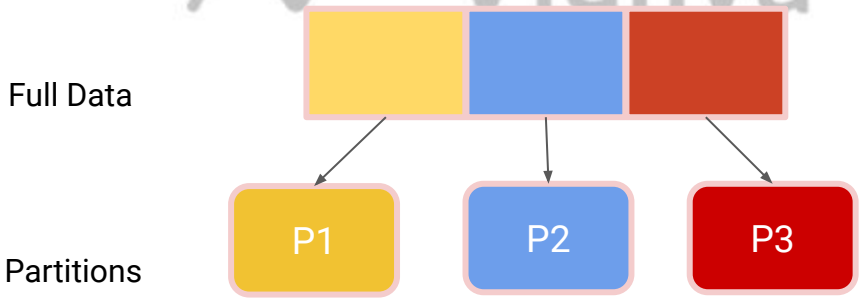
Partitioning

What is Partitioning?

- Spark will split the data into smaller parts called **Partitions**.
- Each of these is then sent to an Executor to be processed.
- Only one partition is computed per executor thread at a time



Partitions

Properties of partitions:

1. Tuples in the same partition are guaranteed to be in the same partition.
2. Each node in the cluster contains at least one or more partitions.
3. The number of partitions is configurable. By default, it equals total number of cores on all available executor nodes.

Partitioning by Example

```
siddharth@siddharth:~$ spark-shell
20/12/30 22:41:11 WARN Utils: Your hostname, siddharth resolves to a loopback address: 127.0.1.1; using 192.
20/12/30 22:41:11 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/opt/spark/spark-3.0.1-bin-hado
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
20/12/30 22:41:12 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using bui
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://192.168.29.93:4040
Spark context available as 'sc' (master = local[*], app id = local-1609348276991).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/

Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM, Java 11.0.9.1)
Type in expressions to have them evaluated.
Type :help for more information.

scala>  val list = List(1,2,3,4,5,6,7,8,9,10)
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val Rdd = sc.parallelize(list)
Rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:26

scala> // number of partitions made by spark

scala> Rdd.partitions.size
res0: Int = 8

scala> //To get the number of cores of the machine

scala> Runtime.getRuntime().availableProcessors()
res1: Int = 8

scala>
```

Partitions are equal to the number of cores on all available executor nodes

Analytics Vidhya

Kinds of Partitioning

Kinds of partitioning available in Spark:

1. Hash Partitioning
2. Range Partitioning

Customization is only available on a paired RDD.

Hash Partitioning

Hash Partitioning attempts to spread the data evenly across various partitions based on the key. ... hashCode method is used to determine the **partition** in **Spark** as

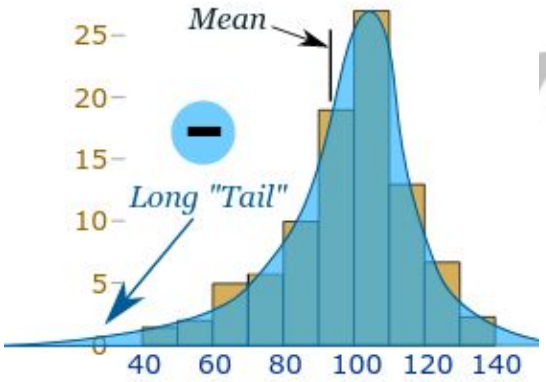
partition = key.hashCode() % numPartitions

```
example = sc.parallelize([(0, u'D'), (0, u'D'), (1, u'E'), (2, u'F')])
example.groupByKey()
```

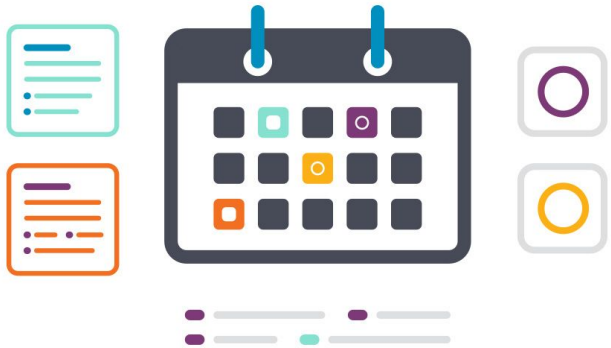

Hash partitioner tries to spread data evenly across the cluster based on the key

Problems - Hash Partitioner

There are 2 issues that arise with the Hash Partitioner:



Data Skew



Scheduling

Data Skew

Consider a paired RDD with keys [8, 96, 240, 400, 401, 800] and a desired number of partitions of 4.

In this case, hash partitioning distributes as follows amongst the partitions:

- **Partition 0:** [8, 96, 240, 400, 800]
- **Partition 1:** [401]
- **Partition 2:**
- **Partition 3:**

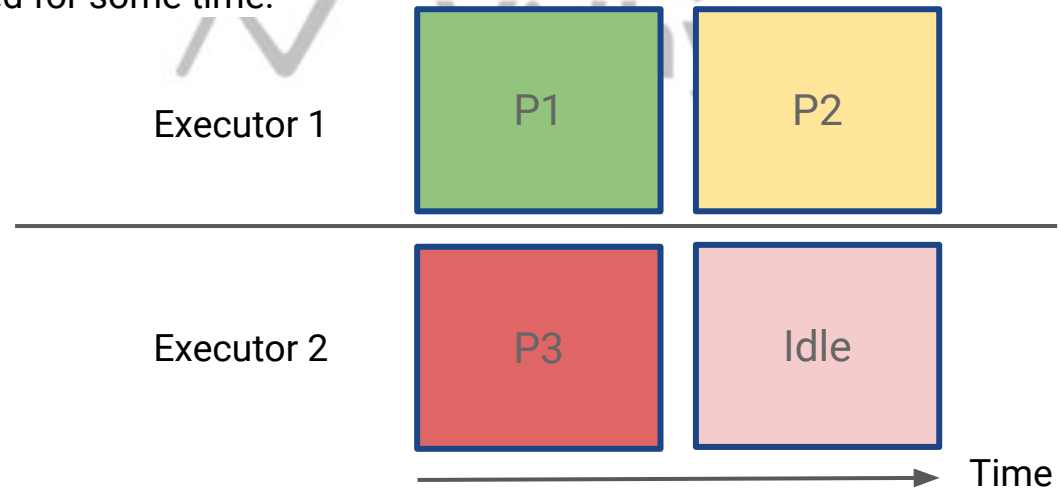
partition = key.hashCode() % numPartitions

The result is very unbalanced distribution of keys which can hurt performance.

Scheduling

The other problem that may occur when splitting into partitions is that there are too few partitions to correctly cover the number of executors available.

An example is given in the diagram below, in which there are 2 Executors and 3 partitions. Executor 1 has an extra partition to compute and therefore takes twice as long as Executor 2. This results in Executor 2 being idle and unused for some time.



Range Partitioning

- **Range partition** algorithm divides the dataset into multiple partitions of consecutive and not overlapping ranges of values.
- **Range partitioning** helps you group similar items inside the same place.
- **For example,** if we partition data on **age**, we could suppose that the users of the same age will like similar type of music, will have similar professional responsibilities and so forth.

Tuples with keys in the same range appear on the same machine.

Range Partitioning: Example

Using Range partitioning can improve performance significantly:

- Assumptions: (a) Keys are non negative (b) 800 is the biggest key in the RDD
- Set of Ranges: [1, 200], [201, 400], [401, 600], [601, 800]

In this case range partitioner distributes the keys as below in partitions:

- **Partition 0:** [8, 96]
- **Partition 1:** [240, 400]
- **Partition 2:** [401]
- **Partition 3:** [800]

The resulting distribution of keys is much more balanced.

Thank You!!

Analytics
Vidhya