



Introduction to Apache Spark



Apache Spark is a fast, in-memory data processing engine which allows data workers to efficiently execute streaming, machine learning or SQL workloads that require fast iterative access to datasets.

Speed

- Run computations in memory.
- Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.
- 100 times faster in memory and 10 times faster even when running on disk than MapReduce.

Generality

- A general programming model that enables developers to write an application by composing arbitrary operators.
- Spark makes it easy to combine different processing models seamlessly in the same application.
- Example:
 - Data classification through Spark machine learning library.
 - Streaming data through source via Spark Streaming.
 - Querying the resulting data in real time through Spark SQL.

Spark Core

- Spark Core is the underlying general execution engine for the Spark platform, all other functionality is built on top of it.
- Provides distributed task dispatching, scheduling, and basic IO functionalities, exposed through an application programming interface centered on the RDD, which is Spark's primary programming abstraction.



- Spark package designed for working with structured data which is built on top of Spark Core.
- Provides an SQL-like interface for working with structured data.
- More and more Spark workflow is moving towards Spark SQL.



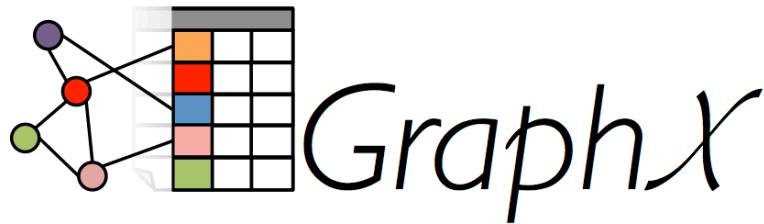
Spark Streaming

- Running on top of Spark, Spark Streaming provides an API for manipulating data streams that closely match the Spark Core's RDD API.
- Enables powerful interactive and analytical applications across both streaming and historical data while inheriting Spark's ease of use and fault tolerance characteristics.



MLlib

- Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms and blazing speed.
- Usable in Java, Scala, and Python as part of Spark applications.
- Consists of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering and dimensionality reduction, etc.



- A graph computation engine built on top of Spark that enables users to interactively create, transform and reason about graph structured data at scale.
- Extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge.

Who use Apache Spark and how?

- Data scientists
- Data processing application engineers

Data Scientists

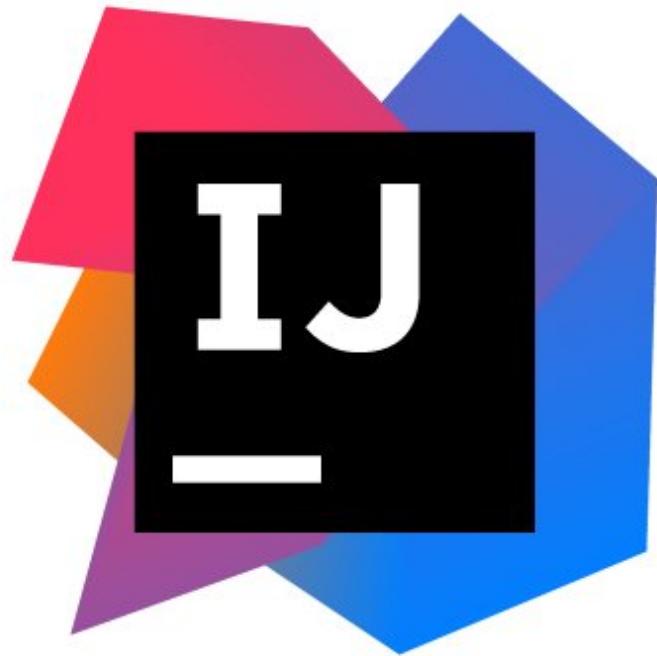
- Identify patterns, trends, risks and opportunities in data.
- Analyze data with the goal of answering a question or discovering insights.
- Ad hoc analysis.
- Spark helps data scientists by supporting the entire data science workflow, from data access, ad-hoc analysis and integration to machine learning and visualization.

Data processing application engineers

- Build applications that lever advanced analytics in partnership with the data scientist.
- General classes of applications are moving to Spark, including compute-intensive applications and applications that require input from data streams such as sensors or social data.
- Spark provides an easy way to parallelize these applications across clusters and hides the complexity of distributed systems programming, network communication, and fault tolerance.

Install GIT

Set up our Spark project with IntelliJ IDEA



Step 1: Generate IDEA configuration files for this Spark project.

Step 2: Download and install IntelliJ IDEA.

Step 3: Set up our Spark project in IntelliJ IDEA.

Step 1: Generate IDEA configuration files for this Spark project.

Step 2: Download and install IntelliJ IDEA.

Step 3: Set up our Spark project in IntelliJ IDEA.

IntelliJ IDEA is the preferred IDE for this course.

If you have a strong preference for Eclipse, you can skip this lecture and follow the next one to set up this project in Eclipse.

Our Spark project is managed by **Gradle**.



Gradle has an IDEA plugin which can generate all the necessary files that are required by IntelliJ IDEA, thus making it possible to open the project directly from IDEA.

The plugin has registered a **Gradle** task called **idea**.

Set up our Spark project with Eclipse

Run our first Spark job

Word Count

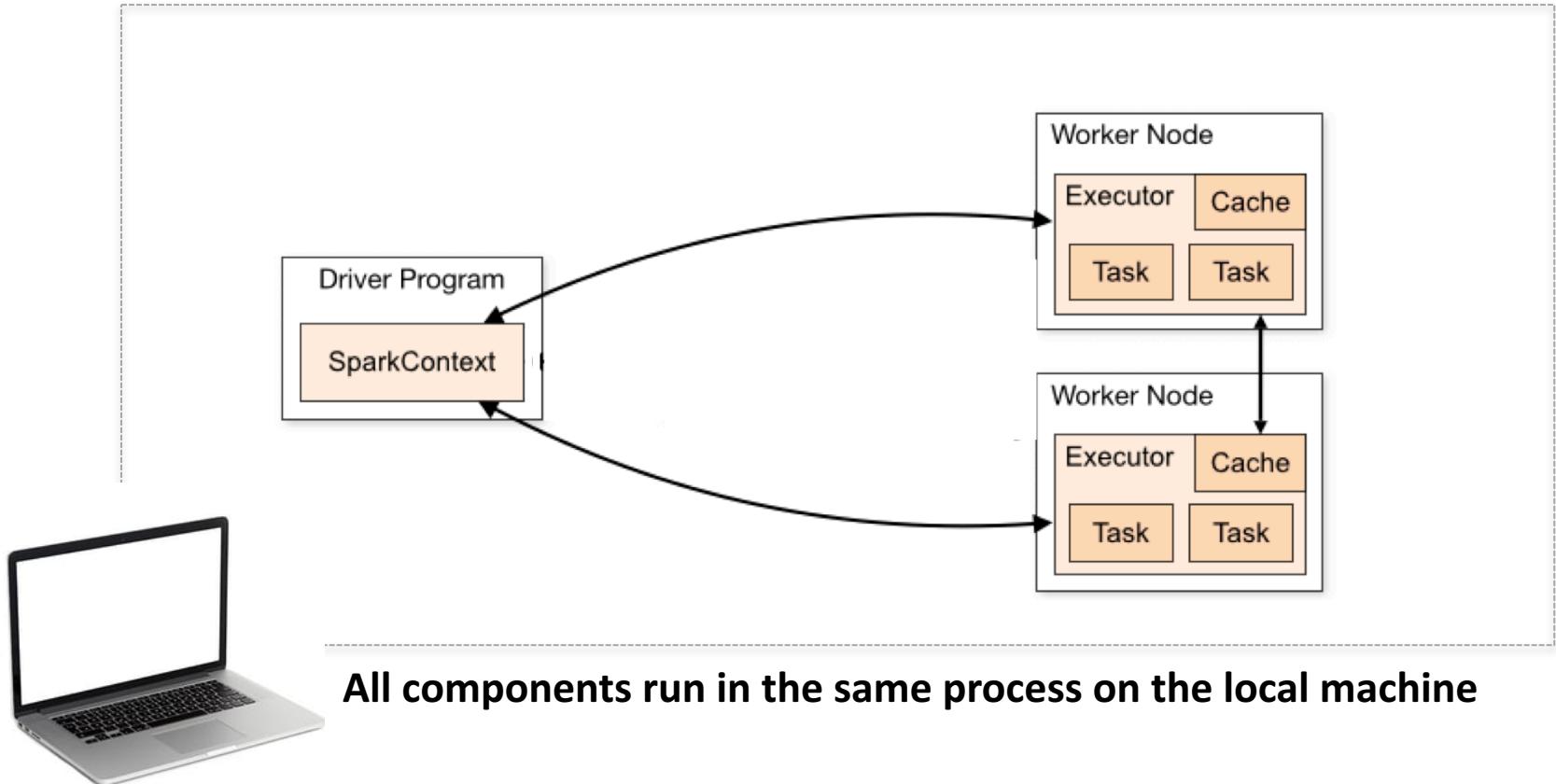
RDD

(Resilient Distributed Datasets)

What is a dataset?

A dataset is basically a collection of data; it can be a list of strings, a list of integers or even a number of rows in a relational database.

Running Spark in the local mode



RDD

- RDDs can contain any types of objects, including user-defined classes.
- An RDD is simply a encapsulation around a very large dataset. In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result.
- Under the hood, Spark will automatically distribute the data contained in RDDs across your cluster and parallelize the operations you perform on them.

How to create a RDD

1. Load RDDs from external storage.

```
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
```

How to create a RDD

2. Take an existing collection in your program and pass it to SparkContext's parallelize method.

```
List<Integer> inputIntegers = Arrays.asList(1, 2, 3, 4, 5);  
JavaRDD<Integer> integerRdd = sc.parallelize(inputIntegers);
```

Important aspects about **RDDs**

RDDs are Distributed

- Each RDD is broken into multiple pieces called partitions, and these partitions are divided across the clusters.
- This partitioning process is done automatically by Spark, so you don't need to worry about all the details that how your data is partitioned across the cluster.

RDDs are **Immutable**

- They cannot be changed after they are created.
- Immutability rules out a significant set of potential problems due to updates from multiple threads at once.
- Immutability makes it safe to share RDDs across processes.

RDDs are **Resilient**

- RDDs are a deterministic function of their input. This plus **immutability** also means the RDDs parts can be recreated at any time.
- In case of any node in the cluster goes down, Spark can recover the parts of the RDDs from the input and pick up from where it left off.
- Spark does the heavy lifting for you to make sure that RDDs are fault tolerant.

What can we do with RDDs?

Transformations

Actions

Transformations

- Apply some functions to the data in RDD to create a new RDD.
- One of the most common transformations is **filter** which will return a new RDD with a subset of the data in the original RDD.

```
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
JavaRDD<String> linesWithFriday = lines.filter(line -> line.contains("Friday"));
```

Actions

- Compute a result based on an RDD.
- One of the most popular Actions is **first**, which returns the first element in an RDD.

```
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
String firstLine = lines.first();
```

Transformations VS Actions

- Transformations and actions are different because of the way how Spark computes RDDs.
- Even though new RDDs can be defined any time, they are only computed by Spark in a **lazy** fashion, which is the **first time** they are used in an **ACTION**.

Lazy evaluation

```
//No thing happens when Spark sees textFile() statement.  
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");  
  
//No thing happens when Spark sees filter() transformation  
JavaRDD<String> linesWithFriday = lines.filter(line -> line.startsWith("Friday"));  
  
//Spark only starts loading the "in/uppercase.text" file when first() action is called on linesWithFriday RDD.  
String firstLineWithFriday = linesWithFriday.first();  
  
// Spark scans the file only until the first line starting with Friday is detected;  
// it doesn't even need to go through the entire file.
```

Lazy Evaluation

- lazy evaluation helps to bring down the number of passes it has to take over our data by grouping operations together.
- Lazy evaluation is done automatically for us by Spark.

Spark RDD general workflow

- Generate initial RDDs from external data.
- Apply transformations.
- Launch actions.

Creating **RDDs**

How to create a RDD

- Take an existing collection in your program and pass it to SparkContext's **parallelize** method.

```
List<Integer> inputIntegers = Arrays.asList(1, 2, 3, 4, 5);  
JavaRDD<Integer> integerRdd = sc.parallelize(inputIntegers);
```

- All the elements in the collection will then be copied to form a distributed dataset that can be operated on in parallel.
- Very Handy to create an RDD with little effort.
- NOT practical working with large datasets.

How to create a RDD

- Load RDDs from external storage by calling **textFile** method on Sparkcontext.

```
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
```

- The external storage is usually a distributed file system such as **Amazon S3** or **HDFS**.
- There are other data sources which can be integrated with Spark and used to create RDDs including JDBC, Cassandra, and Elasticsearch, etc.

Transformations

Transformation

- Transformations are operations on RDDs which will return a new RDD.
- The two most common transformations are **filter** and **map**.

filter() transformation

- Takes in a function and returns an RDD formed by selecting those elements which pass the filter function.
- Can be used to remove some invalid rows to clean up the input RDD or just get a subset of the input RDD based on the filter function.

```
JavaRDD<String> cleanedLines = lines.filter(line -> !line.isEmpty());
```

map() transformation

- Takes in a function and passes each element in the input RDD through the function, with the result of the function being the new value of each element in the resulting RDD.
- It can be used to make HTTP requests to each URL in our input RDD, or it can be used to calculate the square root of each number.

```
JavaRDD<String> URLs = sc.textFile( path: "in/urls.text");
URLs.map(url -> makeHttpRequest(url));
```

- The return type of the map function is not necessarily the same as its input type.

```
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
JavaRDD<Integer> lengths = lines.map(line -> line.length());
```

Solution to Airports by latitude problem

flatMap transformation

flatMap VS map

```
/*
 * Base interface for a map function used in Dataset's map function.
 */
public interface MapFunction<T, U> extends Serializable {
    U call(T value) throws Exception;
}

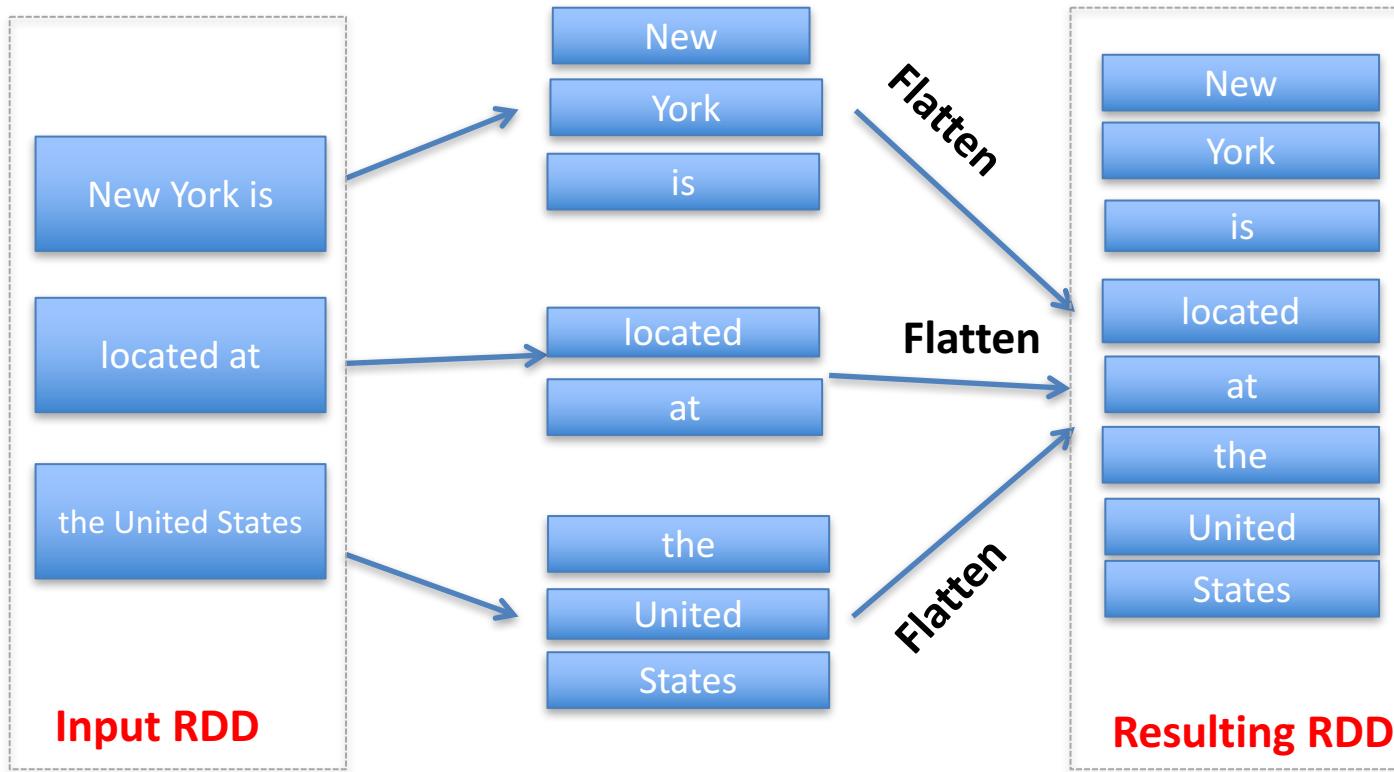
/*
 * A function that returns zero or more output records from each input record.
 */
public interface FlatMapFunction<T, R> extends Serializable {
    Iterator<R> call(T t) throws Exception;
}
```

flatMap VS map

map: 1 to 1 relationship

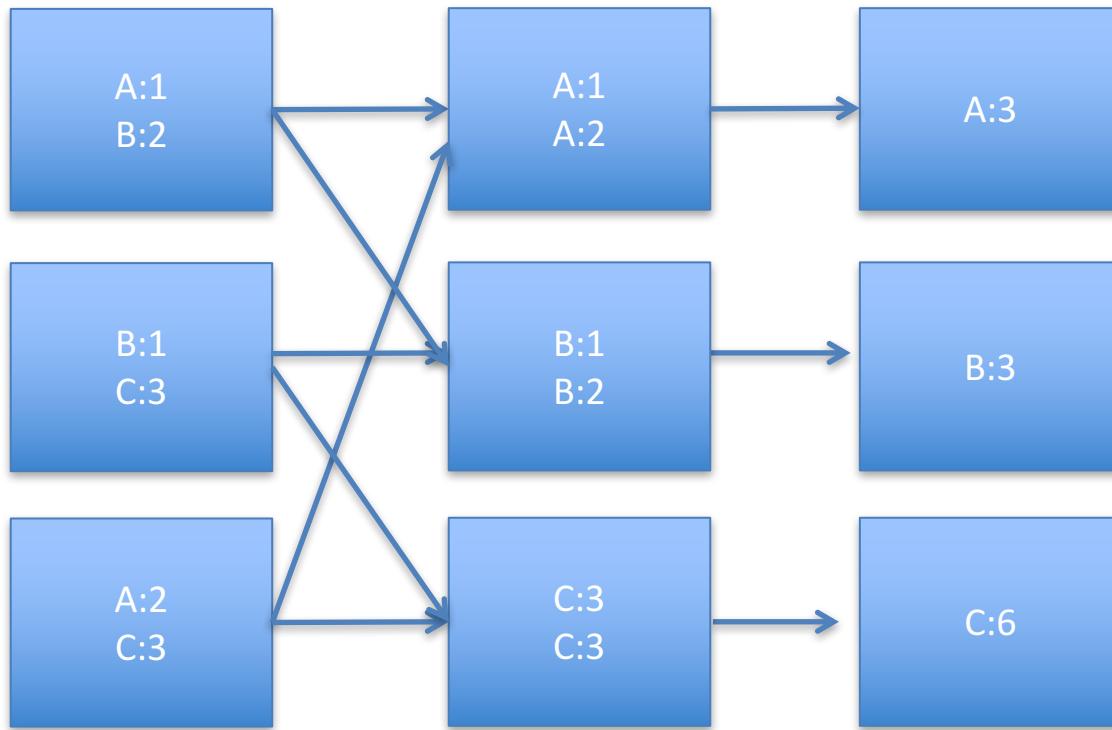
flatMap: 1 to many relationship

flatMap example: split lines by space

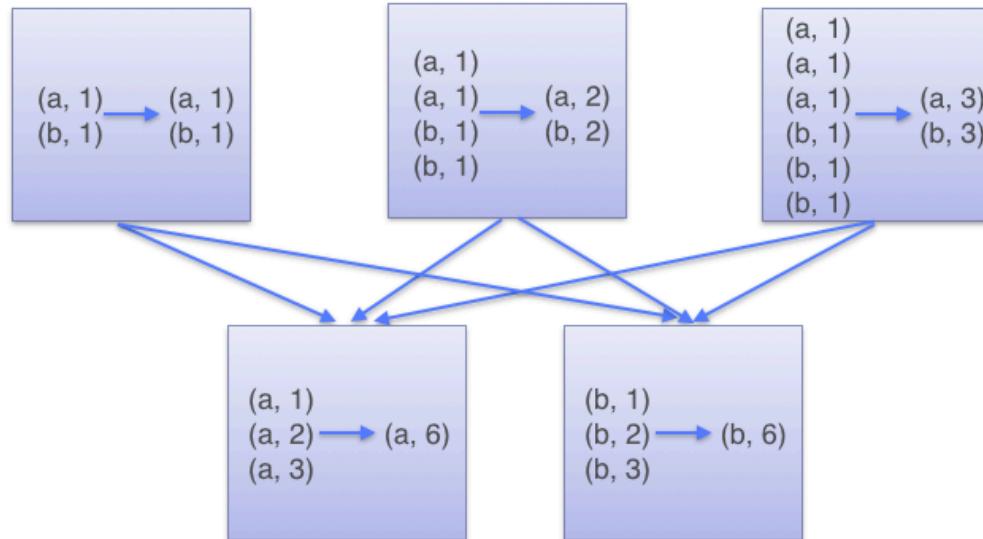


partitionBy

groupByKey



ReduceByKey



flatmap(FlatMapFunction func)

```
/**  
 * A function that returns zero or more output records from each input record.  
 */  
public interface FlatMapFunction<T, R> extends Serializable {  
    Iterator<R> call(T t) throws Exception;  
}
```

```
package org.apache.spark.api.java.function;

import java.io.Serializable;

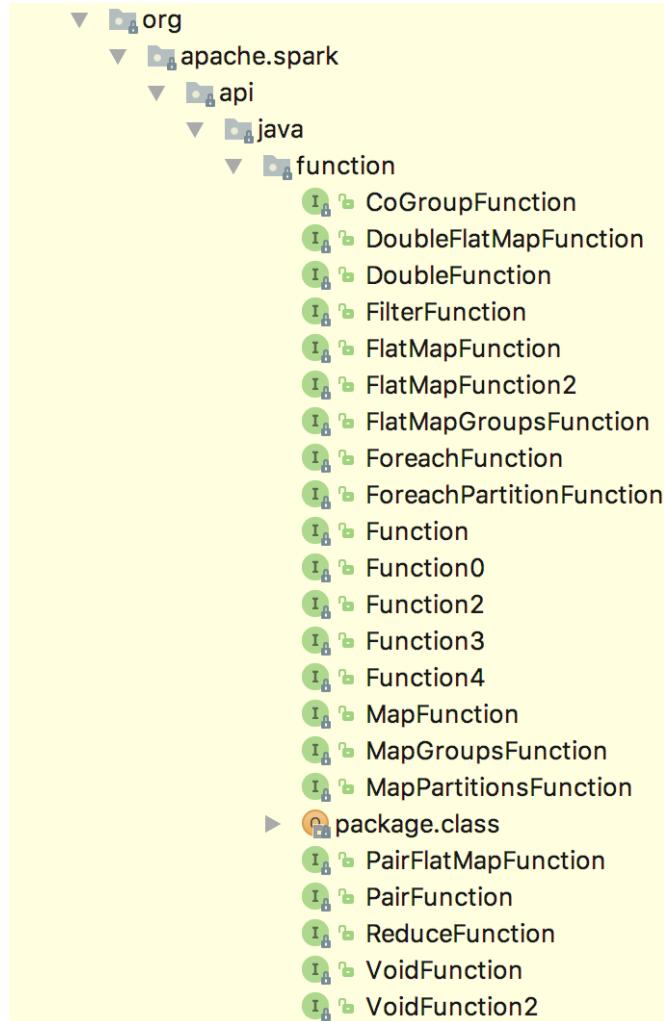
public interface Function2<T1, T2, R> extends Serializable {
    R call(T1 var1, T2 var2) throws Exception;
}
```

```
package org.apache.spark.api.java.function;

import java.io.Serializable;

/**
 * Base interface for functions whose return types do not create special RDDs. PairFunction and
 * DoubleFunction are handled separately, to allow PairRDDs and DoubleRDDs to be constructed
 * when mapping RDDs of other types.
 */
public interface Function<T1, R> extends Serializable {
    R call(T1 v1) throws Exception;
}
```

Passing functions to Spark



```
/*
 * A function that returns zero or more output records from each input record.
 */
public interface FlatMapFunction<T, R> extends Serializable {
    Iterator<R> call(T t) throws Exception;
}
```

```
public class Uppercase {

    public static void main(String[] args) throws Exception {

        SparkConf conf = new SparkConf().setAppName("uppercase").setMaster("local[*]");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");

        // anonymous inner class
        lines.filter(new Function<String, Boolean>() {
            @Override
            public Boolean call(String line) throws Exception {
                return line.startsWith("Friday");
            }
        });

        // named class
        lines.filter(new StartsWithFriday());

        // lambda expression
        lines.filter(line -> line.startsWith("Friday"));
    }

    static class StartsWithFriday implements Function<String, Boolean> {
        public Boolean call( String line){
            return line.startsWith("Friday");
        }
    }
}
```

Set operations

Set operations which are performed on **one** RDD:

- **sample**
- **distinct**

sample

```
public org.apache.spark.api.java.JavaRDD<T> sample(boolean withReplacement, double fraction)
```

- The sample operation will create a random sample from an RDD.
- Useful for testing purpose.

distinct

```
public org.apache.spark.api.java.JavaRDD<T> distinct()
```

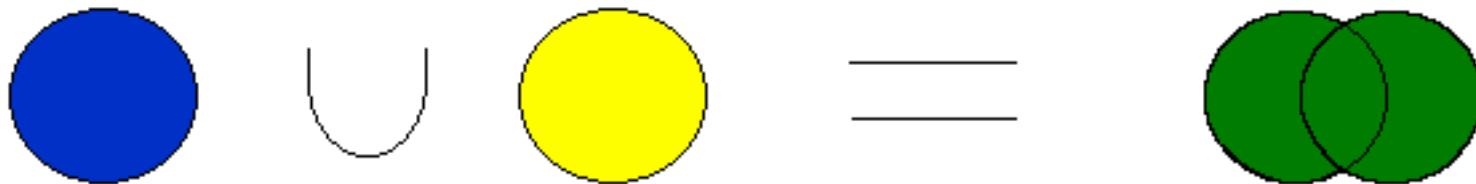
- The distinct transformation returns the distinct rows from the input RDD.
- The distinct transformation is expensive because it requires shuffling all the data across partitions to ensure that we receive only one copy of each element.

LEVEL UP
www.level-up.one

Set operations which are performed on **two** RDDs and produce **one** resulting RDD:

- **union**
- **intersection**
- **subtract**
- **cartesian product**

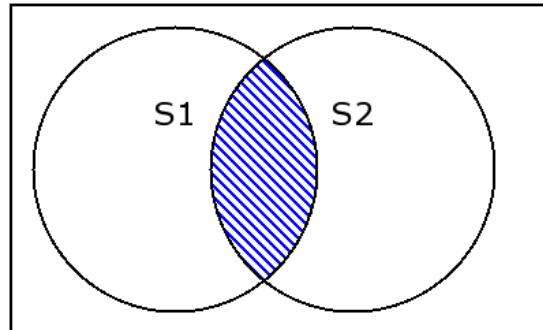
union operation



```
public org.apache.spark.api.java.JavaRDD<T> union(org.apache.spark.api.java.JavaRDD<T> other)
```

- Union operation gives us back an RDD consisting of the data from both input RDDs.
- If there are any duplicates in the input RDDs, the resulting RDD of Spark's union operation will contain duplicates as well.

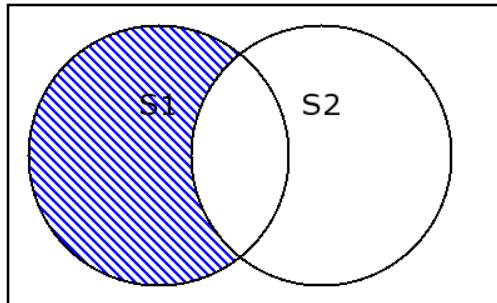
intersection operation



```
public org.apache.spark.api.java.JavaRDD<T> intersection(org.apache.spark.api.java.JavaRDD<T> other)
```

- Intersection operation returns the common elements which appear in both input RDDs.
- Intersection operation removes all duplicates including the duplicates from single RDD before returning the results.
- Intersection operation is quite expensive since it requires shuffling all the data across partitions to identify common elements.

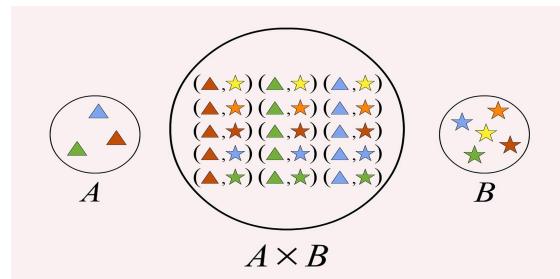
subtract operation



```
public org.apache.spark.api.java.JavaRDD<T> subtract(org.apache.spark.api.java.JavaRDD<T> other)
```

- Subtract operation takes in another RDD as an argument and returns us an RDD that only contains element present in the first RDD and not the second RDD.
- Subtract operation requires a shuffling of all the data which could be quite expensive for large datasets.

cartesian operation



```
public <U> JavaPairRDD<T, U> cartesian(JavaRDDLike<U, ?> other)
```

- Cartesian transformation returns all possible pairs of a and b where a is in the source RDD and b is in the other RDD.
- Cartesian transformation can be very handy if we want to compare the similarity between all possible pairs.

Actions

Actions

- Actions are the operations which will return a final value to the driver program or persist data to an external storage system.
- Actions will force the evaluation of the transformations required for the RDD they were called on.

Common Actions in Spark

- **collect**
- count
- countByValue
- take
- saveAsTextFile
- reduce

collect

- Collect operation retrieves the entire RDD and returns it to the driver program in the form of a regular collection or value.
- If you have a String RDD, when you call collect action on it, you would get a list of strings.
- This is quite useful if your spark program has filtered RDDs down to a relatively small size and you want to deal with it **locally**.
- The entire dataset must fit in memory on a single machine as it all needs to be copied to the driver when the **collect** action is called. So collect action should **NOT** be used on large datasets.
- Collect operation is widely used in unit tests, to compare the value of our RDD with our expected result, as long as the entire contents of the RDD can fit in memory.

Common Actions in Spark

- collect
- count
- countByValue
- take
- saveAsTextFile
- reduce

count and countbyValue

- If you just want to count how many rows in an RDD, **count** operation is a quick way to do that. It would return the count of the elements.
- **countbyValue** will look at unique values in the each row of your RDD and return a map of each unique value to its count.
- **countbyValue** is useful when your RDD contains duplicate rows, and you want to count how many of each unique row value you have.

Common Actions in Spark

- collect
- count
- countByValue
- take
- **saveAsTextFile**
- reduce

take

```
List<String> words = wordRdd.take(n);
```

- **take** action takes **n** elements from an RDD.
- **take** operation can be very useful if you would like to take a peek at the RDD for unit tests and quick debugging.
- **take** will return n elements from the RDD and it will try to reduce the number of partitions it accesses, so it is possible that the take operation could end up giving us back a **biased** collection, and it doesn't necessary return the elements in the order we might expect.

take

```
List<String> words = wordRdd.take(n);
```

- **take** action takes **n** elements from an RDD.
- **take** operation can be very useful if you would like to take a peek at the RDD for unit tests and quick debugging.
- **take** will return n elements from the RDD and it will try to reduce the number of partitions it accesses, so it is possible that the take operation could end up giving us back a **biased** collection, and it doesn't necessary return the elements in the order we might expect.

saveAsTextFile

```
airportsNameAndCityRdd.saveAsTextFile( path: "out/airports.text");
```

saveAsTextFile can be used to write data out to a distributed storage system such as HDFS or Amazon S3, or even local file system.

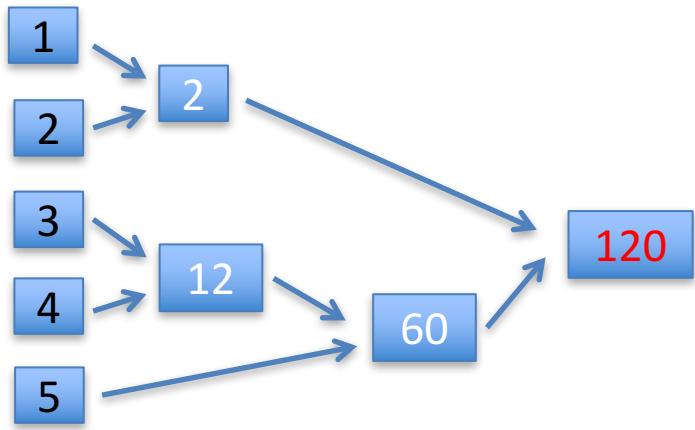
Common Actions in Spark

- collect
- count
- countByValue
- take
- saveAsTextFile
- **reduce**

reduce

```
/**  
 * A two-argument function that takes arguments of type T1 and T2 and returns an R.  
 */  
public interface Function2<T1, T2, R> extends Serializable {  
    R call(T1 v1, T2 v2) throws Exception;  
}  
  
Integer product = integerRdd.reduce((x, y) -> x * y);
```

- **reduce** takes a function that operates on **two** elements of the type in the input RDD and returns **a** new element of the same type. It reduces the elements of this RDD using the specified binary function.
- This function produces the same result when **repetitively** applied on the same set of RDD data, and reduces to a single value.
- With **reduce** operation, we can perform different types of aggregations.



Summary of RDD Operations

- **Transformations** are operations on RDDs that return a new RDD, such as map and filter.
- **Actions** are operations that return a result to the driver program or write it to storage, and kick off a computation, such as count and collect.

Lazy Evaluation

- Transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.
- Rather than thinking of an RDD as containing specific data, it might be better to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations.
- Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together.

Transformations return RDDs, whereas **actions** return some other data type.

Transformations:

```
JavaRDD<String> linesWithFriday = lines.filter(line -> line.contains("Friday"));  
JavaRDD<Integer> lengths = lines.map(line -> line.length());
```

Actions:

```
List<String> words = wordRdd.collect();  
String firstLine = lines.first();
```

Caching and Persistence

Persistence

- Sometimes we would like to call actions on the same RDD multiple times.
 - If we do this naively, RDDs and all of its dependencies are recomputed are recomputed each time an action is called on the RDD.
 - This can be very expensive, especially for some iterative algorithms, which would call actions on the same dataset many times.
-
- If you want to reuse an RDD in multiple actions, you can also ask Spark to persist by calling the `persist()` method on the RDD.
 - When you persist an RDD, the first time it is computed in an action, it will be kept in memory across the nodes.

Different Storage Level

RDD.persist(StorageLevel level)

RDD.cache() = RDD.persist (MEMORY_ONLY)

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as <i>serialized</i> Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer , but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.

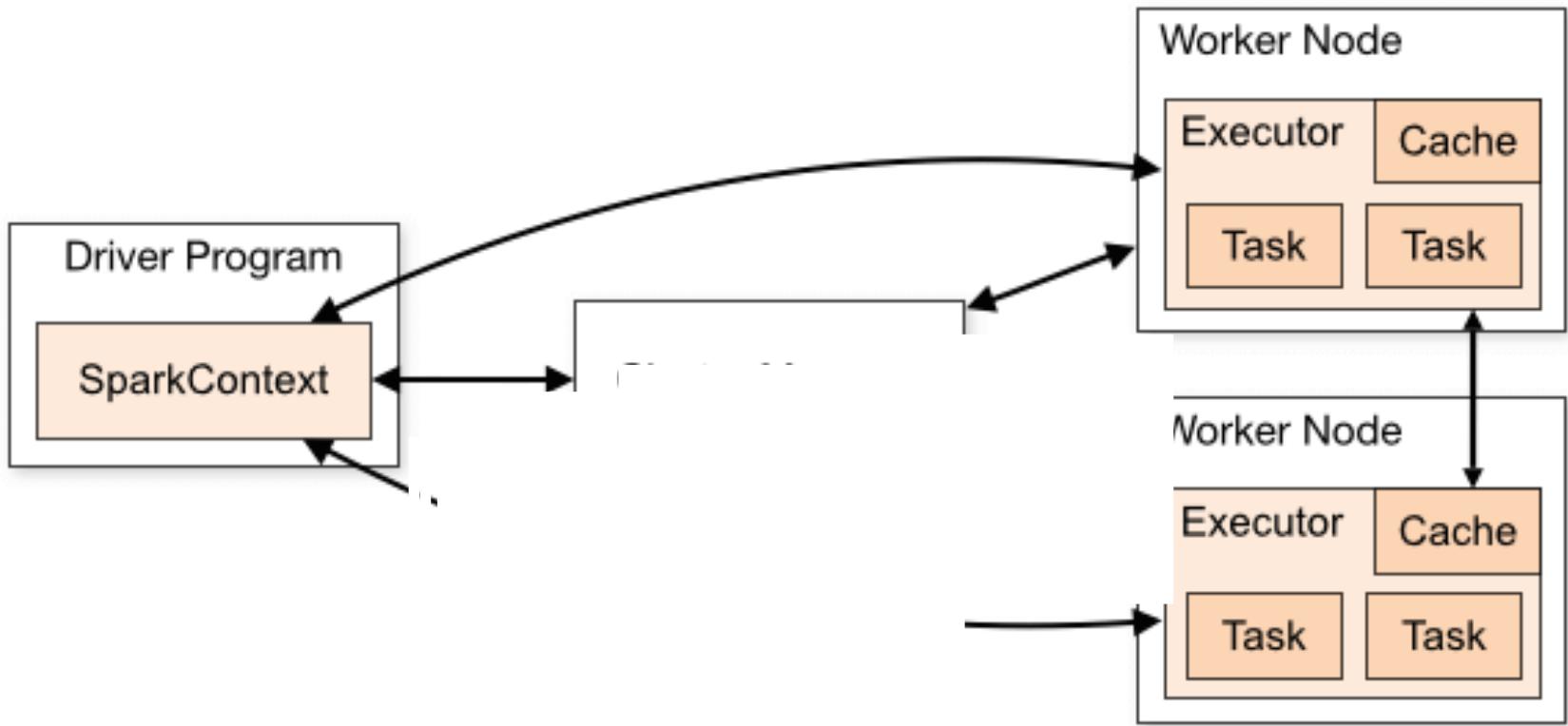
Which Storage Level we should choose?

- Spark's storage levels are meant to provide different trade-offs between memory usage and CPU efficiency.
- If the RDDs can fit comfortably with the default storage level, **MEMORY_ONLY** is the ideal option. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
- If not, try using **MEMORY_ONLY_SER** to make the objects much more space-efficient, but still reasonably fast to access.
- Don't save to disk unless the functions that computed your datasets are expensive, or they filter a significant amount of the data.

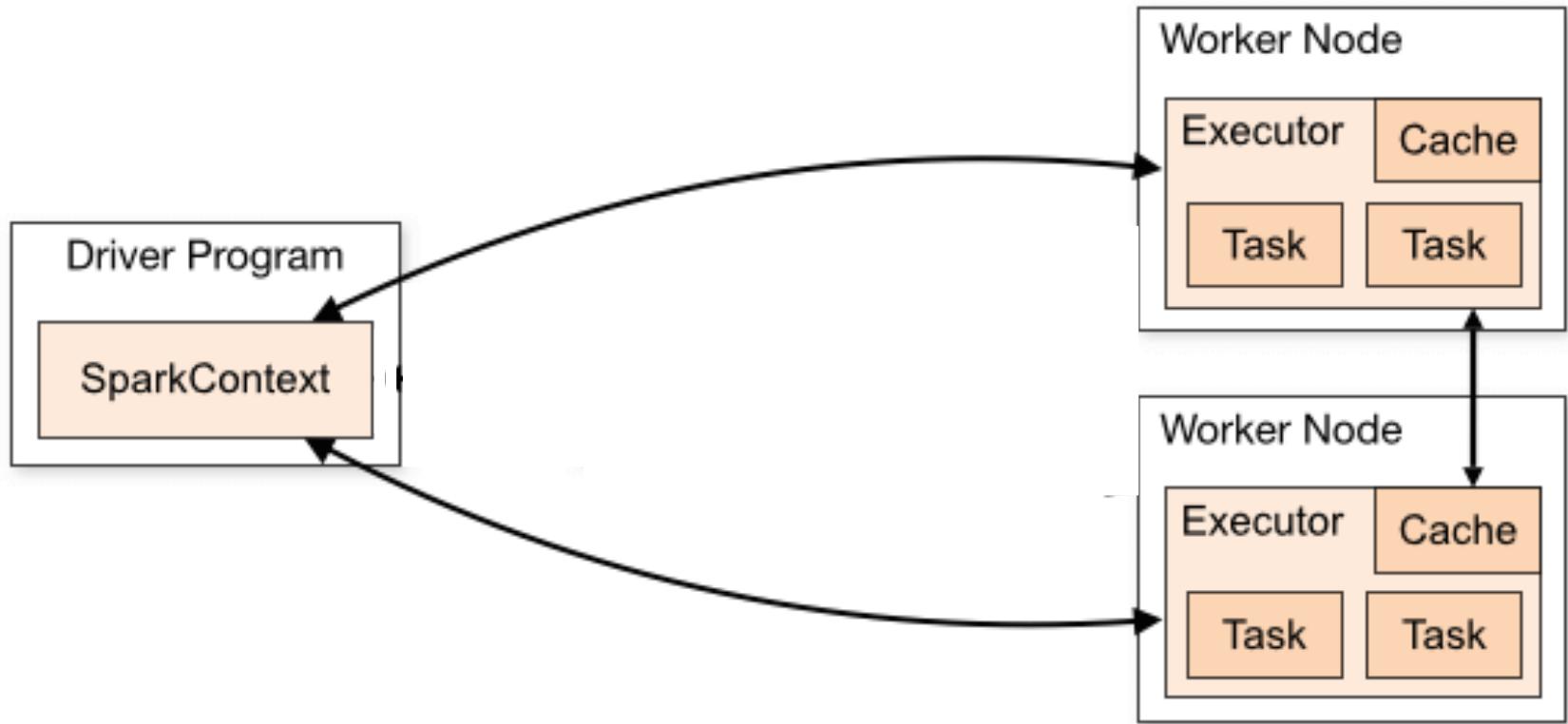
- **What would happen If you attempt to cache too much data to fit in memory?**
 - Spark will evict old partitions automatically using a Least Recently Used cache policy.
 - For the **MEMORY_ONLY** storage level, spark will re-compute these partitions the next time they are needed.
 - For the **MEMORY_AND_DISK** storage level, Spark will write these partitions to disk.
 - In either case, your spark job won't break even if you ask Spark to cache too much data.
 - Caching unnecessary data can cause spark to evict useful data and lead to longer re-computation time.

Sample Solution for
the **Sum of Numbers** problem

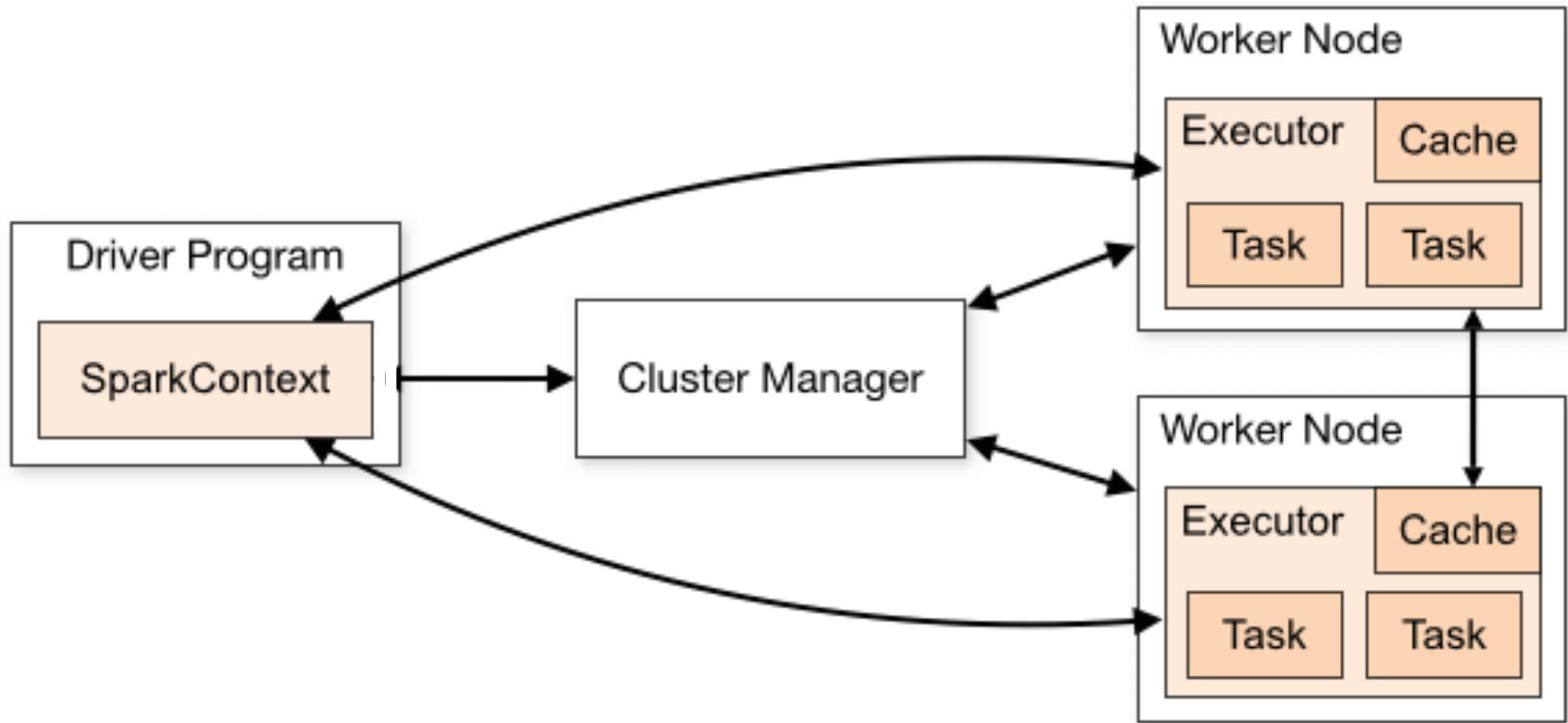
Spark Architecture

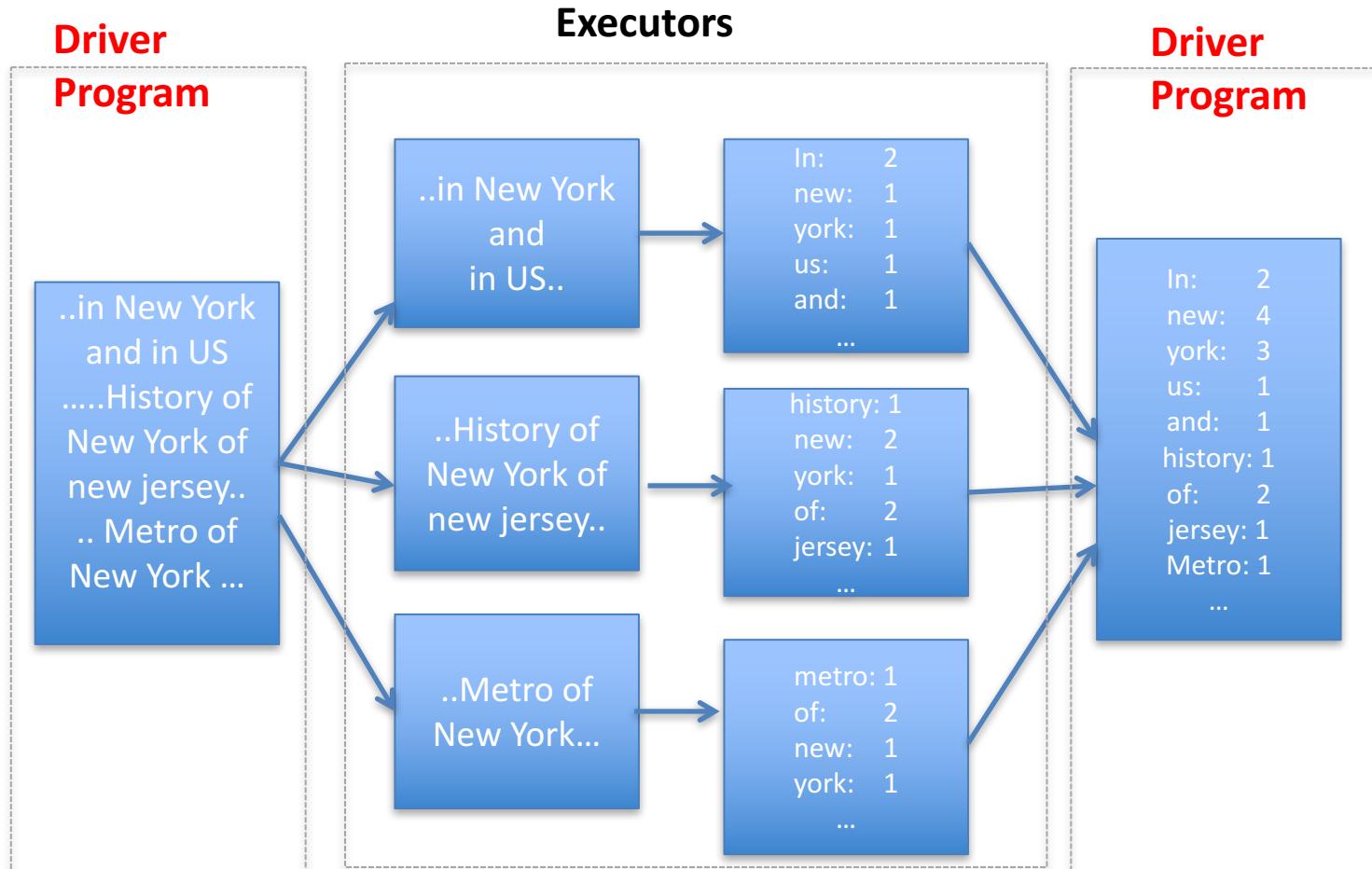


Running Spark in the cluster mode

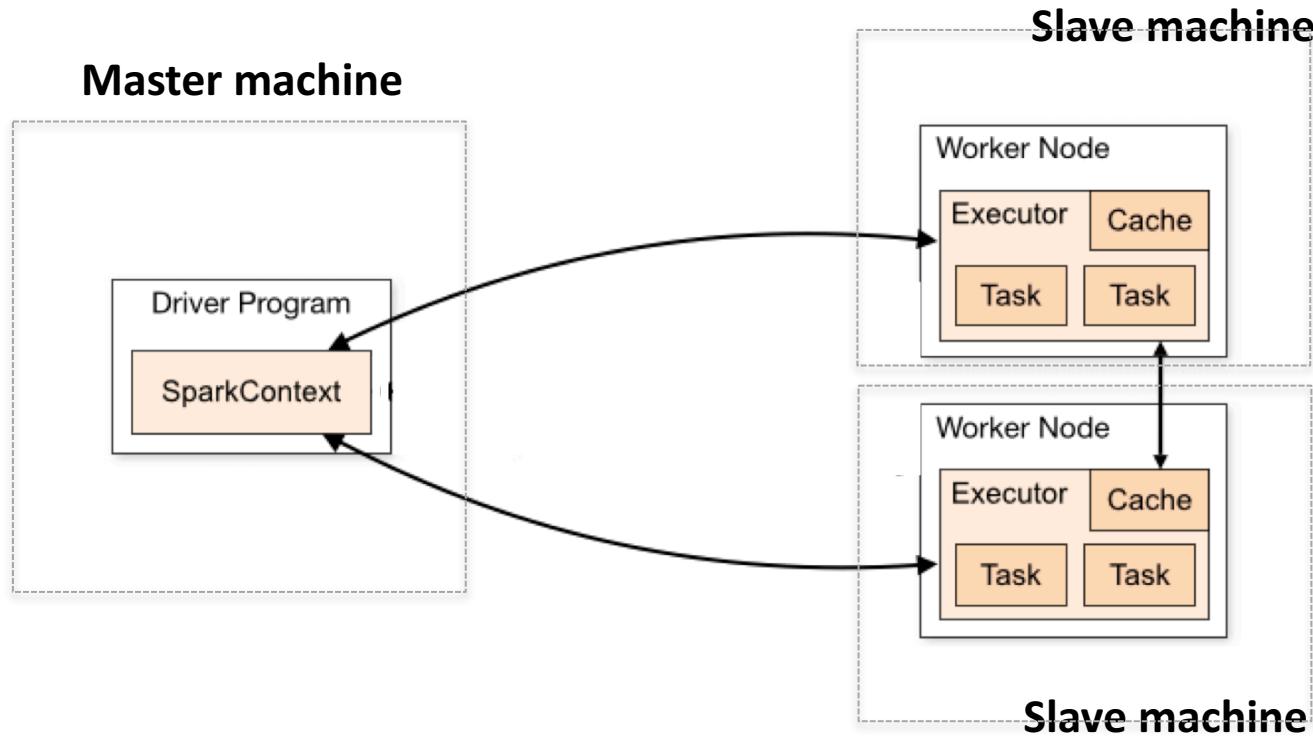


Running Spark in the cluster mode

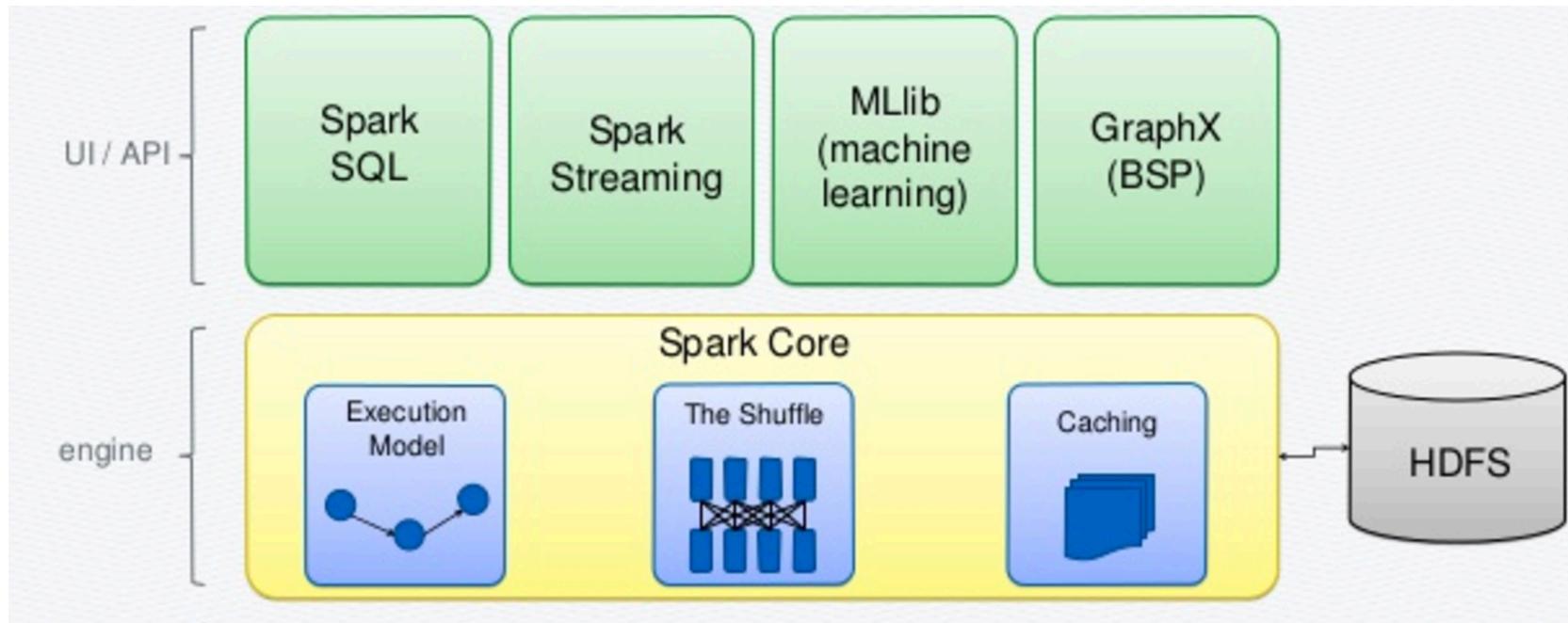




Running Spark in the cluster mode



Spark Components



Introduction to **Pair RDDs**

Pair RDD

- A lot of datasets we see in real life examples are usually **key value pairs**.
- Examples:
 - A dataset which contains passport IDs and the names of the passport holders.
 - A dataset contains course names and a list of students that enrolled in the courses.
- The typical pattern of this kind of dataset is that each row is one key maps to one value or multiple values.
- Spark provides a data structure called **Pair RDD** instead of regular RDDs, which makes working with this kind of data more simpler and more efficient.
- A Pair RDDs is a particular type of RDD that can store key-value pairs.
- Pair RDDs are useful building blocks in many spark programs.

What is Pair RDD

- A Pair RDDs is a particular type of RDD that can store key-value pairs.

Create Pair RDDs

How to create **Pair RDDs**

1. Return Pair RDDs from a list of key value data structure called tuple.
2. Turn a regular RDD into a Pair RDD.

Get Pair RDDs from **tuples**

Java doesn't have a built-in tuple type, so Spark's Java API allow users to create tuples using the *scala.Tuple2* class.

```
Tuple2<Integer, String> tuple = new Tuple2<>(_1: 12, _2: "value");
Integer key = tuple._1();
String value = tuple._2();

public <K, V> JavaPairRDD<K, V> parallelizePairs(List<Tuple2<K, V>> list)
```

```
 /**
 * A function that returns key-value pairs (Tuple2<K, V>), and can be used to
 * construct PairRDDs.
 */
public interface PairFunction<T, K, V> extends Serializable {
    Tuple2<K, V> call(T t) throws Exception;
}
```

Transformations on Pair RDDs

Transformations on Pair RDD

- Pair RDDs are allowed to use all the transformations available to regular RDDs, and thus support the same functions as regular RDDs.
- Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements.

filter transformation

- This **filter** transformation that can be applied to a regular RDD can also be applied to a Pair RDD.
- The **filter** transformation takes in a function and returns a Pair RDD formed by selecting those elements which pass the **filter** function.

map and mapValues transformations

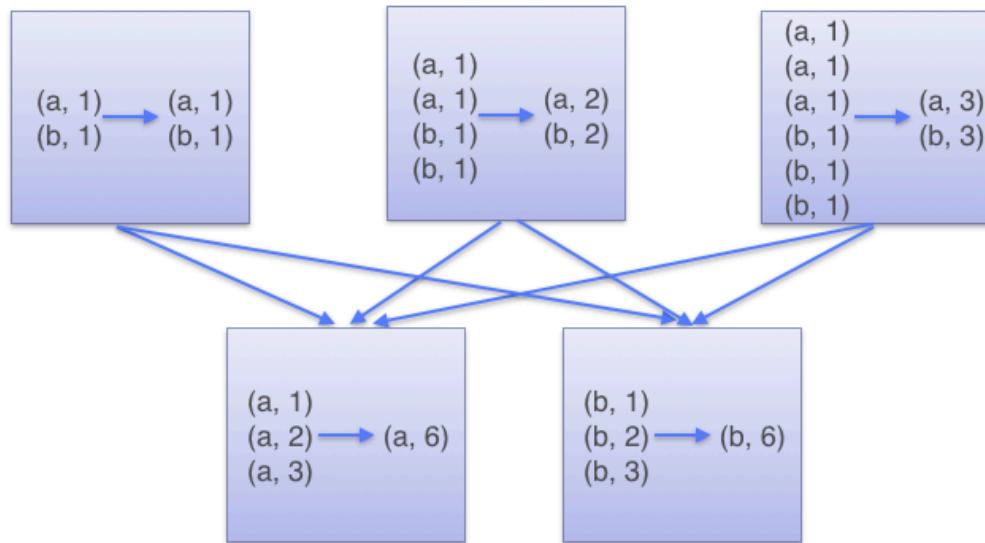
- The `map` transformation also works for Pair RDDs. It can be used to convert an RDD to another one.
- But most of the time, when working with pair RDDs, we don't want to modify the keys, we just want to access the value part of our Pair RDD.
- Since this is a typical pattern, Spark provides the `mapValues` function. The `mapValues` function will be applied to each key value pair and will convert the values based on `mapValues` function, but it will not change the keys.

reduceByKey Aggregations

Aggregation

- When our dataset is described in the format of key-value pairs, it is quite common that we would like to aggregate statistics across all elements with the same key.
- We have looked at the `reduce` actions on regular RDDs, and there is a similar operation for pair RDD, it is called `reduceByKey`.
- `reduceByKey` runs several parallels `reduce` operations, one for each key in the dataset, where each operation combines values that have the same key.
- Considering input datasets could have a huge number of keys, `reduceByKey` operation is not implemented as an action that returns a value to the driver program. Instead, it returns a new RDD consisting of each key and the reduced value for that key.

ReduceByKey



Recent property listings in San Luis Obispo, California

1st column: unique ID for the house

2nd column: location of the property

3rd column: the price of the property in US dollars

4th column: the number of bedrooms of the property

5th column: the number of bathrooms of the property

6th column: the size of the house in square feet

7th column: the price of the house per square foot

8th column: the state of sale

Sample Solution for the **Average House** problem

Task: compute the average price for houses with different number of bedrooms

This could be converted to an aggregation type problem of Pair RDD.

Average price for different type of houses problem:

- key: the number of bedrooms
- value: the average price of the property

Word count problem:

- key: word
- value: the count of each word

groupByKey

groupByKey

- A common use case for Pair RDD is grouping our data by key. For example, viewing all of an account's transactions together.
- If our data is already keyed in the way we want, **groupByKey** will group our data using the key in our Pair RDD.
- Let's say, the input pair RDD has **keys** of **type K** and **values** of **type V**, if we call group by key on the RDD, we get back a Pair RDD of **type K**, and **Iterable V**.

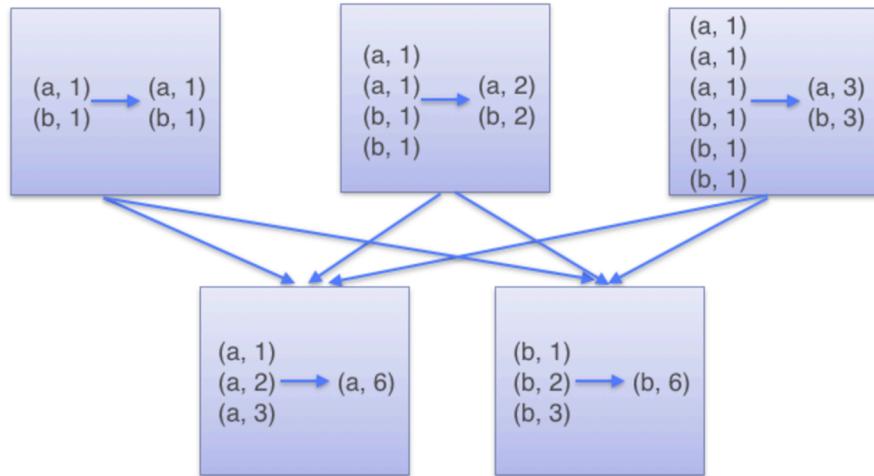
```
public JavaPairRDD<K, Iterable<V>> groupByKey()
```

`groupByKey + [reduce, map, mapValues]`

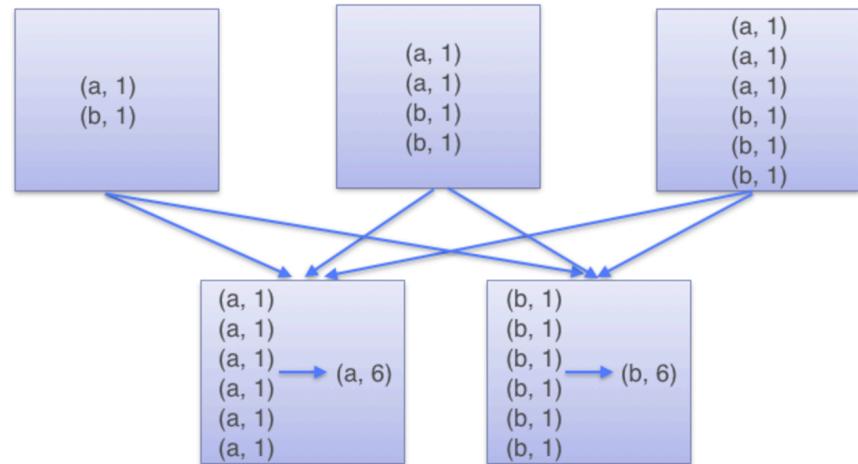
can be replaced by one of the per-key aggregation functions such as **reduceByKey**

ReduceByKey(preferred) VS GroupByKey

ReduceByKey



GroupByKey



sortByKey Transformation

Sort Pair RDD

- We can sort a Pair RDD as long as there is an ordering defined on the key.
- Once we have sorted our Pair RDD, any subsequent call on the sorted Pair RDD to collect or save will return us ordered data.

sortByKey in reverse order

```
public org.apache.spark.api.java.JavaPairRDD<K,V> sortByKey(boolean ascending)  
wordsPairRdd.sortByKey(true);
```

custom comparator

```
public org.apache.spark.api.java.JavaPairRDD<K,V> sortByKey(java.util.Comparator<K> comp)  
countPairRdd.sortByKey((a, b) -> Math.abs(a) - Math.abs(b));
```

- What if we want to **sort** the resulting RDD by the **number of bedrooms** so that we can see exactly how the price changes while the **number of bedrooms** increases?

Sample Solution for the
Sorted Word Count Problem

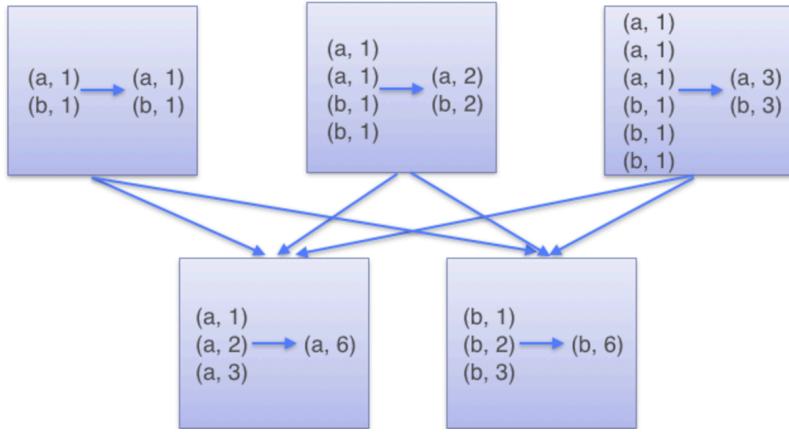
- Sorted Number of Bedrooms problem:
 - The **number of bedrooms** is the **key**
 - Can call **sortByKey** on the Pair RDD
- Sorted Word Count problem:
 - The **count** for each word is the **value** not the **key** of the Pair RDD
 - We can't use **sortByKey** directly on the count

Solution:

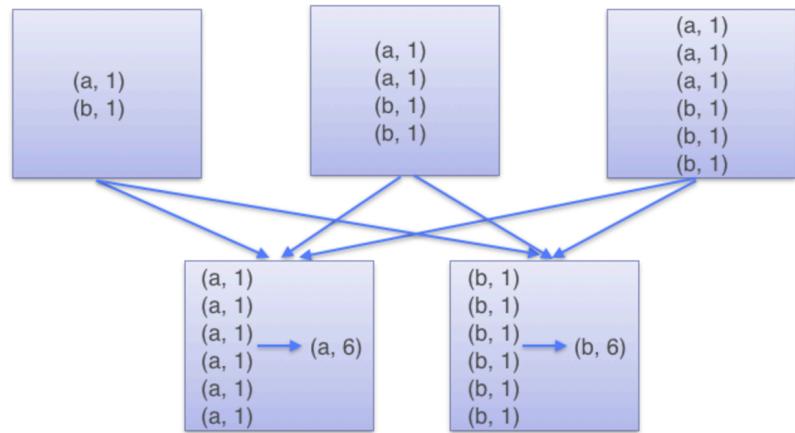
1. Flip the key value of the word count RDD to create a new Pair RDD with the **key** being the **count** and the **value** being the **word**.
2. Do **sortByKey** on the intermediate RDD to sort the count.
3. Flip back the Pair RDD again, with the **key** back to the **word** and the **value** back to the **count**.

Data Partitioning

ReduceByKey



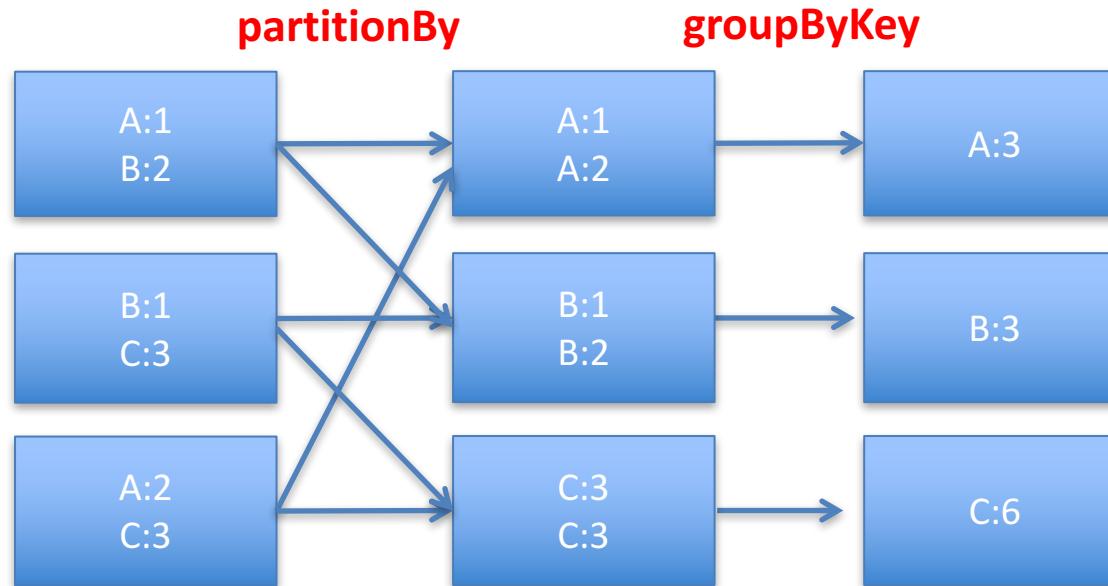
GroupByKey



- In general, we should avoid using `groupByKey` as much as possible.

reduce the amount of shuffle for `groupByKey`

```
val partitionedWordPairRdd = wordPairRdd.partitionBy(new HashPartitioner(4))
partitionedWordPairRdd.persist(StorageLevel.DISK_ONLY)
partitionedWordPairRdd.groupByKey().collect()
```



Operations which would benefit from partitioning

- Join
- leftOuterJoin
- rightOuterJoin
- groupByKey
- reduceByKey
- combineByKey
- lookup.

How `reduceByKey` benefits from partitioning

- Running `reduceByKey` on a pre-partitioned RDD will cause all the values for each key to be computed locally on a single machine, requiring only the final, locally reduced value to be sent from each worker node back to the master.

Operations which would be affected by partitioning

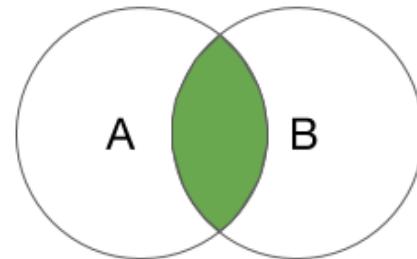
- Operations like `map` could cause the new RDD to forget the parent's partitioning information, as such operations could, in theory, change the key of each element in the RDD.
- General guidance is to prefer `mapValues` over `map` operation.

Join Operation

Join Operations

- Join operation allows us to join two RDDs together which is probably one of the most common operations on a Pair RDD.
- Joins types: `leftOuterJoin`, `rightOuterJoin`, `crossJoin`, `innerJoin`, etc.

Inner Join



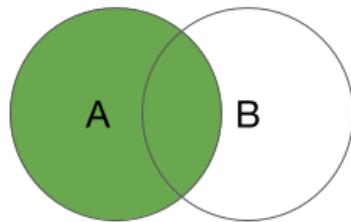
INNER JOIN

```
def join[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))]
```

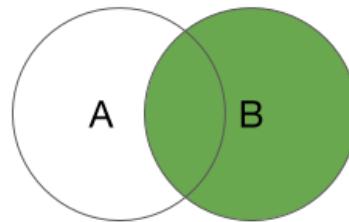
When there are multiple values for the same key in one of the inputs, the resulting pair RDD will have an entry for every possible pair of values with that key from the two input RDDs.

- However sometimes we want the keys in our result as long as they appear in one of the RDD. For instance, if we were joining customer information with feedbacks, we might not want to drop customers if there were not any feedbacks yet.

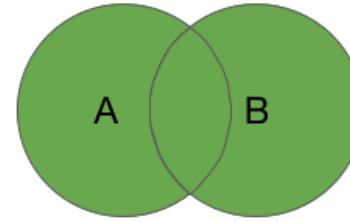
Outer Joins



LEFT OUTER JOIN



RIGHT OUTER JOIN



FULL OUTER JOIN

In this case, we need an outer joins. `leftOuterJoin`, `rightOuterJoin`, and `outerJoin` join Pair RDDs together by key, where one of the Pair RDDs can be missing the key.

The resulting RDD has entries for each key in the source RDDs. The value associated with each key in the resulting RDD is a tuple of the value from the source RDD and an Optional for the value from the other pair RDD.

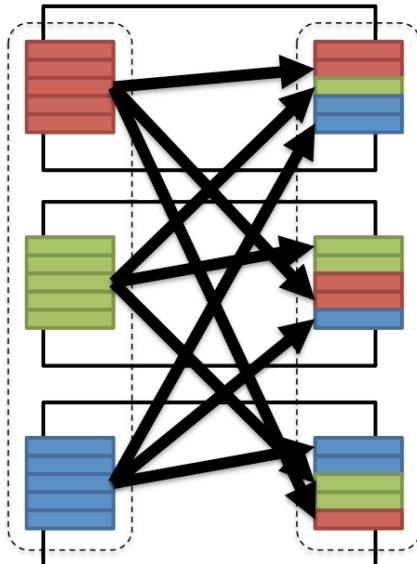
```
public <W> JavaPairRDD<K, scala.Tuple2<V, com.google.common.base.Optional<W>>> leftOuterJoin(JavaPairRDD<K, W> other)
```

Best Practices

- If both RDDs have duplicate keys, join operation can dramatically expand the size of the data. It's recommended to perform a **distinct** or **combineByKey** operation to reduce the key space if possible.
- Join operation may require large network transfers or even create data sets beyond our capability to handle.
- Joins, in general, are expensive since they require that corresponding keys from each RDD are located at the same partition so that they can be combined locally. If the RDDs do not have known partitioners, they will need to be shuffled so that both RDDs share a partitioner and data with the same keys lives in the same partitions.

Shuffled Hash Join

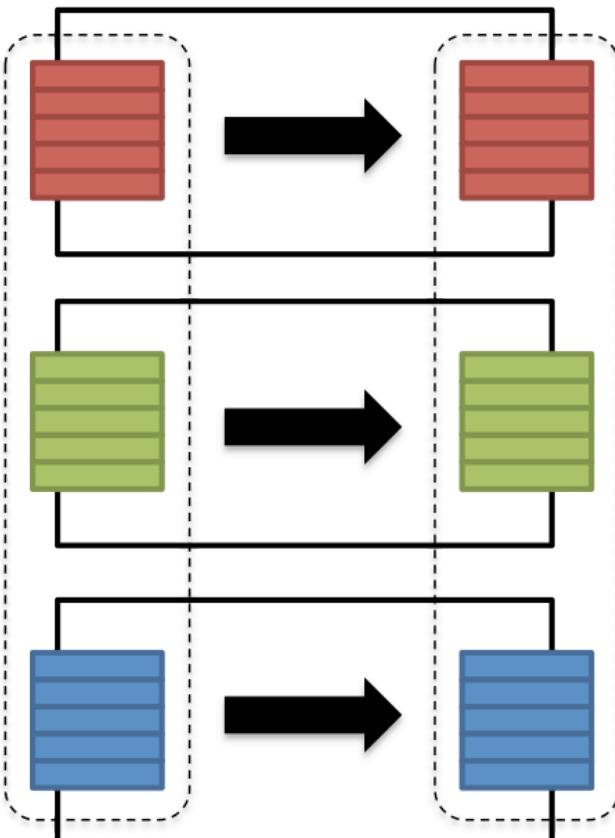
- Input from other partitions are required
- Data shuffling is needed before processing



- To join data, Spark needs the data that is to be joined to live on the same partition.
- The default implementation of join in Spark is a shuffled hash join.
- The shuffled hash join ensures that data on each partition will contain the same keys by partitioning the second dataset with the same default partitioner as the first so that the keys with the same hash value from both datasets are in the same partition.
- While this approach always works, it can be more expensive than necessary because it requires a shuffle.

Avoid Shuffle

- No data movement is needed



- The shuffle can be avoided if Both RDDs have a known **partitioner**.
- If they have the same **partitioner**, the data may be colocated; it can prevent network transfer. So it is recommended to call **partitionby** on the two join RDD with the same **partitioner** before joining them.

```
val partitioner = new HashPartitioner(20)
ages.partitionBy(partitioner)
addresses.partitionBy(partitioner)
```

Accumulators

- Accumulators are variables that are used for aggregating information across the executors. For example, we can calculate how many records are corrupted or count events that occur during job execution for debugging purposes.

Questions we want to answer

- How many records do we have in this survey result?
- How many records are missing the salary middle point?
- How many records are from Canada?

Accumulators

- Tasks on worker nodes cannot access the accumulator's value.
- Accumulators are **write-only** variables.
- This allows accumulators to be implemented efficiently, without having to communicate every update.

Alternative to Accumulators

- Using accumulator is not the only solution to solve this problem.
- It is possible to aggregate values from an entire RDD back to the driver program using actions like `reduce` or `reduceByKey`.
- But sometimes we prefer a simple way to aggregate values that, in the process of transforming a RDD, are generated at a different scale or granularity than that of the RDD itself.

Different Accumulator Types

- Out of the box, Spark supports several accumulators of types such as **Double** and **Long**.
- Spark also allows users to define custom accumulator types and custom aggregation operations such as finding the maximum of the accumulated values instead of adding them.
- You will need to extend the **AccumulatorV2** abstract class to define your custom accumulators.

Task

Besides the `total` and `missingSalaryMidPoint` accumulators, add another accumulator to calculate the number of bytes processed.

Solution to StackOverflow Survey Follow-up Problem:

add another **accumulator** to calculate the bytes processed

Broadcast Variables

Broadcast Variables

- Broadcast variables allow the programmer to keep a **read-only** variable cached on each machine rather than shipping a copy of it with tasks.
- They can be used, for example, to give every node, a copy of a large input dataset, in an efficient manner.
- All broadcast variables will be kept at all the worker nodes for use in one or more Spark operations.

How are those maker spaces distributed across different regions in the UK?

Solution

1. load the **postcode** dataset and broadcast it across the cluster.
2. load the **maker space** dataset and call map operation on the **maker space** RDD to look up the region using the **postcode** of the maker space.

UK Postcode

- The postcode in the **maker space** RDD is the **full postcode**,
 - W1T 3AC
- The postcode in the **postcode** dataset is only the **prefix of the postcode**.
 - W1T
- **Join condition:**
 - If the postcode column in the **maker space** data source starts with the postcode column in the **postcode** data source.

- Join condition:
If the postcode column in the maker space data source starts with the postcode column in the postcode data source

Procedures of using Broadcast Variables

- Create a Broadcast variable `T` by calling `SparkContext.broadcast()` on an object of type `T`.
 - The Broadcast variable can be any type as long as it's serializable because the broadcast needs to be passed from the driver program to all the workers in the Spark cluster across the wire.
 - You can also broadcast a custom Java object, just make sure the object implements the `Serializable` interface.
- The variable will be sent to each node only once and should be treated as `read-only`, meaning updates will not be propagated to other nodes.
- The value of the broadcast can be accessed by calling the `value` method in each node.

Problems if NOT using broadcast variables

- Spark automatically sends all variables referenced in our closures to the worker nodes. While this is convenient, it can also be inefficient because the default task launching mechanism is optimized for small task sizes.
- We can potentially use the same variable in multiple parallel operations, but Spark will send it separately for each operation.
 - This can lead to some performance issue if the size of data to transfer is significant.
 - If we use the same postcode map later, the same postcode map would be sent again to each node.



Spark SQL in Action

Group by Salary Bucket

Salary Middle Point Range	Number of Developers
0 – 20,000	10
20,000 – 40,000	29
40,000 – 60,000	12

Salary Middle Point	Bucket Column Value
51,000	40,000

$$51,000 / 20,000 = 2.55$$

$$\text{Int}(2.55) = 2$$

$$2 * 20,000 = 40,000$$

Catalyst Optimizer

- Spark SQL uses an optimizer called Catalyst to optimize all the queries written both in Spark SQL and DataFrame DSL.
- This optimizer makes queries run much faster than their RDD counterparts.
- The Catalyst is a modular library which is built as a rule-based system. Each rule in the framework focuses on the specific optimization. For example, rule like **ConstantFolding** focuses on removing constant expression from the query.

combineByKey

Package Dependencies

- If your program imports any libraries that are not in the `org.apache.spark` package or part of the language library, you need to ensure that all your dependencies are present at the runtime of your Spark application.
- When you submit an application to Spark, it must ship with its **entire transitive dependency graph** to the cluster.
- It's common practice to rely on a build tool to produce a single large JAR containing the entire transitive dependency graph of an application.

Running Spark Applications on a Cluster

- The user submits an application using `spark-submit`.
- `spark-submit` launches the driver program and invokes the main method specified by the user.
- The driver program contacts the cluster manager to ask for resources to start executors.
- The cluster manager launches executors on behalf of the driver program.
- The driver process runs through the user application. Based on the RDD or dataset operations in the program, the driver sends work to executors in the form of tasks.
- Tasks are run on executor processes to compute and save results.
- If the driver's main method exits or it calls `SparkContext.stop()`, it will terminate the executors.

spark-submit options

```
./bin/spark-submit \  
  --executor-memory 20G \  
  --total-executor-cores 100 \  
  /path/to/examples.jar
```

Benefits of spark-submit

- We can run Spark applications from a command line or execute the script periodically using a Cron job or other scheduling service.
- **spark-submit** script is an available script on any operating system that supports Java. You can develop your spark application on Windows machine and upload the Jar file to a Linux cluster and run the **spark-submit** script on the Linux cluster.

Spark SQL practice:

House Price Problem

Spark SQL

- Structured data is any data that has a schema — that is, a known set of fields for each record.
- Spark SQL provides a **dataset** abstraction that simplifies working with structured datasets. Dataset is similar to tables in a relational database.
- More and more Spark workflow is moving towards Spark SQL.
- **Dataset** has a natural schema, and this lets Spark store data in a more efficient manner and can run SQL queries on it using actual SQL commands.

Important Spark SQL Concept

- DataFrame
- Dataset

DataFrame

- Spark SQL introduces a tabular data abstraction called **DataFrame** since Spark 1.3
- A **DataFrame** is a data abstraction or a domain-specific language for working with structured and semi-structured data.
- **DataFrames** store data in a more efficient manner than native RDDs, taking advantage of their schema.
- It uses the immutable, in-memory, resilient, distributed and parallel capabilities of RDD, and applies a structure called **schema** to the data, allowing Spark to manage the **schema** and only pass data between nodes, in a much more efficient way than using Java serialization.
- Unlike an RDD, data is organized into named **columns**, like a table in a relational database.

Important Spark SQL Concept

- DataFrame
- Dataset

Dataset

- The Dataset API, released since Spark 1.6, it provides:
 - the familiar object-oriented programming style
 - compile-time type safety of the RDD API
 - the benefits of leveraging schema to work with structured data
- A dataset is a set of structured data, not necessarily a row but it could be of a particular **type**.
- Java and Spark will know the **type** of the data in a dataset at compile time.

DataFrame and Dataset

- Starting in Spark 2.0, **DataFrame** APIs merge with **Dataset** APIs.
- **Dataset** takes on two distinct APIs characteristics: a **strongly-typed API** and an **untyped API**.
- Consider **DataFrame** as **untyped** view of a **Dataset**, which is a **Dataset** of **Row** where a **Row** is a generic **untyped** JVM object.
- **Dataset**, by contrast, is a collection of **strongly-typed** JVM objects.

Spark SQL Joins

Spark SQL join Vs. core Spark join

- Spark SQL supports the same basic join types as core Spark.
- Spark SQL Catalyst optimizer can do more of the heavy lifting for us to optimize the join performance.
- Using Spark SQL join, we have to give up some of our control. For example, Spark SQL can sometimes push down or re-order operations to make the joins more efficient. The downside is that we don't have controls over the **partitioner** for Datasets, so we can't manually avoid shuffles as we did with core Spark joins.

Spark SQL Join Types

- The standard SQL join types are supported by Spark SQL and can be specified as the `joinType` when performing a join.

```
public Dataset<Row> join(Dataset<?> right,
                           Column joinExprs,
                           String joinType)

makerSpaceDataset.join(postCodeDataset, col( colName: "postcode"), "left_outer");
```

- Spark SQL join types:
 - inner
 - outer
 - left outer
 - right outer
 - left semi

Inner join

Name	Age
John	20
Henry	50

Name	Country
Lisa	US
Henry	Korea

Name	Age	Country
Henry	50	Korea

Left outer joins

Name	Age
John	20
Henry	50

Name	Country
Lisa	US
Henry	Korea

Name	Age	Country
Henry	50	Korea
John	20	-

Right outer joins

Name	Age
John	20
Henry	50

Name	Country
Lisa	US
Henry	Korea

Name	Age	Country
Henry	50	Korea
Lisa	-	US

Left semi joins

Name	Age
John	20
Henry	50

Name	Country
Lisa	US
Henry	Korea

Name	Age
Henry	50

- The postcode in the **maker space** RDD is the **full postcode**,
 - **W1T 3AC**
- The postcode in the **postcode** dataset is only the **prefix of the postcode**.
 - **W1T**
- **Join condition:**
 - If the postcode column in the **maker space** data source starts with the postcode column in the **postcode** data source.
- **Conner case:**
 - **W14D T2Y** might match both **W14D** and **W14**
- **Solution:**
 - Append a **space** to the **postcode prefix**
 - Then **W14D T2Y** only matches “**W14D**”, not “**W14**”

- Conner case:
 - W14D T2Y

Strongly Typed Dataset

Dataset and DataFrame

- A **dataset** is a **strongly typed** collection of domain-specific objects that can be transformed in parallel using functional or relational operations.
- Each Dataset also has an **untyped view** called a **DataFrame**, which is a Dataset of **Row**.

Dataset<Row> = **DataFrame**

Row

- Row objects represent records inside dataset and are simply fixed-length arrays of fields.
- Row objects have a number of getter functions to obtain the value of each field given its index. The get method takes a column number and returns us an object type; we are responsible for casting the object to the correct type.

```
Object field7 = row.get(7);
```

- For primitive and boxed types, there is a **get type** method, which returns the value of that type.

```
long field1 = row.getLong(i: 1);
boolean field2 = row.getBoolean(i: 2);
```

Encoders

- When it comes to serializing data, the Dataset API has the concept of encoders which translate between JVM representations which are Java objects and Spark's internal binary format.
- Spark has built-in encoders such as integer encoder or long encoder which are very advanced in that they generate bytecode to interact with off-heap data and provide on-demand access to individual attributes without having to de-serialize an entire object.

```
Encoders.INT();
Encoders.LONG();
```

Dataset or RDD?

RDD

- RDD is the primary user-facing API in Spark.
- At the core, RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.
- using RDDs when:
 - Low-level transformation, actions and control on our dataset are needed.
 - Unstructured data, such as media streams or streams of text.
 - Need to manipulate our data with functional programming constructs than domain specific expressions.
 - Optimization and performance benefits available with Datasets are NOT needed.

Use Datasets when

- Rich semantics, high-level abstractions, and domain specific APIs are needed.
- Our processing requires aggregation, averages, sum, SQL queries and columnar access on semi-structured data.
- We want a higher degree of type-safety at compile time, typed JVM objects, and the benefit of Catalyst optimization.
- Unification and simplification of APIs across Spark Libraries are needed.

In summary

- **Consider using Datasets over RDDs, if possible.**
- RDD will remain to be the one of the most critical core components of Spark, and it is the underlying building block for Dataset.

Conversion between **Dataset** and **RDD**

Dataset

- Datasets **are the new hotness.**
- **Mlib** is on a shift to Dataset based API.
- Spark streaming is also moving towards, something called **structured streaming** which is heavily based on Dataset API.

Are RDDs being treated as second class citizens?
Are they being deprecated?

NO

- The RDDs are still the core and fundamental building block of Spark.
- Both DataFrames and Datasets are built on top of RDDs.

Tune the Performance of Spark SQL

Built-in Optimization

Spark SQL has some built-in optimizations such as **predicate push-down** which allows Spark SQL to move some parts of our query **down** to the engine we are querying.

Introduction to Running Spark in a Cluster

- Spark Mode
 - run in IntelliJ IDEA in **Spark local mode**.
 - scale computation by adding more spark nodes and running in **cluster mode**.

- Writing applications for parallel cluster execution uses the same API we have already learned in this course.
- The spark programs we have written so far can run on a cluster out of the box. This allows us to rapidly prototype our spark applications on smaller datasets locally, then run unmodified code against a large dataset and potentially on a large cluster.

Spark Components and User Categories

Cluster manager

- The cluster manager is a pluggable component in Spark.
- Spark is packaged with a built-in cluster manager called the Standalone cluster manager.
- There are other types of Spark manager master such as
 - Hadoop Yarn
 - A resource management and scheduling tool for a Hadoop MapReduce cluster.
 - Apache Mesos
 - Centralized fault-tolerant cluster manager and global resource manager for your entire data center.
- The cluster manager abstracts away the underlying cluster environment so that you can use the same unified high-level spark API to write spark program which can run on different clusters.
- Spark provides a single script you can use to submit your application to it called spark-submit.

Package Spark application

&&

Use *spark-submit*

Steps

1. Download Spark distribution to our local box.
2. Export our Spark application to a jar file.
3. Submit our application to our local Spark cluster through spark-submit script.

Steps

1. Download Spark distribution to our local box.
2. Export our Spark application to a jar file.
3. Submit our application to our local Spark cluster through spark-submit script.

Run Spark Application on Amazon **EMR** cluster (Elastic MapReduce)

Amazon EMR

- Amazon EMR cluster provides a managed Hadoop framework that makes it easy, fast, and cost-effective to process vast amounts of data across dynamically scalable Amazon EC2 instances.
- We can also run other popular distributed frameworks such as Apache Spark and HBase in Amazon EMR, and interact with data in other AWS data stores such as Amazon S3 and Amazon DynamoDB.

S3(Amazon Simple Storage Service)

- We are going run our Spark application on top of the Hadoop cluster, and we will put the input data source into the **S3**.
- **S3** is a distributed storage system and AWS's equivalent to HDFS.
- We want to make sure that
 - Our date is coming from some distributed File system that can be accessed by every node on our Spark cluster.
 - Our Spark application doesn't assume that our input data sits somewhere on our local disk because that will not scale.
- By saving our input data source into **S3**, each spark node deployed on the EMR cluster can read the input data source from **S3**.