

INFO-H515: BIG DATA: DISTRIBUTED MANAGEMENT

Lecture 2: Distributed Processing with Spark

Dimitris Sacharidis

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

WHAT IS SPARK?



- Apache Spark is a fast, **in-memory**, **distributed** data processing engine.

WHAT IS SPARK?



- Apache Spark is a fast, **in-memory**, **distributed** data processing engine.

Speed:

- Run computations in **memory**
- **100 times faster** than MapReduce when running fully in memory
- **10 times faster** than MapReduce, even when running on disk

WHAT IS SPARK?



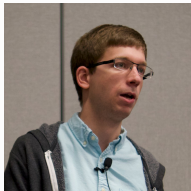
- Apache Spark is a fast, **in-memory, distributed** data processing engine.
- Supports efficient execution of different kinds of workloads: batch, streaming, machine learning and SQL

Ease of programming

- A general programming model based on composing **arbitrary operators**.
- Allows **combining** different processing models seamlessly in the **same application**:
 - Data classification through Spark machine learning library.
 - Streaming data through source via Spark Streaming.
 - Querying the resulting data in real time through Spark SQL.

A VERY BRIEF HISTORY OF SPARK

- **2009:** Spark was created in the [UC Berkeley R&D Lab](#), later it becomes [AMPLab](#).
 - Created by [Matei Zaharia](#) during his PhD studies under [Ion Stoica](#). He also designed the core scheduling algorithms used in Apache Hadoop.
- **2010:** Open sourced under BSD license.
- **2013:** Spark was donated to Apache Software Foundation.
 - Also: founding of [DataBricks](#) by original creators to commercialize Spark.
- **2014:** It becomes a top-level Apache project.
- **2016:** Spark 2.0 released (e.g., SQL2003 support).
- **2020:** Spark 3.0 released.



Matei Zaharia



Ion Stoica

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

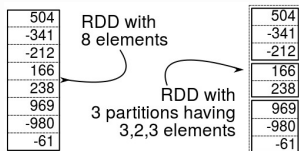
WHAT IS AN RDD?

504
-341
-212
166
238
969
-980
-61

RDD with
8 elements

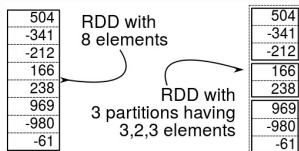
RDD = Resilient Distributed Dataset

- Spark's abstraction for representing a **very large dataset**.
- RDD = collection of elements
- RDDs can contain **any types of objects** as elements , including user-defined classes.
- Under the hood, Spark will automatically **distribute the data** contained in RDDs across your cluster and **parallelize** the operations you perform on them.



RDDs are **distributed**

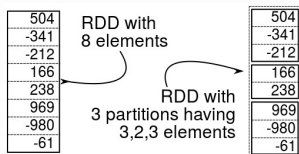
- Each RDD is broken into multiple pieces called **partitions**, and these partitions are **divided** across the nodes in the **cluster**.
- Partitions are operated on **in parallel**



RDDs are **distributed**

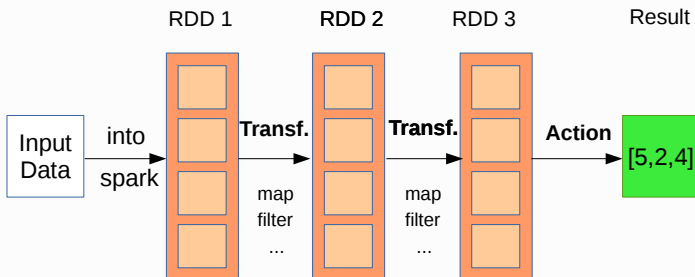
- Each RDD is broken into multiple pieces called **partitions**, and these partitions are **divided** across the nodes in the **cluster**.
- Partitions are operated on **in parallel**
- The partitioning process is typically done **automatically by Spark**
- User can (and sometimes should) influence partitioning process.

RDD BASICS



RDDs are **immutable**

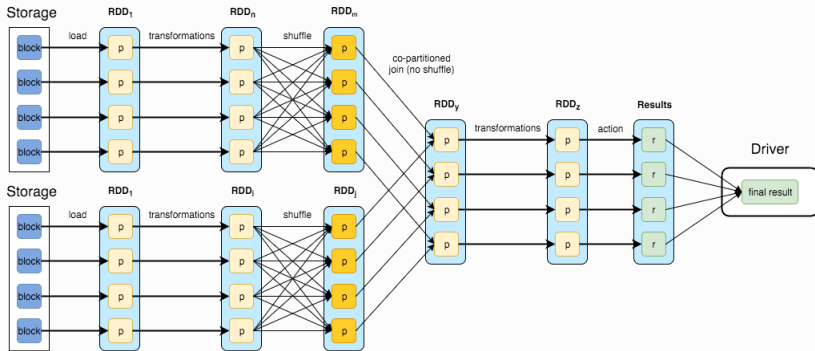
- They **cannot be changed** after they are created
- Immutability **rules out** a significant set of **potential problems** due to updates from **multiple threads** at once.



Basic workflow in Spark:

- Generate **initial RDDs** from external data.
- **Transformations** create new RDDs from existing ones
- **Actions** transform RDDs into normal programming language values (lists, numbers, file, ...).

RDD BASICS



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 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**.

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

HOW TO CREATE AN RDD (1/2)

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

```
sc = SparkContext("local", "MySparkApplicationName")
inputIntegers = list(range(1, 6))
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 when working with **large datasets**.

HOW TO CREATE AN RDD (2/2)

- Load RDDs from external storage by calling `textFile` method on `SparkContext`.

```
sc = SparkContext("local", "textfile")  
lines = sc.textFile("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.

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations**

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

TRANSFORMATIONS

- 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**.

```
filteredLines = lines.filter(lambda line: len(line)>0)
```

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**.

```
URLs = sc.textFile("in/urls.text")  
URLs.map(MakeHttpRequest)
```

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

```
lines = sc.textFile("in/uppercase.text")  
lengths = lines.map(lambda line: len(line))
```

FLATMAP() TRANSFORMATION

- flatMap is a transformation to **create an RDD** from an **existing RDD**.
- It takes **each element** from an existing RDD and it can produce **0, 1 or many outputs for each element**.

FLATMAP VS MAP (1/2)

```
def map(self, f, preservespartitioning=False):
```

```
    """
```

Return a new RDD by applying a function to each element of this RDD.

```
>>> rdd = sc.parallelize(["a a", "b b", "c c c"])
>>> sorted(rdd.map(lambda x: x.split()).collect())
[('a', 'a'), ('b', 'b'), ('c', 'c', 'c')]
    """
```

```
def flatMap(self, f, preservesPartitioning=False):
```

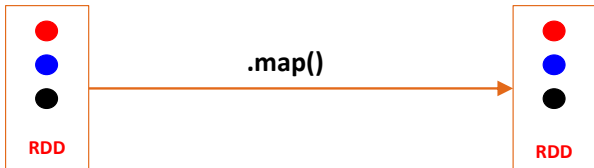
```
    """
```

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
>>> rdd = sc.parallelize(["a a", "b b", "c c c"])
>>> sorted(rdd.flatMap(lambda x: x.split()).collect())
['a', 'a', 'b', 'b', 'c', 'c', 'c']
    """
```


map:

1 to 1 relationship



flatMap:

1 to many relationship



SET OPERATIONS

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

- sample
- distinct

```
def sample(self, withReplacement, fraction, seed=None):
    """
    Return a sampled subset of this RDD.

    :param withReplacement: can elements be sampled multiple times
        (replaced when sampled out)
    :param fraction: expected size of the sample as a fraction of this
        RDD's size
        without replacement: probability that each element is chosen;
        fraction must be [0, 1]
        with replacement: expected number of times each element is chosen;
        fraction must be >= 0

    :param seed: seed for the random number generator

    .. note:: This is not guaranteed to provide exactly the fraction specified
        of the total count of the given :class:'DataFrame'.
    """
```

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

DISTINCT

```
def distinct(self, numPartitions=None):  
    """  
    Return a new RDD containing the distinct elements in this RDD.  
  
    >>> sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())  
    [1, 2, 3]  
    """
```

- 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.

DISTINCT

```
def distinct(self, numPartitions=None):  
    """  
    Return a new RDD containing the distinct elements in this RDD.  
  
    >>> sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())  
    [1, 2, 3]  
    """
```

- 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.

Shuffling:

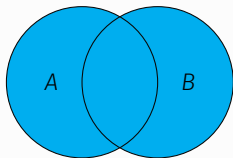
- Physical movement of data between partitions is called **shuffling**. It occurs when data from multiple partitions needs to be combined in order to build partitions for a new RDD .

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

- union
- intersection
- subtract
- cartesian product

UNION

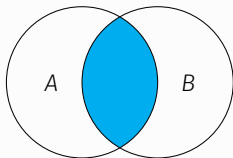
```
def union(self, other):  
    """  
    Return the union of this RDD and another one.  
    """
```



- 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

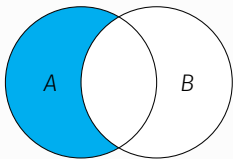
```
def intersection(self, other):  
    """  
    Return the intersection of this RDD and another one. The output will  
    not contain any duplicate elements, even if the input RDDs did.  
  
    .. note:: This method performs a shuffle internally.  
    """
```



- 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

```
def subtract(self, other, numPartitions=None):  
    """  
    Return each value in C{self} that is not contained in C{other}.  
    """
```



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

CARTESIAN PRODUCT

```
def cartesian(self, other):  
    """  
    Return the Cartesian product of this RDD and another one, that is, the  
    RDD of all pairs of elements C{(a, b)} where C{a} is in C{self} and  
    C{b} is in C{other}.  
    """
```

$$\{1, 2, 3, 4\} \times \{a, b, c\}$$
$$=$$
$$\left\{ \begin{array}{lll} (1, a), & (1, b), & (1, c), \\ (2, a), & (2, b), & (2, c), \\ (3, a), & (3, b), & (3, c) \end{array} \right\}$$

- 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.

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions**

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

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.

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

COLLECT ACTION

- 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**.

COUNT AND COUNTBYVALUE ACTIONS

- If you just want to count how many rows in an RDD, the `count` operation is a quick way to do that. It would return the count of the elements.
- `countbyValue` will look at unique values in 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 values you have.

TAKE ACTION

```
words = wordRdd.take(3)
```

- **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 ACTION

```
airportsNameAndCityRdd.saveAsTextFile('out/airports.text')
```

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

REDUCE ACTION

```
def reduce(self, f):  
    """  
    Reduces the elements of this RDD using the specified commutative and  
    associative binary operator. Currently reduces partitions locally.  
    """  
    product = integerRdd.reduce(lambda x, y: x*y)
```

- The **reduce** action 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 the **reduce** action, we can perform different types of **aggregations**.

- **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.

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables**

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

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

```
>>> accum = sc.accumulator(0)
>>> accum
Accumulator<id=0, value=0>

>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
```

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.
 - **Normal variables are resent for each operation!**

```
>>> broadcastVar = sc.broadcast([1, 2, 3])  
<pyspark.broadcast.Broadcast object at 0x102789f10>
```

```
>>> broadcastVar.value  
[1, 2, 3]
```

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

- 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 is each row is **one key** mapped 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 **simpler** and **more efficient**.
- A Pair RDDs is a particular type of RDD that can store **key-value pairs**.

HOW TO CREATE PAIR RDD'S

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 RDD'S FROM TUPLES

- Python Tuples

```
my_tuple = "Lily", 23          name = my_tuple[0]
my_tuple = ("Lily", 23)        age = my_tuple[1]
my_tuple = tuple(["Lily", 23])
```

- Convert lists of tuples to Pair RDDs with `parallelize`

```
sc.parallelize([("panda", 0), ("pink", 3)])
```

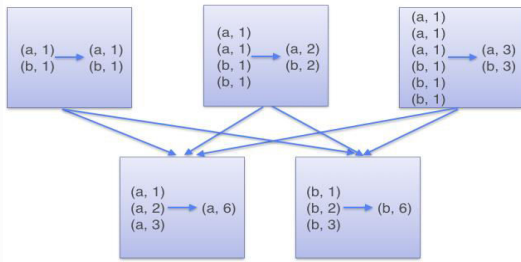
- Pair RDDs are allowed to use **all the transformations** available to regular RDDs, and thus support the same functions as **regular RDDs**.
- Since pair RDDs contain tuples, we need to pass functions that **operate on tuples** rather than on individual elements.

- The **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.

- 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.

- 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 **parallel 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 TRANSFORMATION



GROUPBYKEY TRANSFORMATION

- 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 containing pairs with fields of **type K**, and **type Iterable V**.

```
def groupByKey(self, numPartitions=None, partitionFunc=portable_hash):  
    """  
    Group the values for each key in the RDD into a single sequence.  
    Hash-partitions the resulting RDD with numPartitions partitions.  
    """
```


SORTBYKEY AND SORTBY TRANSFORMATION

```
def sortByKey(self, ascending=True, numPartitions=None, keyfunc=lambda x: x):
```

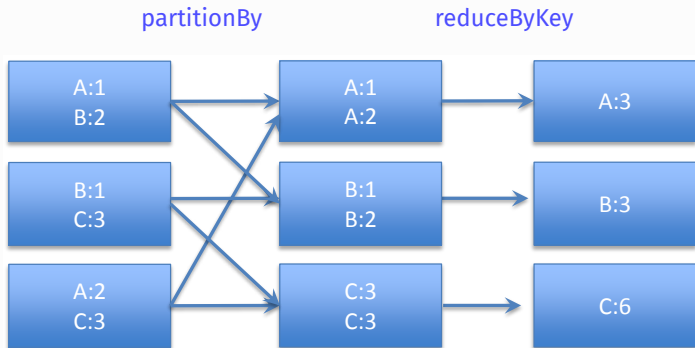
- We can sort a Pair RDD as long as there is an **ordering** defined on the **key**.
- Once we have sorted our Pair RDD, any **subsequent call** on the sorted Pair RDD to collect or save will **return us ordered data**.
- ... and if we do not want to **sort** by the key but **by the value**, we can use **SortBy**.

```
def sortBy(self, keyfunc, ascending=True, numPartitions=None):
```

- The **sortBy** transformation maps the pair RDD to (keyfunc(x), x) tuples using the function it receives as parameter, **applies sortByKey** and, finally, return the values.

REDUCE THE AMOUNT OF SHUFFLE FOR GROUPBYKEY

```
partitionedwordPairRdd = wordPairRdd.partitionBy(3, lambda x: x[0])  
partitionedwordPairRdd.reduceByKey(lambda x,y: x+y).collect()
```

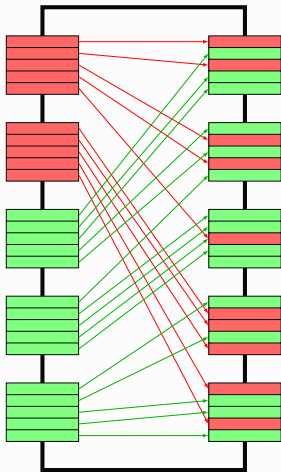


OPERATIONS WHICH CAN BENEFIT FROM PARTITIONING

- **Join** (joins on the **key** of the key-value pairs)
- **leftOuterJoin**
- **rightOuterJoin**
- **groupByKey**
- **reduceByKey**
- **combineByKey**
- **lookup**

SHUFFLED HASH JOIN

- The **shuffled hash join** is the default implementation of Join in Spark.



- Both pair RDDs are **simultaneously** partitioned via a shuffle, based on the same **default partitioner**.
- So key-value pairs with **similar keys** end up in the **same partition**.
- On that partition we perform locally a **classical join algorithm**.
 - If the two RDDs were already partitioned using the same partitioner, the shuffle can be avoided.
 - If you call **partitionBy** before the join, the partitions might already be colocated, and no network traffic is needed.

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation**

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

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 computed by Spark in a lazy fashion**, which means they are not evaluated until they are needed for the result of an **Action**.

LAZY EVALUATION

```
# Nothing would happen when Spark sees textFile() statement.
lines = sc.textFile("in/uppercase.text")

# Nothing would happen when Spark sees filter() transformation.
lineswithFriday = lines.filter(lambda line: line.startswith("Friday"))

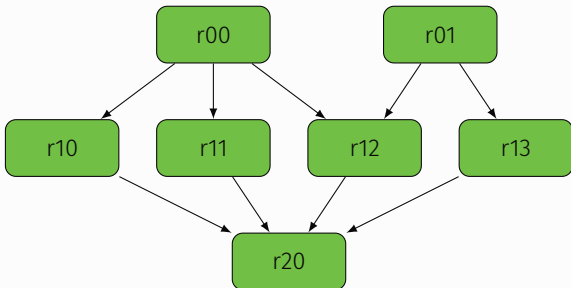
# Spark only starts loading the uppercase.text file when first()
# action is called on the lineswithFriday RDD.
lineswithFriday.first()

# 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
```

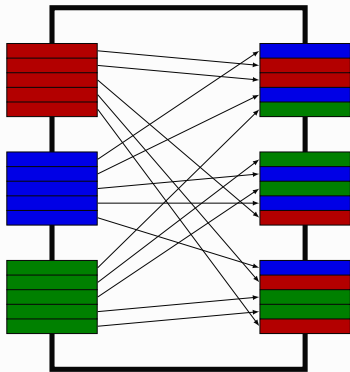

- **Transformations** on RDDs are **lazily evaluated**, meaning that Spark will not begin to execute until it sees an action.
- Rather than thinking of an RDD as containing specific data, it might be better to think of each RDD as consisting of **instructions** on how to compute the data that we build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

THE RDD LINEAGE GRAPH

```
r00 = sc.parallelize(range(10))  
r01 = sc.parallelize(range(0, 100, 10))  
r10 = r00.cartesian(r00)  
r11 = r00.map(lambda x: (x, x))  
r12 = r00.zip(r01)  
r13 = r01.keyBy(lambda x: x/20)  
r20 = sc.union([r10, r11, r12, r13])
```



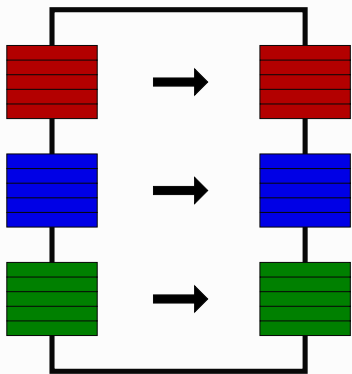
- Transformations that require a shuffle phase



- Intersection
- Distinct
- ReduceByKey
- GroupByKey
- Join
- Cartesian
- Repartition
- Coalesce
- ...

NARROW TRANSFORMATIONS

- Transformations that do not need a shuffle phase
- Are grouped by Spark in a single stage, which is called **pipelining**



- Map
- FlatMap
- MapPartition
- Filter
- Sample
- Union
- ...

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence**

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

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 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.

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> rdd.persist().is_cached
True
```

DIFFERENT STORAGE LEVEL

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 serialized 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.

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

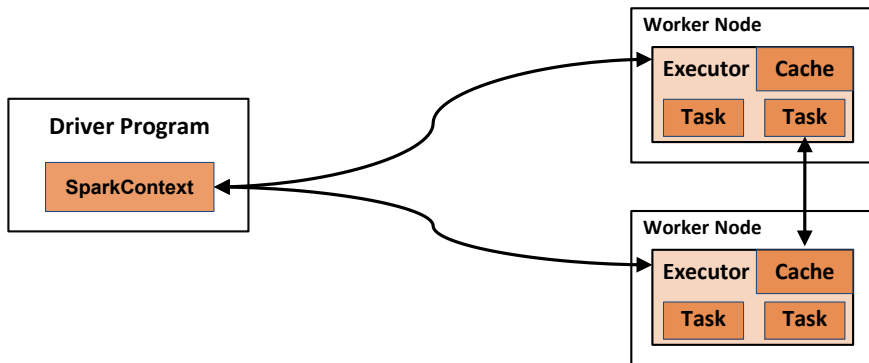
- Master-Slave Architecture**

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

SPARK – MASTER-SLAVE ARCHITECTURE



Driver Program

..in New York
and in US
.....History of
New York of
new jersey..
.. Metro of
New York ...

Executors

..in New York
and
in US..

In:	2
new:	1
york:	1
us:	1
and:	1

..History of
New York of
new jersey..

history:	1
new:	2
york:	1
of:	2
jersey:	1

..Metro of
New York...

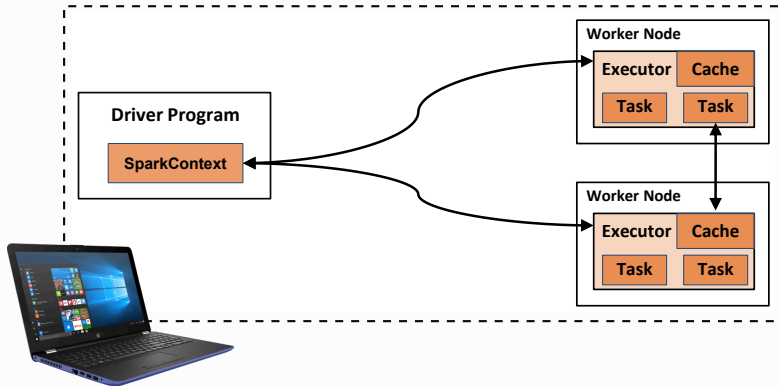
metro:	1
of:	2
new:	1
york:	1

Driver Program

In:	2
new:	4
york:	3
us:	1
and:	1
history:	1
of:	2
jersey:	1
Metro:	1
...	

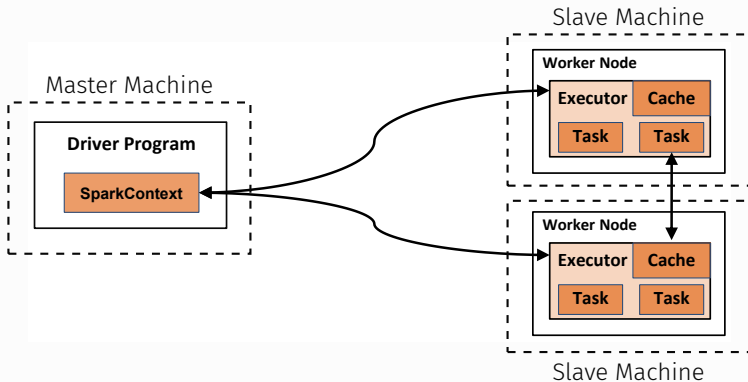
RUNNING SPARK IN LOCAL MODE

All components run in the same process on the local machine



RUNNING SPARK IN CLUSTER MODE

Components (may) run on different machines



OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

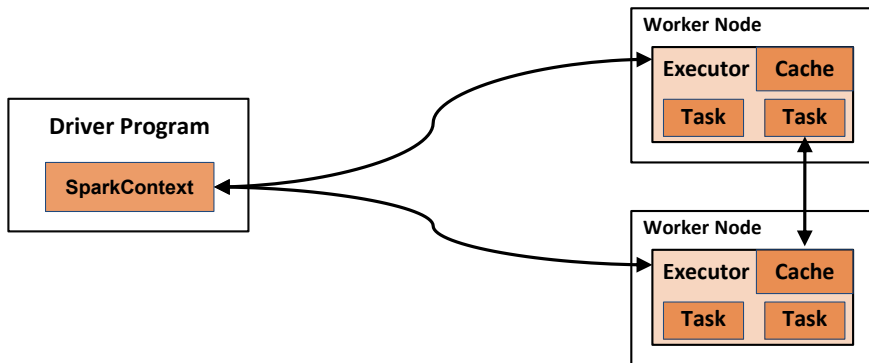
- Master-Slave Architecture

Running Spark in a cluster

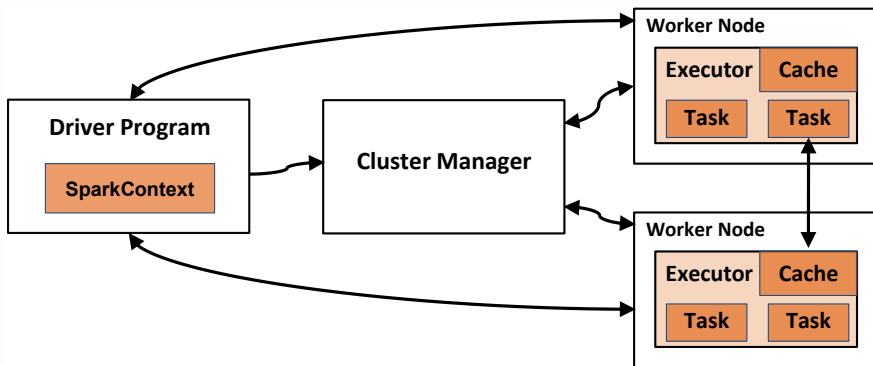
The Apache Spark Eco System

Further Reading

RUNNING SPARK IN CLUSTER MODE



RUNNING SPARK WITH A 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**.
- You can use other types of cluster managers 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.
- You can use **spark-submit** to submit an application to the cluster

RUNNING SPARK APPLICATIONS ON A CLUSTER

- The user submits an application using the `spark-submit` script
 - available in Spark's bin directory
- 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 dataframe 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.

Typical usage

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

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 a Windows machine and upload the Python script to a Linux cluster and run the spark-submit script on the Linux cluster.

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

Running Spark in a cluster

The Apache Spark Eco System

Further Reading

THE APACHE SPARK ECO SYSTEM

Spark
SQL

Spark
Streaming
(Streaming)

MLlib
(Machine
learning)

GraphX
(Graph
computation)

SparkR
(R on Spark)

Apache Spark Core API

R

SQL

Python

Scala

Java



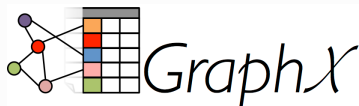
- Spark package designed for working with **relational data** which is built on top of Spark Core.
- Provides an **SQL-like** interface for working with structured data.
- Offers two special types of RDDs:
 - **DataFrames** Table-like structure. Weakly typed. (Scala, Java, Python, R)
 - **DataSets** Strongly typed collections. (Only available in Scala and Java.)
- Fairly sophisticated optimizer for SQL queries: **Spark SQL Catalyst**



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



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



- 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.

OUTLINE

Introduction

Resilient Distributed Datasets (RDDs)

Operations on RDDs

- Creation

- Transformations

- Actions

- Special global variables

Pair RDDs

Spark program execution

- Lazy Evaluation

- Caching and Persistence

- Master-Slave Architecture

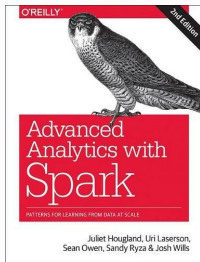
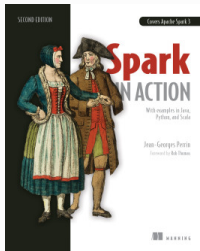
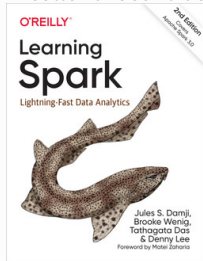
Running Spark in a cluster

The Apache Spark Eco System




Further Reading

FURTHER READING

Excellent recent books:



- **Learning Spark 2nd Edition (2020)**, written by Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee
- **Spark in Action 2nd Edition (2020)**, written by Jean-Georges Perrin
- **High Performance Spark (2017)**, written by Holden Karau and Rachel Warren
- **Advanced Analytics with Spark 2nd Edition (2017)**, written by Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills


 The Internals of Apache Spark  

The Internals of Apache Spark 3.2.0



Welcome to **The Internals of Apache Spark** online book! 👍

I'm [Jacek Laskowski](#), an IT freelancer specializing in [Apache Spark](#), [Delta Lake](#) and [Apache Kafka](#) (with brief forays into a wider data engineering space, e.g. [Trino](#) and [ksqlDB](#), mostly during [Warsaw Data Engineering](#) meetups).

I'm very excited to have you here and hope you will enjoy exploring the internals of Apache Spark as much as I have.

 **Flannery O'Connor**

I write to discover what I know.

 "The Internals Of" series 

Expect text and code snippets from a variety of public sources. Attribution follows.

Now, let's take a deep dive into [Apache Spark](#) 🔥

The Internals of Apache Spark (2021), written by Jacek Laskowski

QUESTIONS?

ACKNOWLEDGEMENTS

Based on content created by Jan Hidders and Stijn Vansummeren.