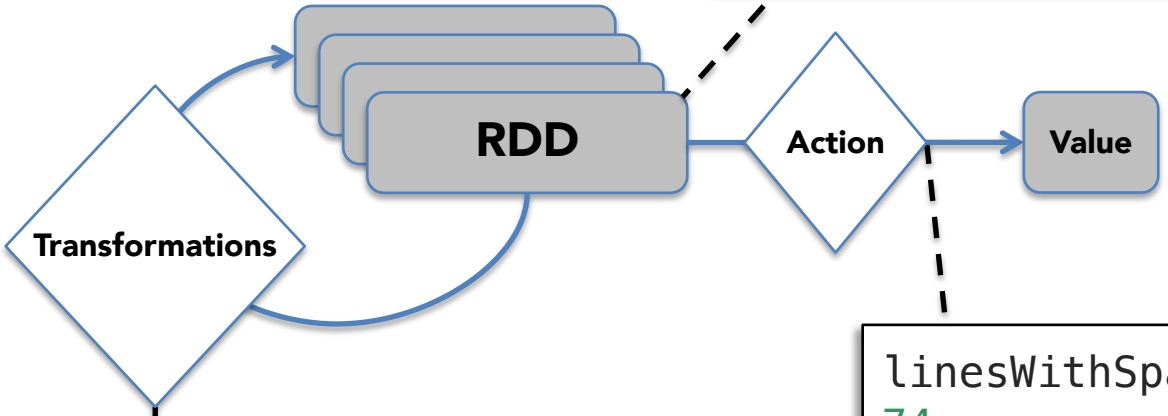
- As we previously said, RDDs are actually distributed across machines.
- Thus, the transformations and actions are executed in parallel. The results are only sent to the driver in the final step.



Working with RDDs

Note: this reads the file on each worker node in parallel, not on the driver node

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

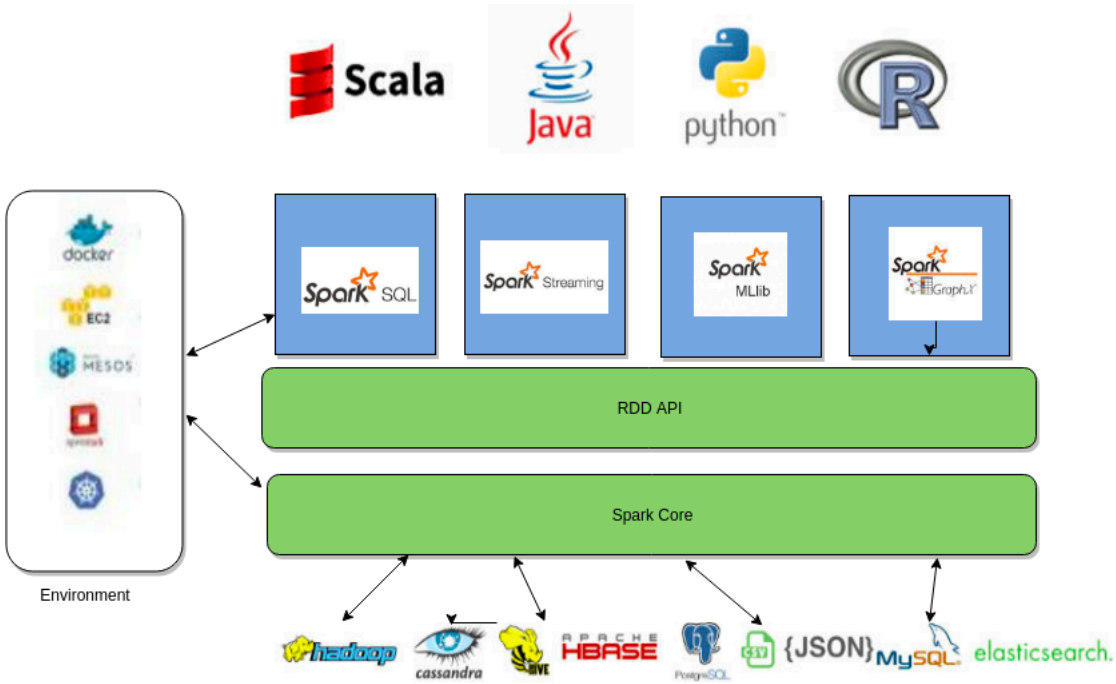



```
linesWithSpark.count()
74

linesWithSpark.first()
# Apache Spark
```

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

Today's Plan

- Introduction and Basics
- Working with RDDs
- **Caching and DAGs**
- DataFrames and Datasets

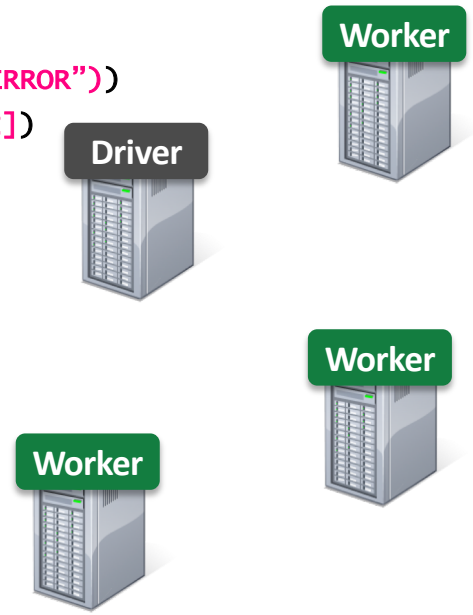


Source

Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

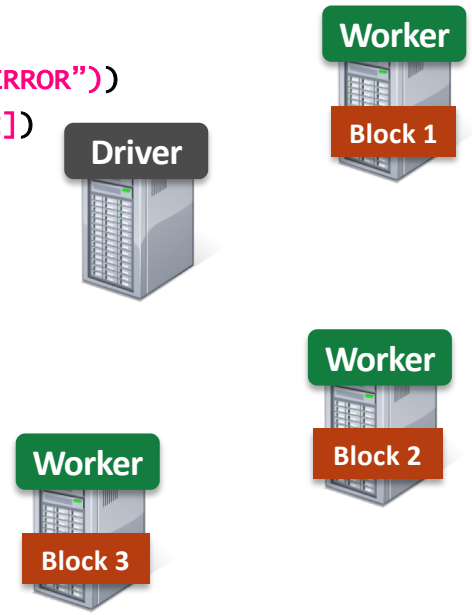
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

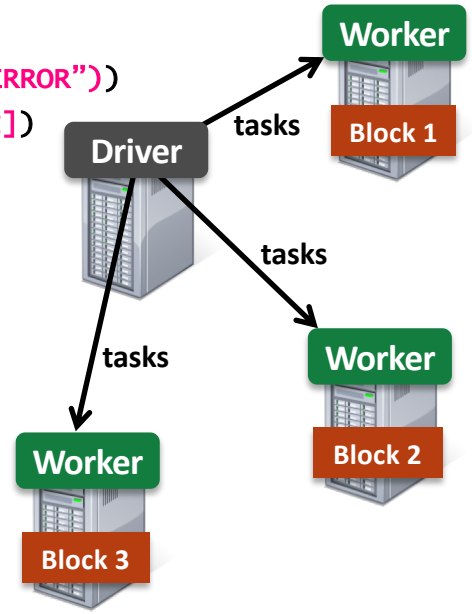
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

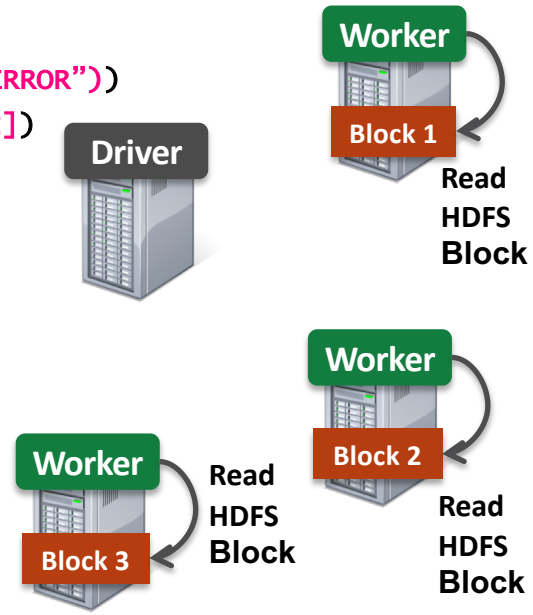
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

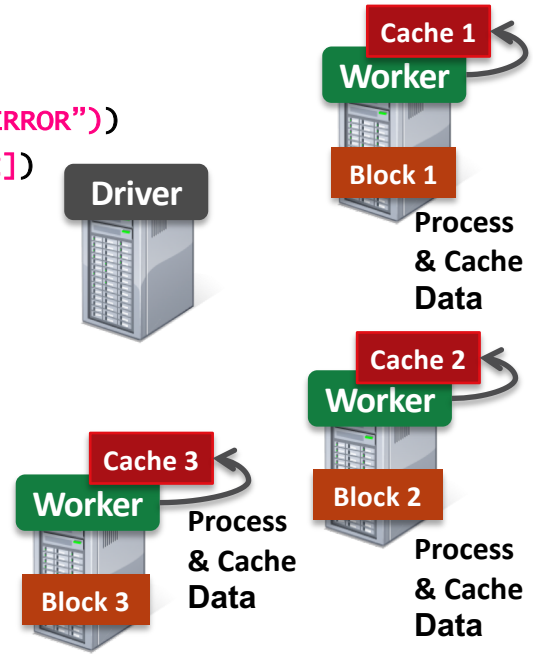
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

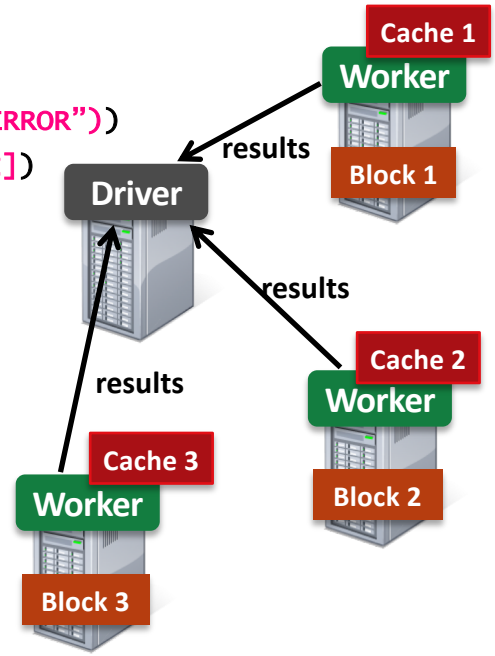
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

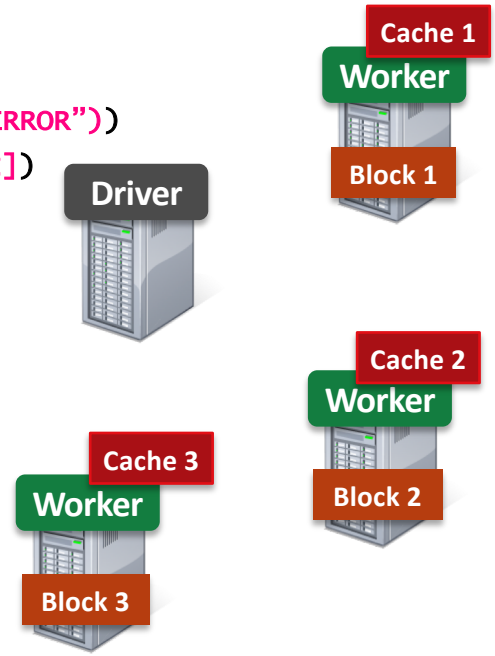
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

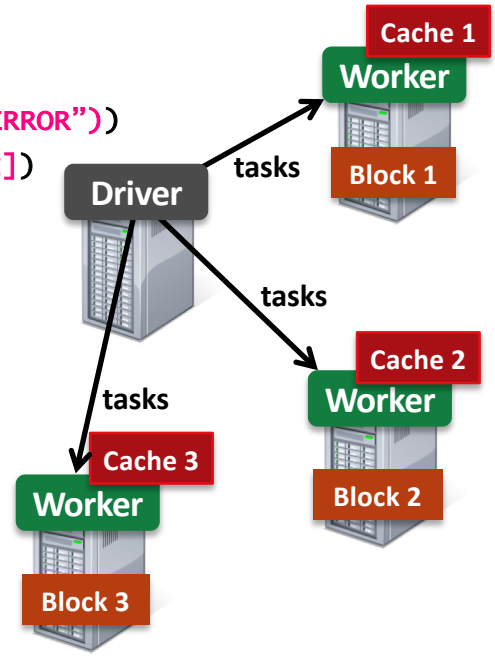
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()

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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

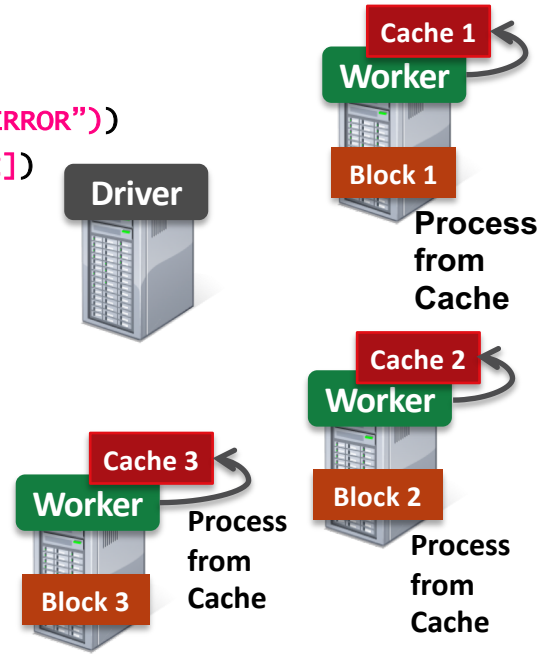
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

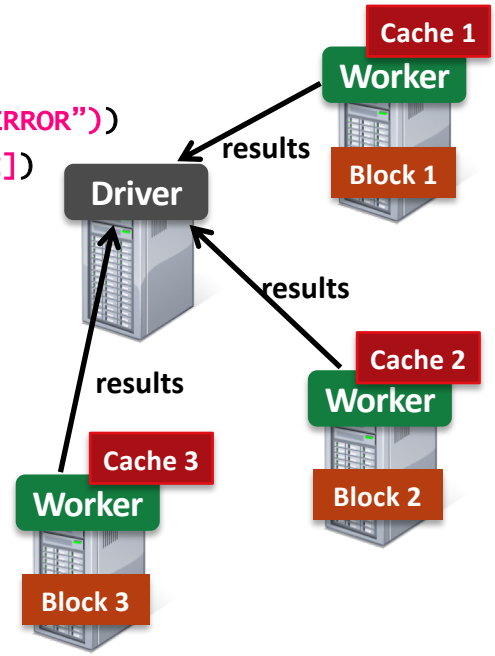
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


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



Caching

Log Mining example: Load error messages from a log into memory, then interactively search for various patterns

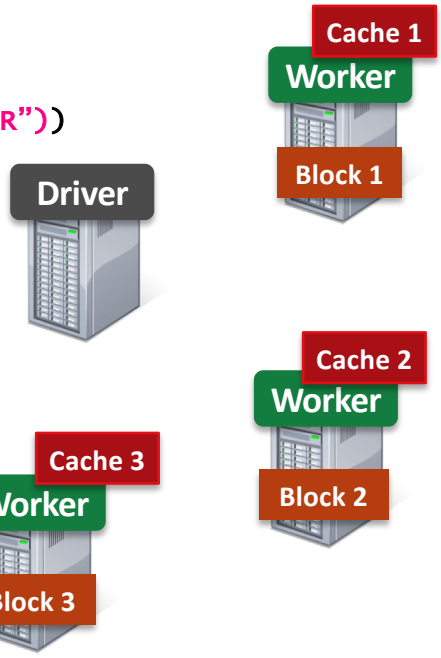
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


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



Caching

Log Mining example: 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()
```

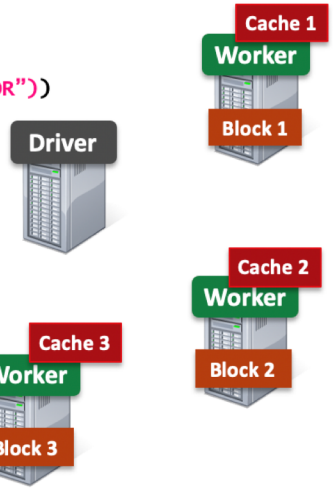


Cache your data ➔ Faster Results
Full-text search of Wikipedia
- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk

Caching

- `cache()`: saves an RDD to memory (of each worker node).
- `persist(options)`: can be used to save an RDD to memory disk, or off-heap memory
- When should we cache or not cache an RDD?
  - When it is expensive to compute and needs to be re-used multiple times.
  - If worker nodes have not enough memory, they will evict the "least recently used" RDDs. So, be aware of memory limitations when caching.

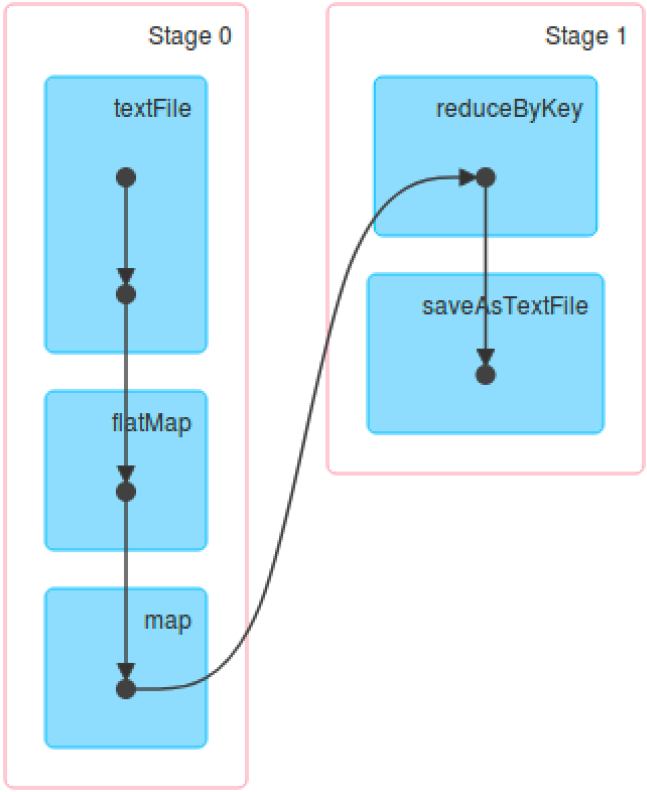
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

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

Directed Acyclic Graph (DAG)

- Internally, Spark creates a graph ("directed acyclic graph") which represents all the RDD objects and how they will be transformed.
- Transformations construct this graph; actions trigger computations on it.

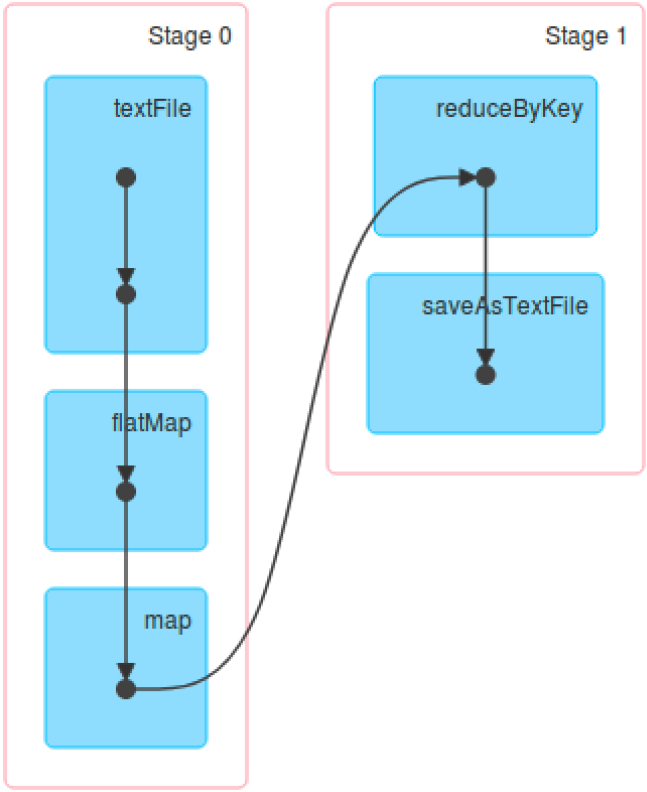



```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.save("...")
```

WordCount (Spark)

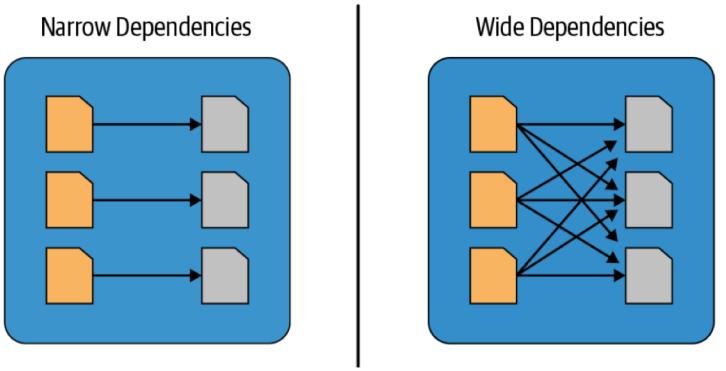
Lineage and Fault Tolerance

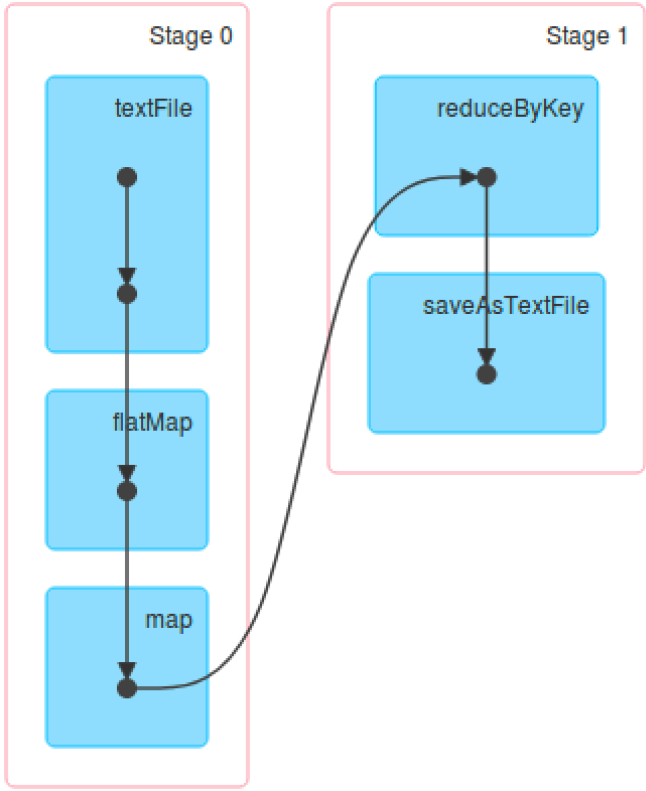
- Unlike Hadoop, Spark does not use replication to allow fault tolerance. Why?
  - Spark tries to store all the data in memory, not disk. Memory capacity is much more limited than disk, so simply duplicating all data is expensive.
- **Lineage approach**: if a worker node goes down, we replace it by a new worker node, and use the graph (DAG) to recompute the data in the lost partition.
  - Note that we only need to recompute the RDDs from the lost partition.



Narrow and Wide Dependencies

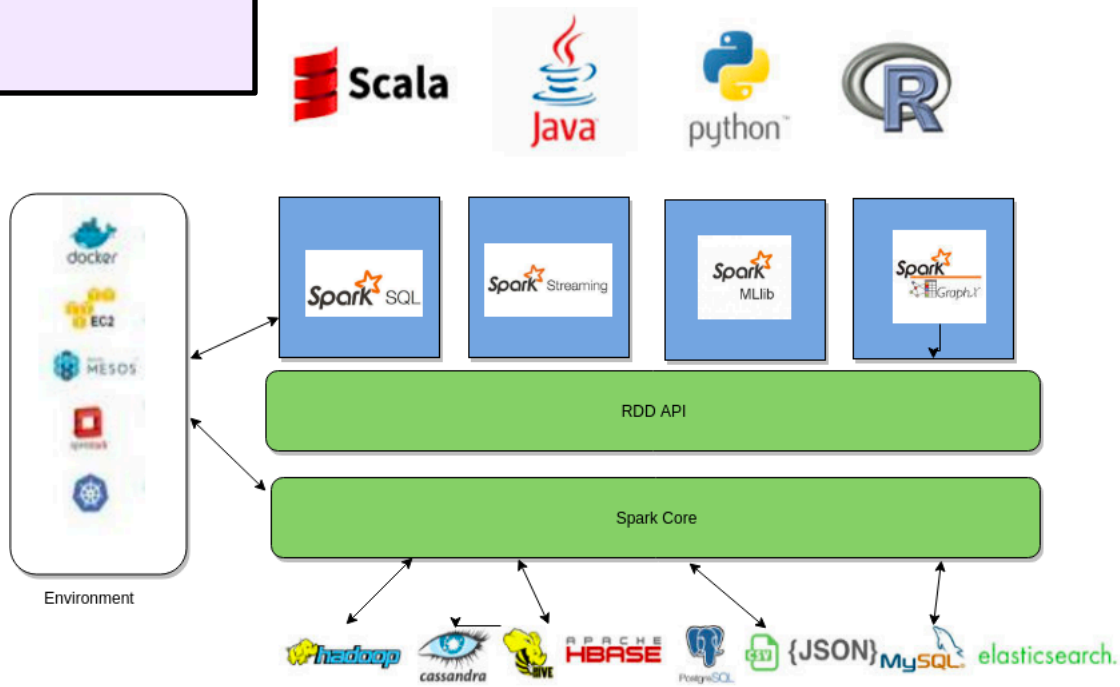
- **Narrow dependencies** refer to transformations which only require data from 1 partition
  - E.g. map, filter, contains
- **Wide dependencies** refer to transformations which require multiple partitions
  - E.g. groupBy, orderBy
- In the DAG, consecutive narrow dependencies are grouped together as "stages".
- Data only needs to be shuffled (i.e. exchange of data, incurring disk and network I/O) when going between stages.
- Within stages, Spark tries to perform consecutive transformations on the same machines.
- Minimizing shuffling is good practice for improving performance.





Today's Plan

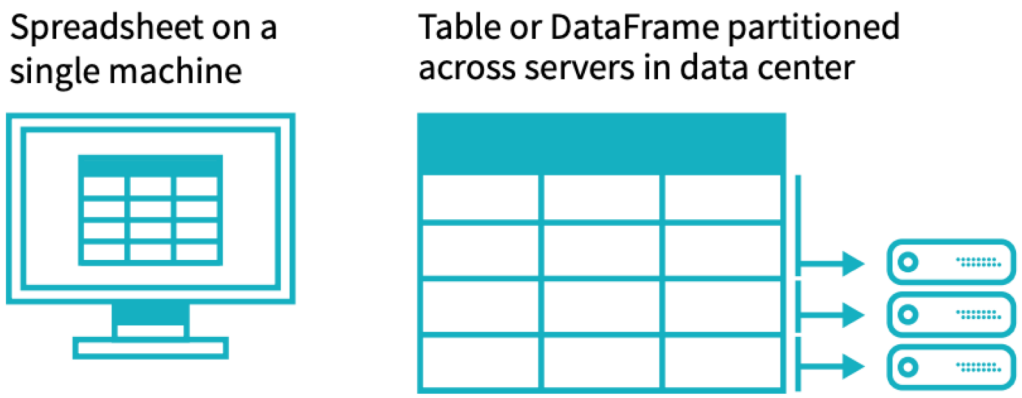
- Introduction and Basics
- Working with RDDs
- Caching and DAGs
- **DataFrames and Datasets**



Source

DataFrames

- A DataFrame represents a table of data, similar to tables in SQL, or DataFrames in pandas.
- Compared to RDDs, this is a higher level interface, e.g. it has transformations that resemble SQL operations.
  - DataFrames (and Datasets) are the recommended interface for working with Spark – they are easier to use than RDDs and almost all tasks can be done with them, while only rarely using the RDD functions.
  - However, all DataFrame operations are still ultimately compiled down to RDD operations by Spark.



DataFrames: example

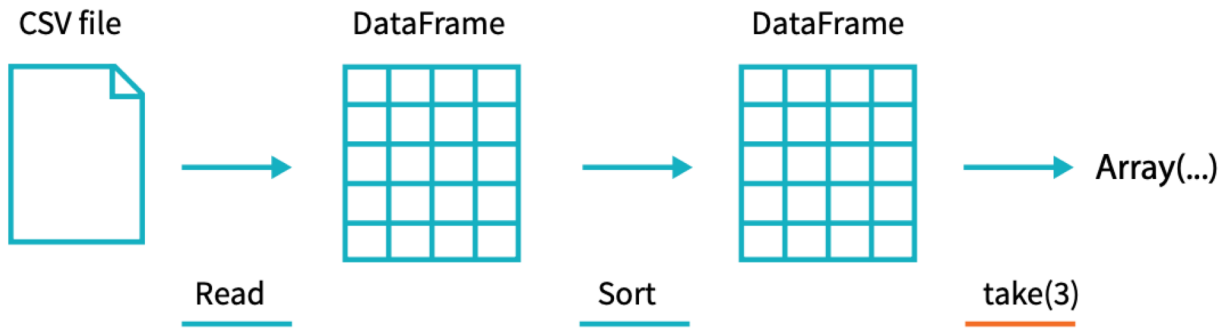
```
flightData2015 = spark\
.read\
.option("inferSchema", "true")\
.option("header", "true")\
.csv("/mnt/defg/flight-data/csv/2015-summary.csv")
```

- Reads in a DataFrame from a CSV file.

```
flightData2015.sort("count").take(3)
```

- Sorts by 'count' and output the first 3 rows (action)

```
Array([United States,Romania,15], [United States,Croatia...
```



DataFrames: transformations

- An easy way to transform DataFrames is to use SQL queries. This takes in a DataFrame and returns a DataFrame (the output of the query).

```
flightData2015.createOrReplaceTempView("flight_data_2015")
maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
maxSql.collect()
```

DataFrames: DataFrame interface

- We can also run the exact same query as follows:

```
from pyspark.sql.functions import desc
flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.collect()
```

- Generally, these transformation functions (groupBy, sort, …) take in either strings or "column objects", which represent columns.
  - For example, "desc" here returns a column object.

Datasets

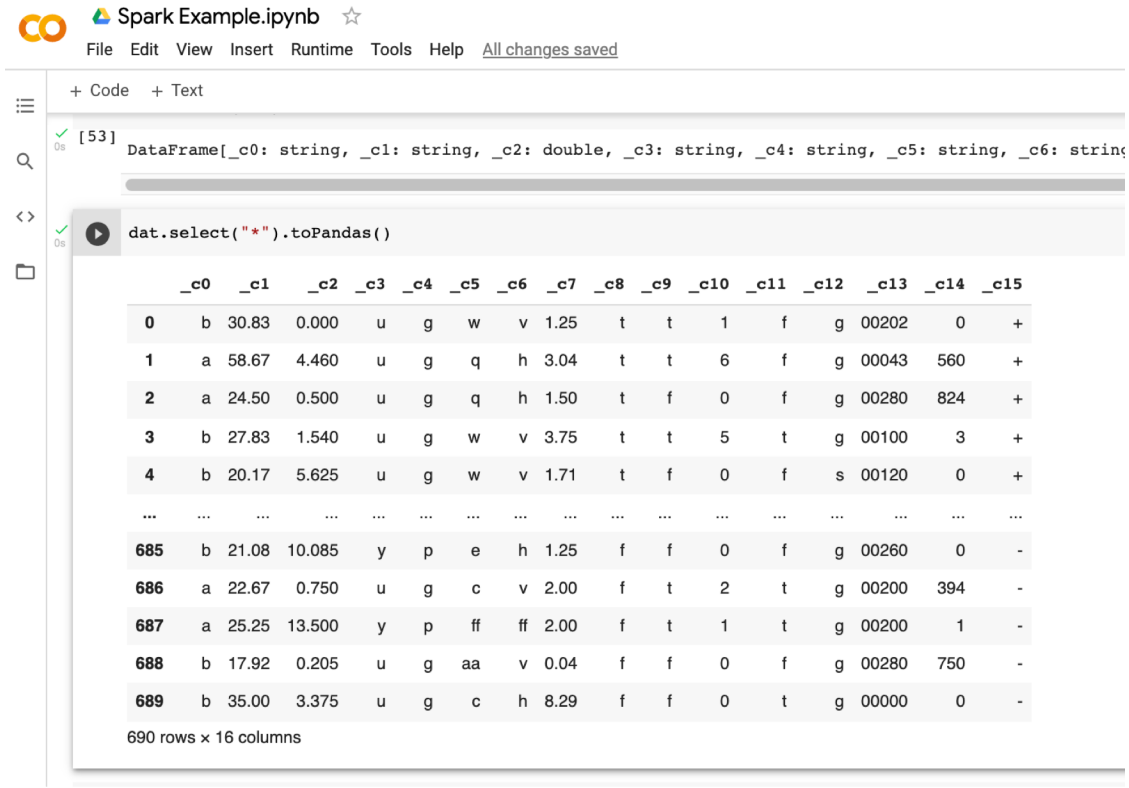
- Datasets are similar to DataFrames, but are type-safe.
  - In fact, in Spark (Scala), DataFrame is just an alias for Dataset[Row]
  - However, Datasets are not available in Python and R, since these are dynamically typed languages

```
case class Flight(DEST_COUNTRY_NAME: String,
ORIGIN_COUNTRY_NAME: String, count: BigInt)

val flightsDF = spark.read.parquet("/mnt/defg/flight-
data/parquet/2010-summary.parquet/")

val flights = flightsDF.as[Flight]

flights.collect()
```

- The Dataset `flights` is type safe – its type is the "Flight" class.
- Now when calling `collect()`, it will also return objects of the "Flight" class, instead of rows.

Example Spark Notebook

- To experiment with simple Spark commands without needing to install / setup anything on your computer, you can run Spark on Google Colab
- See the simple example notebook at https://colab.research.google.com/drive/1qtNpkieNEUzyF2NnXTyqyGL3LQD1TVII#scrollTo=pUgUMWYUKAU3



	_c0	_c1	_c2	_c3	_c4	_c5	_c6	_c7	_c8	_c9	_c10	_c11	_c12	_c13	_c14	_c15
0	b	30.83	0.000	u	g	w	v	1.25	t	t	1	f	g	00202	0	+
1	a	58.67	4.460	u	g	q	h	3.04	t	t	6	f	g	00043	560	+
2	a	24.50	0.500	u	g	q	h	1.50	t	f	0	f	g	00280	824	+
3	b	27.83	1.540	u	g	w	v	3.75	t	t	5	t	g	00100	3	+
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0	f	s	00120	0	+
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
685	b	21.08	10.085	y	p	e	h	1.25	f	f	0	f	g	00260	0	-
686	a	22.67	0.750	u	g	c	v	2.00	f	t	2	t	g	00200	394	-
687	a	25.25	13.500	y	p	ff	ff	2.00	f	t	1	t	g	00200	1	-
688	b	17.92	0.205	u	g	aa	v	0.04	f	f	0	f	g	00280	750	-
689	b	35.00	3.375	u	g	c	h	8.29	f	f	0	t	g	00000	0	-

690 rows × 16 columns

Acknowledgements

- https://www.pinterest.com/pin/739364463807740043/
- https://colab.research.google.com/github/jmbanda/BigDataProgramming_2019/blob/master/Chapter_5_Loading_and_Saving_Data_in_Spark.ipynb
- Databricks, "The Data Engineer's Guide to Spark"
- Jules S. Damji, Brooke Wenig, Tathagata Das & Denny Lee, "Learning Spark: Lightning-Fast Data Analytics"