LEVEL UP
www.level-up.one

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.

Install Java 8 and GIT

Why do we need to install the Java Development Kit (JDK)

- Spark is built on top of the **Scala** programming language, which runs on the **Java Virtual Machine**.
- To run our programs we will use the Python API for Spark: **PySpark**
- PySpark is built on top of **Spark's Java API**, so we need the JDK to be able to run our programs.

Install Java 8 and GIT

Install GIT

Set up Spark!

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.

RDD

- RDDs can contain **any types of objects**, including user-defined classes.

- An RDD is simply a **capsulation** 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.

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.

```
lines = sc.textFile("in/uppercase.text")
linesWithFriday = lines.filter(lambda line: "Friday" in line)
```

Actions

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

```
lines = sc.textFile("in/uppercase.text")
firstLine = lines.first()
```

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.

```
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 working with **large datasets**.

How to create a RDD

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

```
sc = SparkContext("local", "texfile")
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 Elastisearch, 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**.

```
cleanedLines = lines.filter(lambda line: line.strip())
```

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))
```

LEVEL UP
www.level-up.one

Solution to

Airports by latitude problem

LEVEL UP
www.level-up.one

flatMap transformation

flatMap

- 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

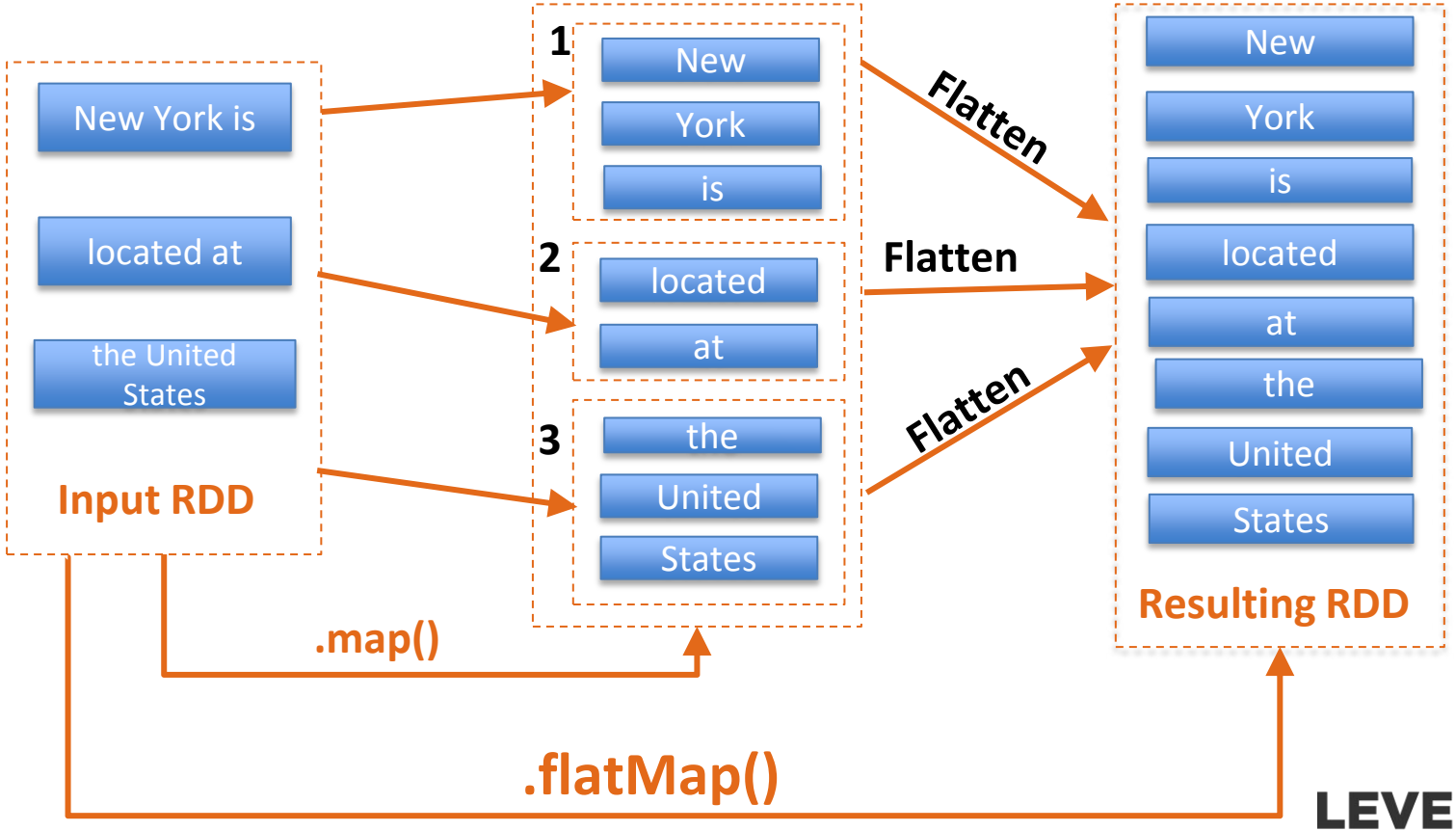
```
def map(self, f, preservesPartitioning=False):
    """
    Return a new RDD by applying a function to each element of this RDD.

    >>> rdd = sc.parallelize(["b", "a", "c"])
    >>> sorted(rdd.map(lambda x: (x, 1)).collect())
    [('a', 1), ('b', 1), ('c', 1)]
    """
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([2, 3, 4])
    >>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
    [1, 1, 1, 2, 2, 3]
    >>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
    [(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
    """
```

LEVEL UP
www.level-up.one

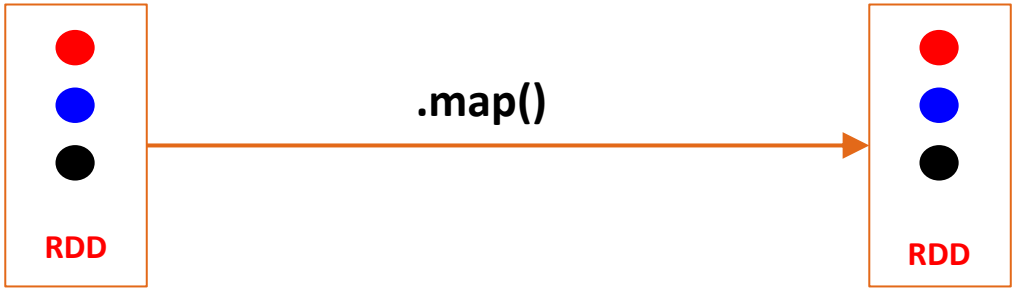
flatMap example: **split lines by space**



LEVEL UP
www.level-up.one

flatMap VS map

map: 1 to 1 relationship



flatMap: 1 to many relationship

LEVEL UP
www.level-up.one

To the code!

flatMap

```
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([2, 3, 4])
    >>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
    [1, 1, 1, 2, 2, 3]
    >>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
    [(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
    """
```

Set operations

Set operations

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

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

sample

```
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.

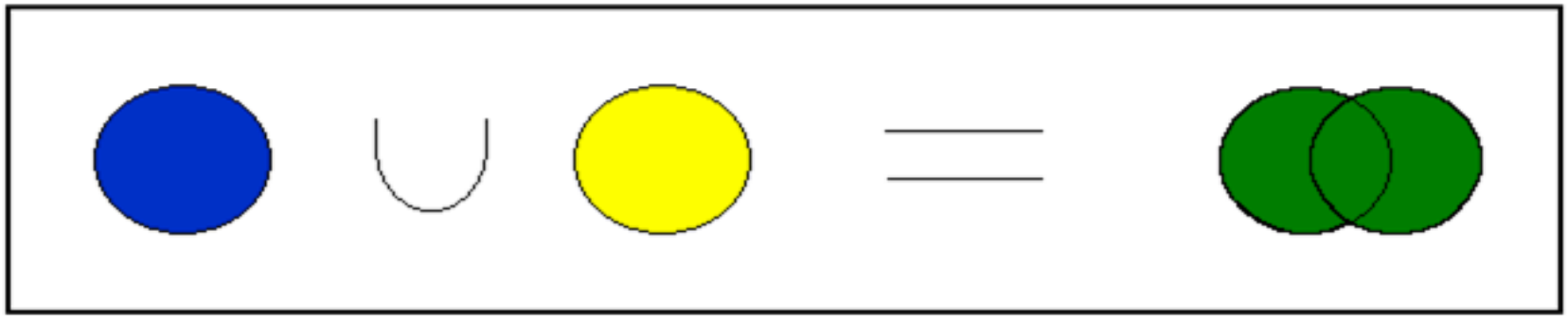
- The distinct transformation is **expensive** because it requires shuffling all the data across partitions to ensure that we receive only one copy of each element.

Set operations

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

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

union operation

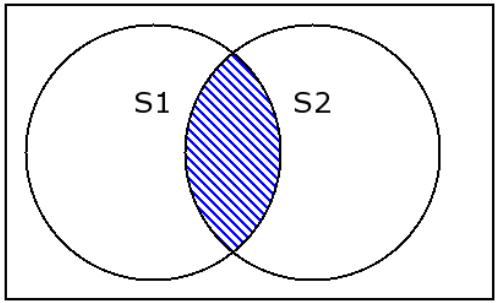


```
def union(self, other):
    """
    Return the union of this RDD and another one.
```

- 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.

LEVEL UP
www.level-up.one

intersection operation

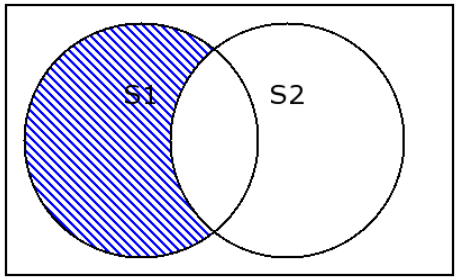



```
def intersection(self, other):
    """
    Return the intersection of this RDD and another one. The output will
    not contain any duplicate elements, even if the input RDDs did.

    .. 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 operation



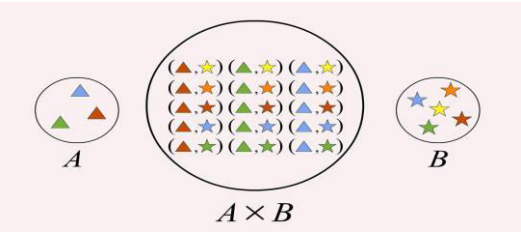

```
def subtract(self, other, numPartitions=None):
    """
    Return each value in C{self} that is not contained in C{other}.
```

- Subtract operation takes in another RDD as an argument and returns us an RDD that **only contains 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.

LEVEL UP
www.level-up.one

cartesian operation




```
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}.
```

- 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.

LEVEL UP
www.level-up.one

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

```
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.

Common Actions in Spark

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

saveAsTextFile

```
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 local file system.

Common Actions in Spark

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

reduce

```
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)
```

- **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**.

Sample solution for the Sum of Numbers problem

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 about 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.

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**.

Summary of RDD Operations

Summary

- **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.

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

```
# 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
```

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:

```
linesWithFriday = lines.filter(lambda line: "Friday" in line)

lengths = lines.map(lambda line: len(line))
```

Actions:

```
words = wordRdd.collect()
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 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 *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?

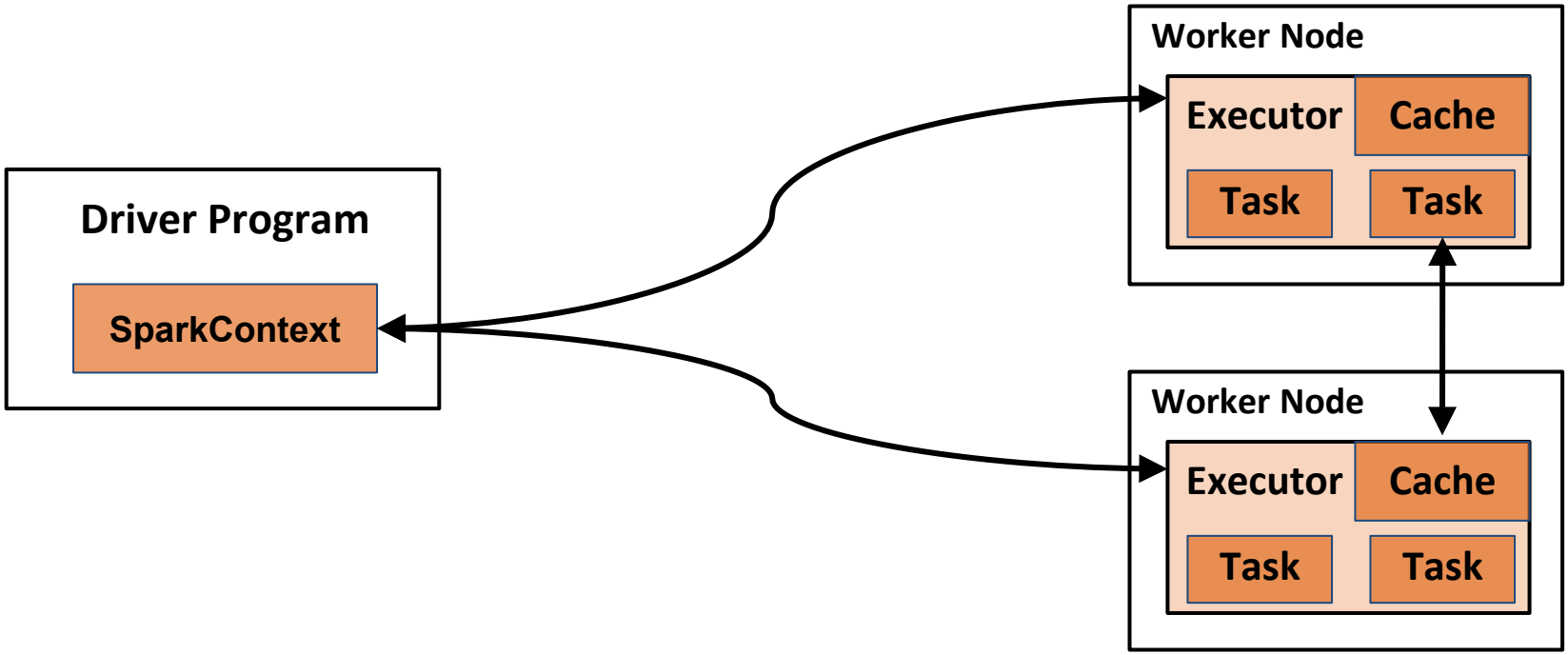
- Spark's storage levels are meant to provide different **trade-offs** between **memory usage** and **CPU efficiency**.
- If the RDDs can fit comfortably with the default storage level, **MEMORY_ONLY** is the ideal option. This is the **most CPU-efficient option**, allowing operations on the RDDs to 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.

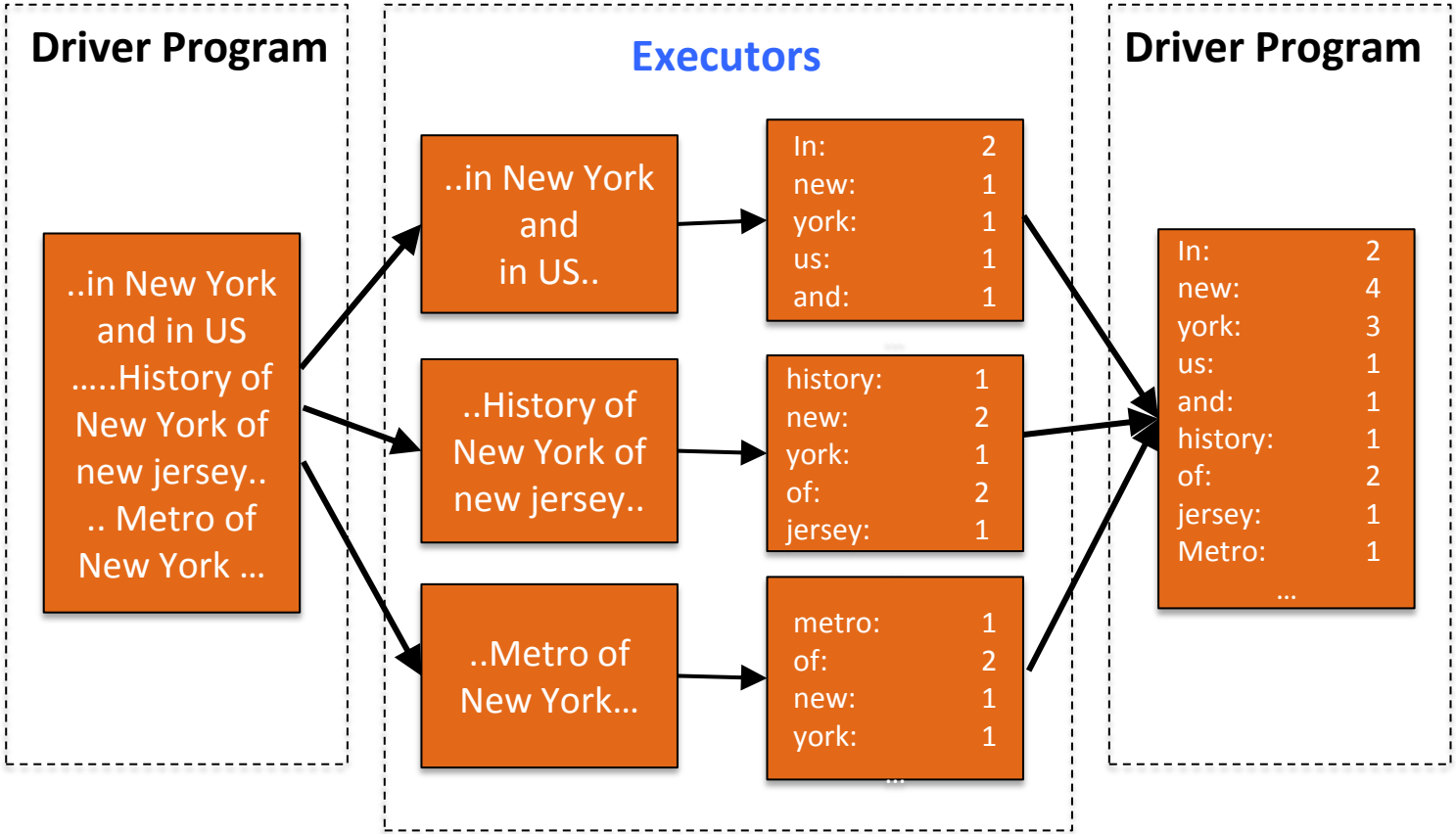
- **What would happen If you attempt to cache too much data to fit in memory?**

  - 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.

Spark Architecture

Spark - Master-Slave Architecture



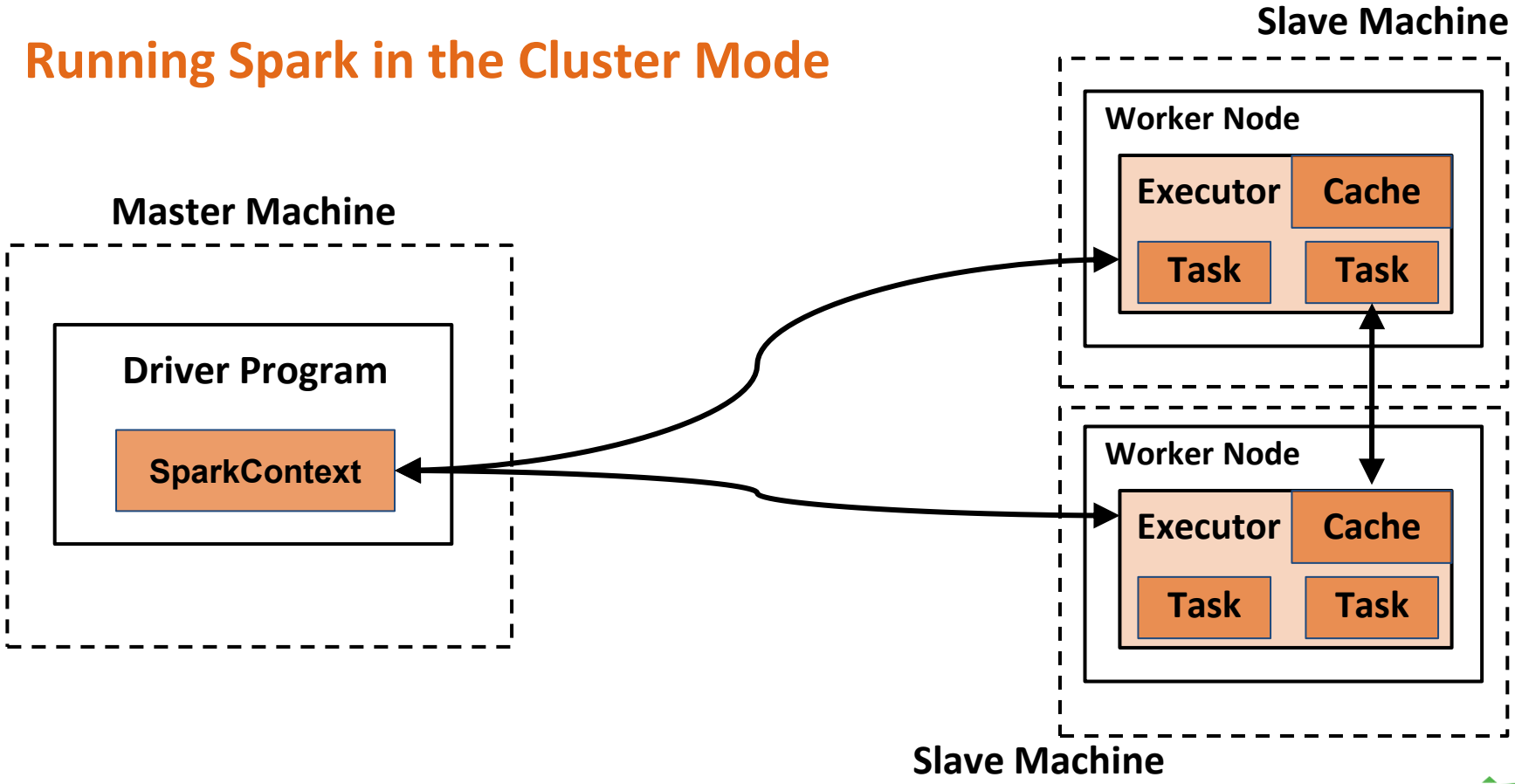


Running Spark in the Local Mode

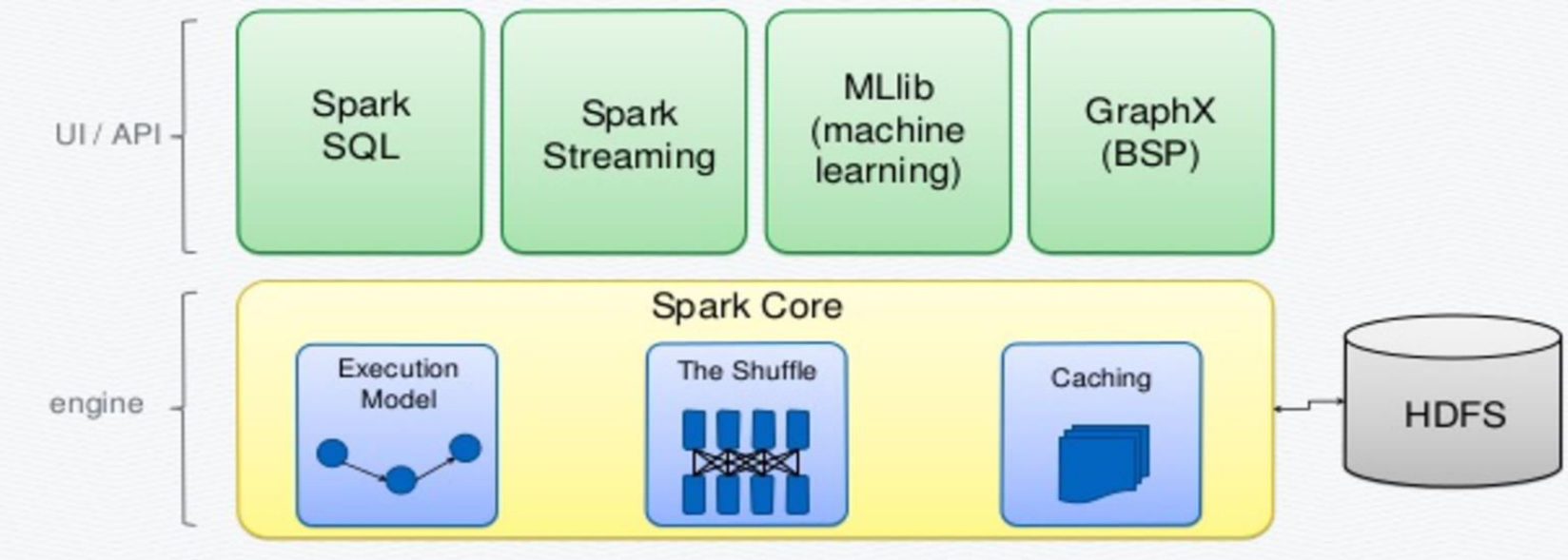
All components run in the same process on the local machine

LEVEL UP
www.level-up.one

Running Spark in the Cluster Mode



Spark Components

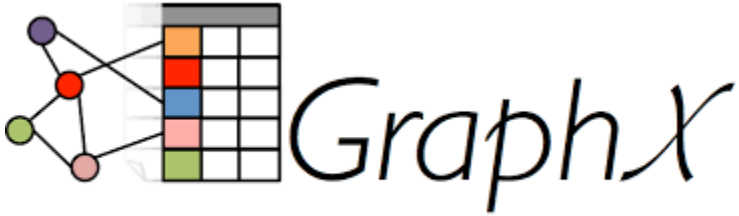


- Spark package designed for working with **structured data** which is built on top of Spark Core.
- Provides an **SQL-like interface** for working with structured data.

- 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.

- 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, 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 leverage **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 and 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

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 is 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.

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

- Python Tuples

```
my_tuple = "Lily", 23
my_tuple = ("Lily", 23)
my_tuple = tuple(["Lily", 23])
```

```
name = my_tuple[0]
age = my_tuple[1]
```

- Convert Tuples to Pair RDDs

```
def parallelize(self, c, numSlices=None):
    """
    Distribute a local Python collection to form an RDD. Using xrange
    is recommended if the input represents a range for performance.
```

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

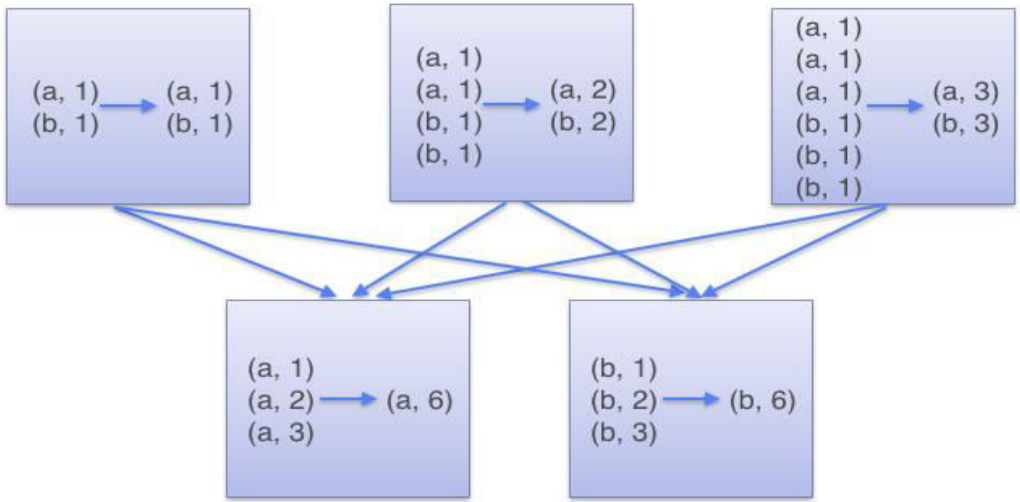
- The **map** transformation also works for Pair RDDs. It can be used to convert an RDD to another one.
- But most of the time, when working with pair RDDs, we don't want to modify the keys, we just want to **access the value** part of our Pair RDD.
- Since this is a typical pattern, Spark provides the **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 aggregation

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



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

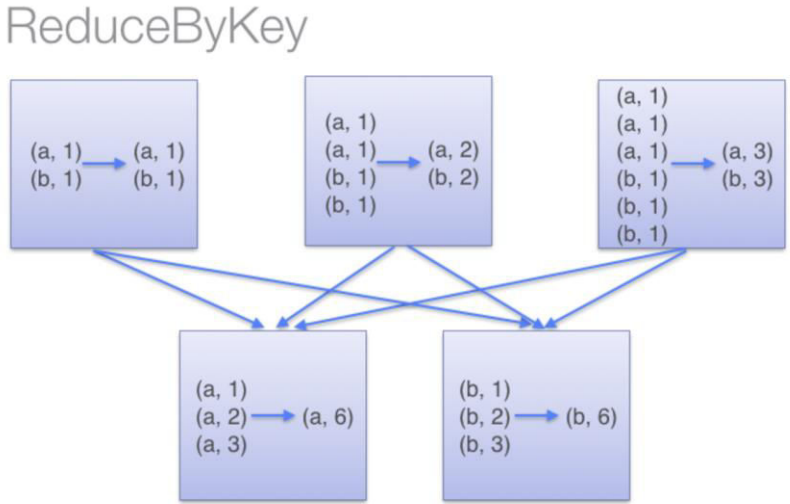
- A common use case for Pair RDD is **grouping our data by key**. For example, viewing all of an account's transactions together.
- If our data is already keyed in the way we want, **groupByKey** will group our data using the key in our Pair RDD.
- Let's say, the input pair RDD has **keys** of **type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

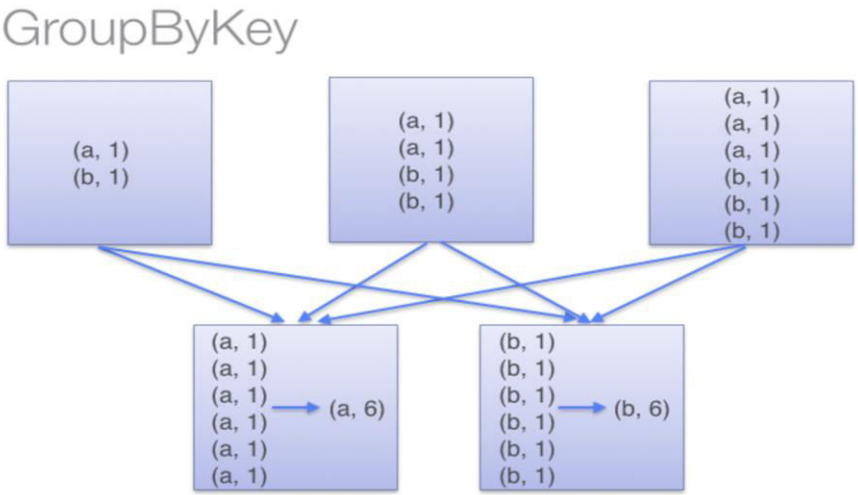
```
def groupByKey(self, numPartitions=None, partitionFunc=portable_hash):
    """
    Group the values for each key in the RDD into a single sequence.
    Hash-partitions the resulting RDD with numPartitions partitions.
```

groupByKey + [**reduce**, **map**, **mapValues**]

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

ReduceByKey (preferred) *VS* GroupByKey





LEVEL UP
www.level-up.one

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

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

```
housePriceAvg.sortByKey(ascending=False)
```

- 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

LEVEL UP
www.level-up.one

- **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**.

A better solution:

1. Use the **sortBy** transformation!

sortBy

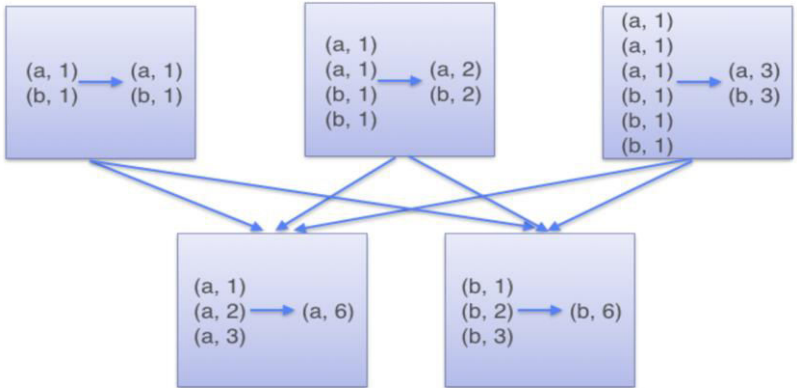
```
def sortBy(self, keyfunc, ascending=True, numPartitions=None):
    """
    Sorts this RDD by the given keyfunc
```

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

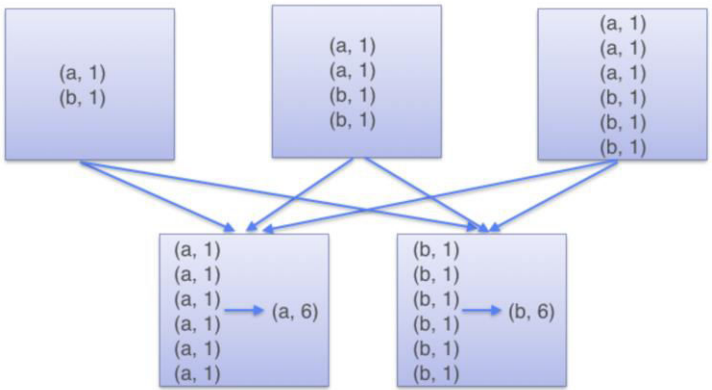
- This allows us to **sort** a pair RDD directly by its **value**.

Data Partitioning

ReduceByKey



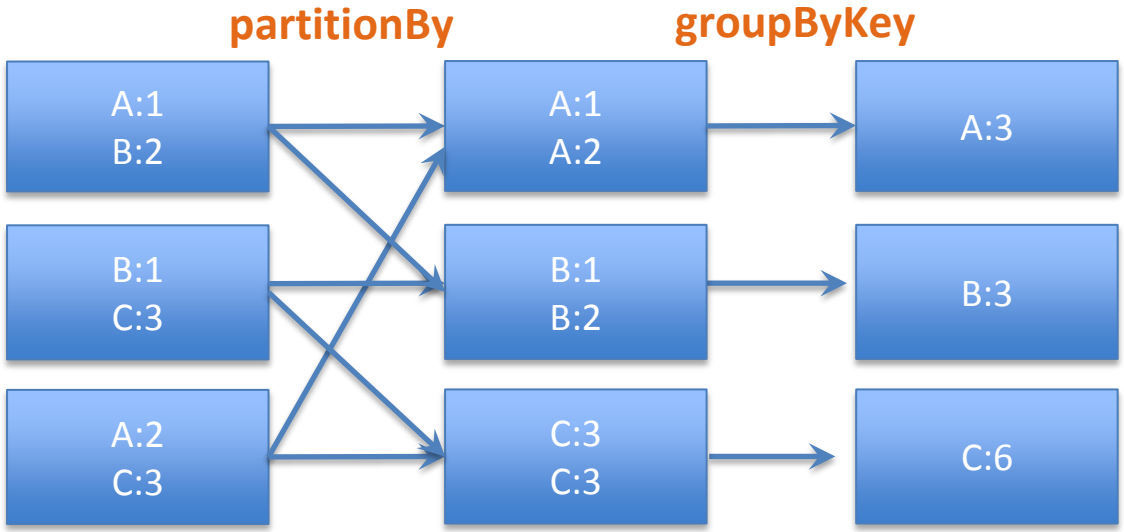
GroupByKey



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

Reduce the amount of shuffle for groupByKey

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



LEVEL UP
www.level-up.one

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.

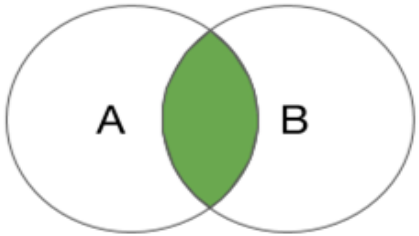
- General guidance is to prefer **mapValues** over **map** operation.

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



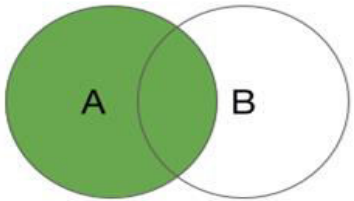
INNER JOIN

```
def join(self, other, numPartitions=None):
    """
    Return an RDD containing all pairs of elements with matching keys in
    C{self} and C{other}.
```

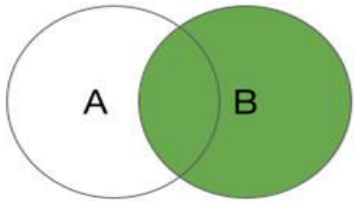
When there are **multiple values** for the **same key** in one of the inputs, the resulting pair RDD will have an **entry for every possible pair** of values with that key from the two input RDDs.

- 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.

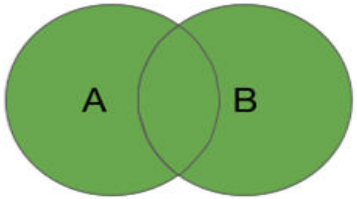
Outer Joins



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.

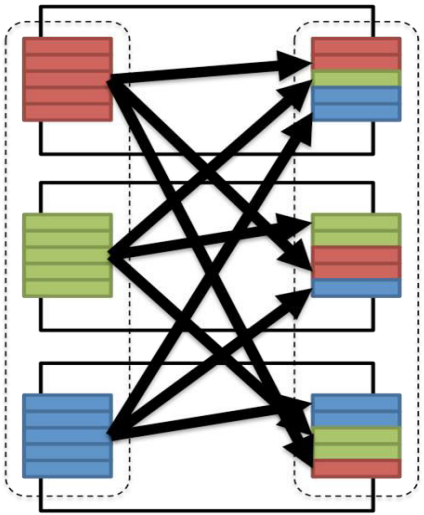
LEVEL UP
www.level-up.one

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

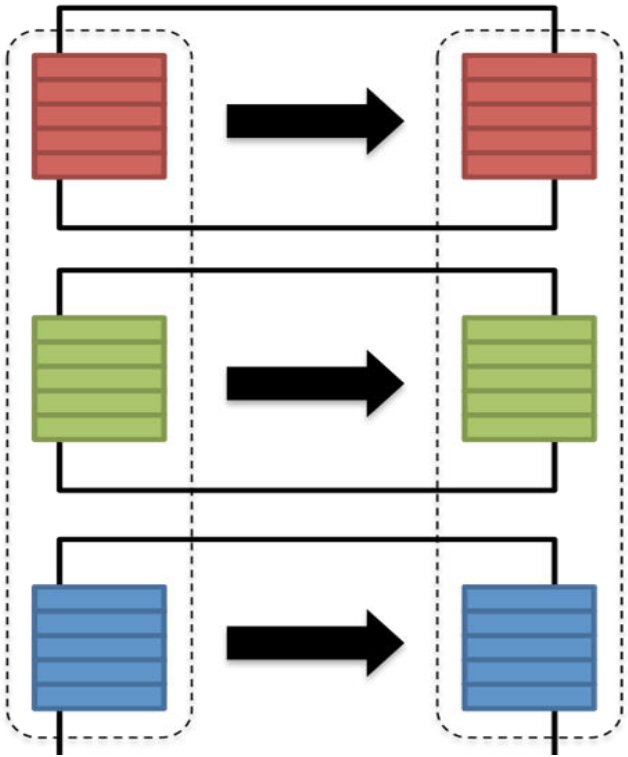
- Input from other partitions are required
- Data shuffling is needed before processing



- 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.

```
from pyspark.rdd import portable_hash
ages.partitionBy(20, partitionFunc = portable_hash)
addresses.partitionBy(20, partitionFunc = portable_hash)
```

LEVEL UP
www.level-up.one

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 **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.

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?

LEVEL UP
www.level-up.one

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
  - E14 9TC
  - SW12 7YC
- The postcode in the **postcode** dataset is only the **prefix of the postcode**.
  - W1T
  - E14
  - SW12

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 passed from the driver program to all the workers in the Spark cluster across the wire.
- 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 dictionary later, the same postcode dictionary would be sent again to each node.

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 let's 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.
- The Dataset API is **not available in Python.**

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.
- The Dataset API is **only available on Java and Scala**.
- For Python we stick with the **DataFrame API.**

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

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.

Spark SQL practice:

House Price Problem

LEVEL UP
www.level-up.one

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 DataFrames, 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 **how** when performing a join.

```
def join(self, other, on=None, how=None):
```

```
joined = makerSpace.join(postCode, makerSpace["Postcode"], "left_outer")
```

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

LEVEL UP
www.level-up.one

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** data source is the **full postcode**.
  - **W1T 3AC**
- The postcode in the **postcode** data source 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 "**

LEVEL UP
www.level-up.one

DataFrame or RDD?

DataFrame

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

DataFrame 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 DataFrames are **NOT needed**.

Use DataFrames 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 the **benefit of Catalyst optimization**.
- **Unification and simplification of APIs** across Spark Libraries are needed.

In summary

- **Consider using DataFrames 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 DataFrames.

Conversion between DataFrame and RDD

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.

Caching

- If you find yourself performing some queries or transformations on a dataframe repeatedly, we should consider **caching the dataframe**, which can be done by calling the cache method on the dataframe.

```
responseDataFrame.cache()
```

- When caching a dataframe, Spark SQL uses an **in-memory columnar storage** for the dataframe.
- If our subsequent queries depend only on subsets of the data, Spark SQL will **minimize the data read, and automatically tune compression** to reduce garbage collection pressure and memory usage.

Configure Spark Properties

```
session = SparkSession.builder \
    .config("CONFIG_KEY","CONFIG_VALUE") \
    .config("spark.sql.codegen", value = False) \
    .getOrCreate()
```

Configure Spark Properties

- **spark.sql.codegen**

```
session = SparkSession.builder \
    .config("spark.sql.codegen", value = False) \
    .getOrCreate()
```

- It will ask Spark SQL to **compile each query to Java byte code** before executing it.
- This codegen option could make **long queries or repeated queries** substantially **faster**, as Spark generates specific code to run them.
- For **short queries or some non-repeated ad-hoc queries**, this option could add **unnecessary overhead**, as Spark has to run a compiler for each query.
- It's recommended to use **codegen** option for workflows which involves **large queries**, or with the same **repeated query**.

Configure Spark Properties

- **spark.sql.inMemoryColumnarStorage.batchSize**

```
session = SparkSession.builder \
    .config("spark.sql.inMemoryColumnarStorage.batchSize", value = 1000) \
    .getOrCreate()
```

- When caching dataframe, Spark groups together the **records in batches** of the size given by this option and compresses each batch.
- The default batch size is 1000.
- Having a larger batch size can **improve memory utilization and compression.**
- A batch with a **large number of records** might be hard to build up in memory and can lead to an **OutOfMemoryError**.

Introduction to
Running Spark in a Cluster

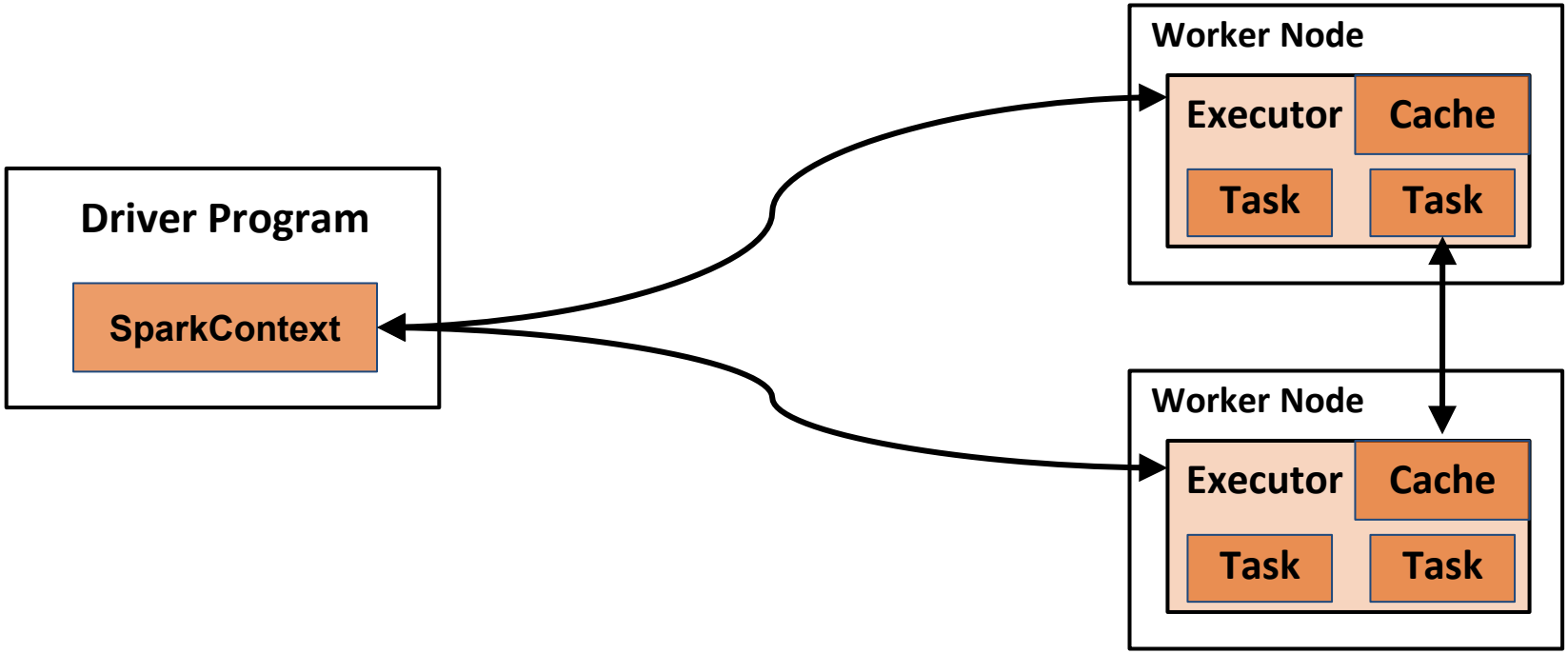
LEVEL UP
www.level-up.one

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

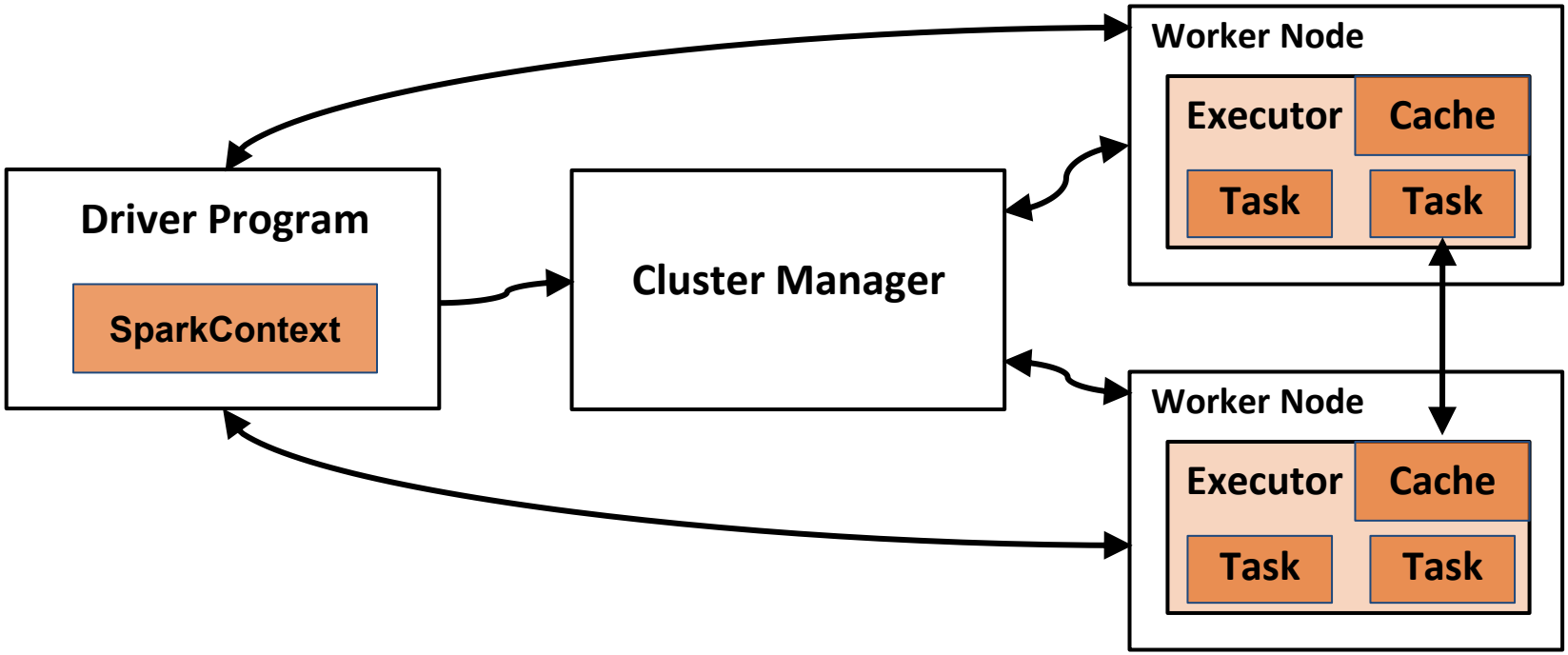
- Writing applications for **parallel cluster execution** uses the **same API** we have already learned in this course.

- 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.

Running Spark in the Cluster Mode



Running Spark in the Cluster Mode



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.
- You can use **spark-submit** to submit an application to the cluster

spark-submit

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 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**.

spark-submit options

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

Run Spark Application on Amazon EMR cluster

(Elastic MapReduce)

LEVEL UP
www.level-up.one

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 data 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.

AWS is NOT free

- AWS **charges** by how much time and how many machines are running with given type, any storage space, etc.