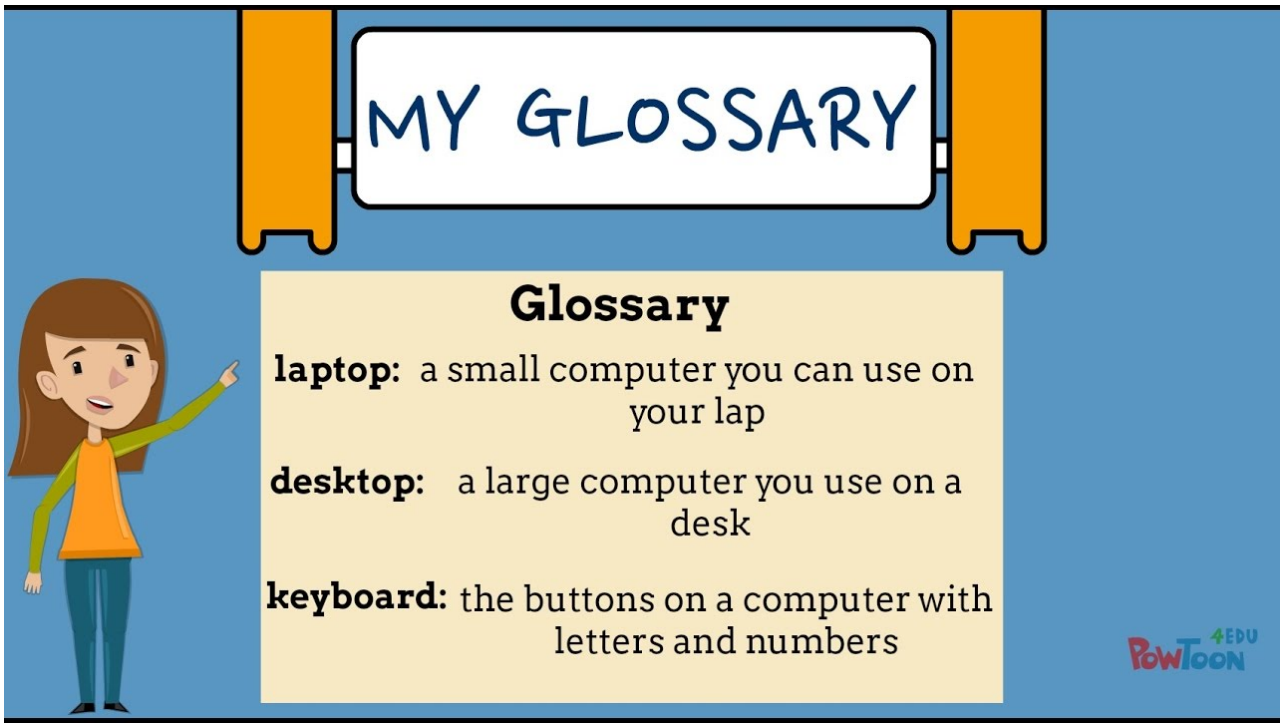
Glossary of Big Data Terms

```
Compiled by: Mahmoud Parsian

Last updated: 1/13/2023
```



Introduction

Big data is a vast and complex field that is constantly evolving, and for that reason, it's important to understand the basic common terms and the more technical vocabulary so that your understanding can evolve with it.

Big data environment involves many tools and technologies:

- Data preparation from multiple sources
- Engine for large-scale data analytics (such as Spark)
- ETL processes to analyze prepare data for Query engine
- Relational database systems

- Query engines such as Amazon Athena, Google BigQuery, Snowflake
- + much more...

The purpose of this glossary is to shed some light on the fundamental definitions of big data and MapReduce, and Spark. This document is a list of terms, words, and concepts found in or relating to big data, MapReduce, and Spark.

Algorithm

- A mathematical formula that can perform certain analyses on data
- An algorithm is a procedure used for solving a problem or performing a computation.
- An algorithm is a set of well-defined steps to solve a problem
- For example, given a set of words, sort them in ascending order
- For example, given a set of text documents, find frequecy of every unique word
- For example, given a set of numbers, find (minimum, maximum) of given numbers

Typically an algorithm is implemented using a programming language such as Python, Java, SQL, ...

In big data world, an algorithm can be implemented using a compute engine such as MapReduce and Spark.

In The Art of Computer Programming, a famous computer scientist, Donald E. Knuth, defines an algorithm as a set of steps, or rules, with five basic properties:

> 1) Finiteness. An algorithm must always terminate after a finite number of steps.
>
> 2) Definiteness. Each step of an algorithm must be precisely defined
>
> 3) Input. An algorithm has zero or more inputs
>
> 4) Output. An algorithm has one or more outputs
>
> 5) Effectiveness. An algorithm is also generally expected to be effective

Distributed algorithm

A distributed algorithm is an algorithm designed to run on computer hardware constructed from interconnected processors. Distributed algorithms are used in different application areas of distributed computing, such as DNA analysis, telecommunications, scientific computing, distributed information processing, and real-time process control. Standard problems solved by

distributed algorithms include leader election, consensus, distributed search, spanning tree generation, mutual exclusion, finding association of genes in DNA, and resource allocation. Distributed algorithms run in parallel/concurrent environments.

Apache Spark can be used to implement and run distributed algorithms.

In implementing distributed algorithms, you have to make sure that your aggregations and reductions are semantically correct (since these are executed partition by partition) regardless of the number of partitions for your data. For example, you need to remember that average of an average is not an average.

Aggregation

- A process of searching, gathering and presenting data.
- Data aggregation refers to the process of collecting data and presenting it in a summarised format. The data can be gathered from multiple sources to be combined for a summary.

Data Aggregation

Data aggregation refers to the collection of data from multiple sources to bring all the data together into a common athenaeum for the purpose of reporting and/or analysis.

- Data aggregation is the process of compiling typically some large amounts of information from a given database and organizing it into a more consumable and comprehensive medium.
- For example, find average age of customer by product
- For example, find median rating for movies rated last year

Analytics

- The discovery of insights in data, find interesting patterns in data
- For example, given a graph, find (identify) all of the triangles
- For example, given a DNA data, find genes, which are assocaited with each other

Anonymization

- Making data anonymous; removing all data points that could lead to identify a person
- For example, replacing social security numbers with fake 18 digit numbers

- For example, replacing patient name with fake ID.

API

- An Application Programming Interface is a set of function definitions, protocols, and tools for building application software
- For example, MapReduce paradigm provides `map()` and `reduce()` functions
- For example, Apache Spark provides
  - `map()`, `flatMap()`, `filter()` and `mapPartitions()` transformations
  - reducers: `groupByKey()`, `reduceByKey()`, `combineByKey()`

Application

- A computer software that enables a computer to perform a certain task
- For example, a payroll application, which issues monthly checks to employees
- For example, a MapReduce application, which identifies duplicate records
- For example, an Spark application, which finds close and related communities in a given graph
- For example, an Spark application, which finds rare variants for DNA samples

Data sizes

- Bit: `0` or `1`
- Byte: 8 bits (`00000000 .. 11111111`) : can represent 256 combinations (0 to 255)
- KB = Kilo Byte = 1,024 bytes = 2^{10} bytes ~ 1000 bytes
- MB = Mega Byte = 1,024 x 1,024 bytes = 1,048,576 bytes ~ 1000 KB
- GB = Giga Byte = 1,024 x 1,024 x 1,024 bytes = 1,073,741,824 bytes ~ 1000 MB
- TB = Tera Byte = 1,024 x 1,024 x 1,024 x 1024 bytes = 1,099,511,627,776 bytes ~ 1000 GB
- PB = Peta Byte = 1,024 x 1,024 x 1,024 x 1024 x 1024 bytes = 1,125,899,906,842,624 bytes ~ 1000 TB
- EB = Exa Byte = 1,152,921,504,606,846,976 (= 2^{60}) bytes ~ 1000 PB
- ZB = Zetta Byte = 1,208,925,819,614,629,174,706,176 bytes (= 2^{80}) bytes

~ denotes "about"

Behavioural Analytics

- It is a kind of analytics that informs about the how, why and what instead of just the who

and when. It looks at humanized patterns in the data

Big Data

Big data is an umbrella term for any collection of data sets so large or complex that it becomes difficult to process them using traditional data-processing applications. In a nutshell, big data refers to data that is so large, fast or complex that it's difficult or impossible to process using traditional methods. Also, big data deals with accessing and storing large amounts of information for analytics.

So, what is Big Data? Big Data is a large data set with increasing volume, variety and velocity.

Big data solutions may have many components (to mention some):

- Distributed File System
- Analytics Engine (such as Spark)
- Query Engine (Such as Snowflake, Amazon Athena, ...)
- ETL Support
- Relational database systems
- ...

Big Data Platforms/Solutions

- Apache Hadoop, which implements a MapReduce paradigm. Hadoop is slow and very complex (does not take advantage of RAM/memory)
- Apache Spark, which implements a superset of MapReduce paradigm: it is fast, and has a very simple and powerful API and works about 100 times faster than Hadoop (Spark takes advantage of memory and embraces in-memory computing). Spark can be used for ETL and implementing many types of distributed algorithms.
- Apache Tez
- Amazon Athena (mainly used as a query engine)
- Snowflake (mainly used as a query engine)
- Google BigQuery: is a serverless and multicloud data warehouse designed to help you turn big data into valuable business insights

Biometrics

The use of data and technology to identify people by one or more of their physical traits (for example, face recognition)

Data modelling

The analysis of data sets using data modelling techniques to create insights from the data:

- data summarization,
- data aggregation,
- joining data

Data set

A collection of (structured, semi-structured, and unstructured) data

Data Type

In computer science and computer programming, a **data type** (or simply type) is a set of possible values and a set of allowed operations on it. A data type tells the compiler or interpreter how the programmer intends to use the data.

For example,

- Java is a strongly typed (strong typing means that the type of a value doesn't change in unexpected ways) language, every variable must be defined by an explicit data type before usage. Java is considered strongly typed because it demands the declaration of every variable with a data type. Users cannot create a variable without the range of values it can hold.
    - Java example

```
// bob's data type is int
int bob = 1;

// bob can not change its type
// String bob = "bob";

// but, you can use anther variable name
String bob_name = "bob";
```

- Python is strongly, dynamically typed:
    - Strong typing means that the type of a value doesn't change in unexpected ways. A

string containing only digits doesn't magically become a number, as may happen in Perl. Every change of type requires an explicit conversion.

- Dynamic typing means that runtime objects (values) have a type, as opposed to static typing where variables have a type.

  - Python example

    ```
    # bob's data type is int
    bob = 1

    # bob's data type changes to str
    bob = "bob"
    ```

    This works because the variable does not have a type; it can name any object. After `bob=1`, you'll find that `type(bob)` returns int, but after `bob="bob"`, it returns `str`. (Note that type is a regular function, so it evaluates its argument, then returns the type of the value.)

Primitive data type

A data type that allows you to represent a single data value in a single column position. In a nutshell, a primitive data type is either a data type that is built into a programming language, or one that could be characterized as a basic structure for building more sophisticated data types.

- Java examples:

  ```
  int a = 10;
  boolean b = true;
  double d = 2.4;
  String s = "fox";
  ```

- Python examples:

  ```
  a = 10
  b = True
  d = 2.4
  s = "fox"
  ```

Composite data type

In computer science, a composite data type or compound data type is any data type which can be constructed in a program using the programming language's primitive data types.

- Java examples:

```
import java.util.Arrays;
import java.util.List;
...
int[] a = {10, 11, 12};
List<String> names = Arrays.asList("n1", "n2", "n3");
```

- Python examples:

```
a = [10, 11, 12];
names = ("n1", "n2", "n3") # immutable
names = ["n1", "n2", "n3"] # mutable
```

Hadoop

Hadoop is an open-source framework that is built to enable the process and storage of big data across a distributed file system. Hadoop implements MapReduce paradigm, it is slow and complex and uses disk for read/write operations. Hadoop does not take advantage of in-memory computing. Hadoop runs a computing cluster.

Hadoop takes care of running your MapReduce code across a cluster of machines. Its responsibilities include chunking up the input data, sending it to each machine, running your code on each chunk, checking that the code ran, passing any results either on to further processing stages or to the final output location, performing the sort that occurs between the map and reduce stages and sending each chunk of that sorted data to the right machine, and writing debugging information on each job’s progress, among other things.

Hadoop provides:

- MapReduce: you can run MapReduce jobs
- HDFS: Hadoop Distributed File System

What is the difference between Hadoop and

RDBMS?

- Hadoop is an implementation of MapReduce paradigm
- RDBMS denoes a relational database system such as Oracle, MySQL, Maria

Criteria	Hadoop	RDBMS
Data Types	Processes semi-structured and unstructured data	Processes structured data
Schema	Schema on Read	Schema on Write
Best Fit for Applications	Data discovery and Massive Storage/Processing of Unstructured data.	Best suited for OLTP and ACID transactions
Speed	Writes are Fast	Reads are Fast
Data Updates	Write once, Read many times	Read/Write many times
Data Access	Batch	Interactive and Batch
Data Size	Tera bytes to Peta bytes	Giga bytes to Tera bytes

What makes Hadoop Fault tolerant?

Hadoop is said to be highly fault tolerant. Hadoop achieves this feat through the process of data replication. Data is replicated across multiple nodes in a Hadoop cluster. The data is associated with a replication factor, which indicates the number of copies of the data that are present across the various nodes in a Hadoop cluster. For example, if the replication factor is 4, the data will be present in four different nodes of the Hadoop cluster, where each node will contain one copy each. In this manner, if there is a failure in any one of the nodes, the data will not be lost, but can be recovered from one of the other nodes which contains copies or replicas of the data.

If replication factor is `N`, then `N-1` nodes can safely fail without impacting a running job.

Big Data Formats

Data comes in many varied formats:

- Avro

    - Avro stores the data definition in JSON format making it easy to read and interpret

- Parquet
    - Parquet is an open source, binary, column-oriented data file format designed for efficient data storage and retrieval

- ORC
    - The Optimized Row Columnar (ORC) file format provides a highly efficient way to store Hive data.

- Text files (log data, CSV, ...)
- XML
- JSON
- ...

Parquet Files

A columnar file format that supports block level compression and is optimized for query performance as it allows selection of 10 or less columns from from 50+ columns records.

Apache Spark can read/write from/to Parquet data format.

Tez

Apache Tez (which implements MapReduce paradigm) is a framework to create high performance applications for batch and data processing. YARN of Apache Hadoop coordinates with it to provide the developer framework and API for writing applications of batch workloads.

The Tez is aimed at building an application framework which allows for a complex directed-acyclic-graph (DAG) of tasks for processing data. It is currently built atop Apache Hadoop YARN.

HBase

HBase is n open source, non-relational, distributed database running in conjunction with Hadoop

HDFS

Hadoop Distributed File System; a distributed file system designed to run on commodity

hardware. You can place huge amount of data in HDFS. You can create new files/directories. You can delete files, but you can not edit/update files in place.

Commodity server/hardware

Commodity hardware (computer), sometimes known as off-the-shelf server/hardware, is a computer device or IT component that is relatively inexpensive, widely available and basically interchangeable with other hardware of its type. Since commodity hardware is not expensive, it is used in building/creating clusters for big data computing (scale-out architecture). Commodity hardware is often deployed for high availability and disaster recovery purposes.

Fault Tolerance and Data Replication.

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance.

Block size can be configured. For example, let block size to be 512MB. Now, let's place a file (sample.txt) of 1800MB in HDFS:

```
1800MB = 512MB (Block-1) + 512MB (Block-2) + 512MB (Block-3) + 264MB (Block-4)
Lets denote
           Block-1 by B1
           Block-2 by B2
           Block-3 by B3
           Block-4 by B4
```

Note that the last block has only 264MB of useful data.

Let's say, we have a cluster of 6 nodes (one master and 5 worker nodes {W1, W2, W3, W4, W5} and master does not store any data), also assume that the replication factor is 2, therefore, blocks will be placed as:

```
    W1: B1, B4
    W2: B2, B3
    W3: B3, B1
    W4: B4
    W5: B2
```

Fault Tolerance: if replication factor is `N`, then `(N-1)` nodes can safely fail without a job fails.

High-Performance-Computing (HPC)

Using supercomputers to solve highly complex and advanced computing problems. This is a scale-up architecture and not a scale-out architecture.

Hadoop and Spark use scale-out architectures.

History of MapReduce

MapReduce was developed by Google back in 2004 by Jeffery Dean and Sanjay Ghemawat of Google (Dean & Ghemawat, 2004). In their paper, “MAPREDUCE: SIMPLIFIED DATA PROCESSING ON LARGE CLUSTERS,” and was inspired by the `map()` and `reduce()` functions commonly used in functional programming. At that time, Google’s proprietary MapReduce system ran on the Google File System (GFS). Apache Hadoop is an open-source implementation of Google's MapReduce.

MapReduce

Mapreduce is a software framework for processing vast amounts of data. MapReduce is a parallel programming model for processing data on a distributed system. MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

In a nutshell, MapReduce provides 3 functions to analyze huge amounts of data:

- `map()` provided by programmer: process the records of the data set:

```
# key: partition number of record number, which might be ignored
# or the “key” might refer to the offset address for each record
# value : an actual input record
map(key, value) -> {(K2, V2), ...}

NOTE: If a mapper does not emit any (K2, V2), then it
means that the input record is filtered out.
```

- `reduce()` provided by programmer: merges the output from mappers:

```
# key: unique key as K2
# values : [v1, v2, ...], values associated by K2
# the order of values {v1, v2, ...} are undefined.
reduce(key, values) -> {(K3, V3), ...}

NOTE: If a reducer does not emit any (K3, V3), then it
means that the key (as K2) is filtered out.
```

- `combine()` provided by programmer [optional]
    - Mini-Reducer
    - Optimizes the result sets from the mappers before sending them to the reducers

The genie/magic of MapReduce is a Sort & Shuffle phase (provided by MapReduce implementation), which groups keys generated by all mappers. For example, if all mappers have created the following (key, value) pairs:

```
(C, 4), (C, 5),
(A, 2), (A, 3),
(B, 1), (B, 2), (B, 3), (B, 1),
(D, 7)
```

then Sort & Shuffle phase creates the following (key, value) pairs (not in any particular order) to be consumed by reducers:

```
(A, [2, 3])
(B, [1, 2, 3, 1])
(C, [4, 5])
(D, [7])
```

Options for MapReduce implementation:

- Hadoop (slow and complex) is an implementation of MapReduce.
- Spark (fast and simple) is a superset implementation of MapReduce.

What is an Example of a Mapper in MapReduce

Imagine that you have records, which describe values for genes and each record is identified as:

```
<gene_id><,><value_1><,><value_2>
```

Sample records might be:

```
INS,1.1,1.4
INSR,1.7,1.2
```

Suppose the goal is to find the median value for the smaller of the two gene values. Therefore we need to produce (key, value) pairs such that key is a `gene_id` and value is minimum of `<value_1>` and `<value_2>`.

The following pseudo-code will accomplish the mapper task:

```
# key: record number or offset of a record number
# key will be ignored since we do not need it
# value: an actual record with the format of:
# <gene_id><,><value_1><,><value_2>
map(key, value) {
   # tokenize input record
   tokens = value.split(",")
   gene_id = tokens[0]
   value_1 = double(tokens[1])
   value_2 = double(tokens[2])
   minimum = min(value_1, value_2)
   # now emit output of the mapper:
   emit(gene_id, minimum)
}
```

For example, if we had the following input:

```
INS,1.3,1.5
INS,1.1,1.4
INSR,1.7,1.2
INS,1.6,1.0
INSR,0.7,1.2
```

Then output of mappers will be:

```
(INS, 1.3)
(INS, 1.1)
(INSR, 1.2)
(INS, 1.0)
(INSR, 0.7)
```

Note that, for the preceding mappers output, the Sort & Shuffle phase will produce the follwong (key, values) pairs to be consumed ny the reducers.

```
(INS, [1.3, 1.1, 1.0])
(INSR, [1.2, 0.7])
```

What is an Example of a Reducer in MapReduce

Imagine that mappers have produced the following output: (key, value) where key is a gene_id and value is an associated gene value:

```
(INS, 1.3)
(INS, 1.1)
(INSR, 1.2)
(INS, 1.0)
(INSR, 0.7)
```

Note that, for the preceding mappers output, the Sort & Shuffle phase will produce the follwong (key, values) pairs to be consumed by the reducers.

```
(INS, [1.3, 1.1, 1.0])
(INSR, [1.2, 0.7])
```

Now, assume that the goal of reducers is to find the median of values per key (as a gene_id). For simplicity, we assume that there exists a `median()` function, which accepts a list of values and computes the median of given values.

```
# key: a unique gene_id
# values: Iteable<Double> (i.e., as a list of values)
reduce(key, values) {
   median_value = median(values)
   # now output final (key, value)
   emit(key, median_value)
}
```

Therefore, with this reducer, reducers will create the following (key, value) pairs:

```
(INS, 1.1)
(INSR, 0.95)
```

What is an Example of a Combiner in MapReduce

Consider a classic word count program in MapReduce. Let's Consider 3 partitions with mappers output:

```
Partition-1   Partition-2    Partition-3
===========   ===========    ===========
(A, 1)        (A, 1)         (C, 1)
(A, 1)        (B, 1)         (C, 1)
(B, 1)        (B, 1)         (C, 1)
(B, 1)        (C, 1)         (C, 1)
(B, 1)                       (B, 1)
```

Without a combiner, Sort & Shuffle will output the following (for all partitions):

```
(A, [1, 1, 1])
(B, [1, 1, 1, 1, 1, 1])
(C, [1, 1, 1, 1, 1])
```

With a combiner, Sort & Shuffle will output the following (for all partitions):

```
(A, [2, 1])
(B, [3, 2, 1])
(C, [1, 4])
```

As you can see, with a combiner, values are combined for the same key on a partition-by-partition basis. In MapReduce, combiners are mini-reducer optimizations and they reduce network traffic by combining many values into a single value.

How does MapReduce work?

A MapReduce system (an implementation of MapReduce mpdel) is usually composed of three steps (even though it's generalized as the combination of Map and Reduce operations/functions). The MapReduce operations are:

- **Map**: The input data is first split (partitioned) into smaller blocks. For example, the Hadoop framework then decides how many mappers to use, based on the size of the data to be processed and the memory block available on each mapper server. Each block is then assigned to a mapper for processing. Each 'worker' node applies the map function to the local data, and writes the output to temporary storage. The primary (master) node ensures that only a single copy of the redundant input data is processed.

  ```
  map(key, value) -> { (K2, V2), ...}
  ```

- **Shuffle, combine and partition**: worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node. As an optional process the combiner (a reducer) can run individually on each mapper server to reduce the data on each mapper even further making reducing the data footprint and shuffling and sorting easier. Partition (not optional) is the process that decides how the data has to be presented to the reducer and also assigns it to a particular reducer. Sort & Shuffle output (note that mappers have created `N` unique keys -- such as K2):

  ```
  (key_1, [V_11, V_12, ...])
  ...
  (key_N, [V_N1, V_N2, ...])
  ```

- **Reduce**: A reducer cannot start while a mapper is still in progress. Worker nodes process each group of (key, value) pairs output data, in parallel to produce (key,value) pairs as output. All the map output values that have the same key are assigned to a single reducer, which then aggregates the values for that key. Unlike the map function which is mandatory to filter and sort the initial data, the reduce function is optional.

Word Count in MapReduce

Given a set of text documents (as input), Word Count algorithm finds frequencies of unique words in input. The `map()` and `reduce()` functions are provided as a **pseudo-code**.

- Mapper function

```
# key: partition number, record number, offset in input file, ignored.
# value: an actual input record
map(key, value) {
  words = value.split(" ")
  for w in words {
     emit(w, 1)
  }
}
```

- Reducer function (long version)

```
# key: a unique word
# values: Iterable<Integer>
reduce(key, values) {
  total = 0
  for n in values {
     total += n
  }
  emit(key, total)
}
```

- Reducer function (short version)

```
# key: a unique word
# values: Iterable<Integer>
reduce(key, values) {
  total = sum(values)
  emit(key, total)
}
```

- Combiner function (short version)

```
# key: a unique word
# values: Iterable<Integer>
combine(key, values) {
  total = sum(values)
  emit(key, total)
}
```

Finding Average in MapReduce

Given a set of gene*id(s) and gene*value(s) (as input), the average algorithm finds average of gene values per gene_id for canceric genes. Assume that the input is formatted as:

```
<gene_id_as_string><,><gene_value_as_double><,><cancer-or-benign>

where <cancer-or-benign> has value as {"cancer", "benign"}
```

The `map()` and `reduce()` functions are provided as a **pseudo-code**.

- Mapper function

```
# key: partition number, record number, offset in input file, ignored.
# value: an actual input record as:
# <gene_id_as_string><,><gene_value_as_double><,><cancer-or-benign>
map(key, value) {
  tokens = value.split(",")
  gene_id = tokens[0]
  gene_value = tokens[1]
  status = tokens[2]
  if (status == "cancer" ) {
     emit(gene_id, gene_value)
  }
}
```

- Reducer function (long version)

```
# key: a unique gene_id
# values: Iterable<double>
reduce(key, values) {
  total = 0
  count = 0
  for v in values {
     total += v
     count += 1
  }
  avg = total / count
  emit(key, avg)
}
```

- Reducer function (short version)

```
# key: a unique gene_id
# values: Iterable<double>
reduce(key, values) {
  total = sum(values)
  count = len(values)
  avg = total / count
  emit(key, avg)
}
```

To have a combiner function, we have to change the output of mappers (since avg of avg is not an avg). This means that avg function is a commutative, but not assocaitive. Changing output of mappers will make it commutative and associative.

Commutative means that:

```
avg(a, b) = avg(b, a)
```

Associative means that:

```
avg(avg(a, b), c) = avg(a, avg(b, c))
```

For details on commutative and associative properties refer to Data Aldorithms with Spark.

- Revised Mapper function

```
# key: partition number, record number, offset in input file, ignored.
# value: an actual input record as:
# <gene_id_as_string><,><gene_value_as_double><,><cancer-or-benign>
map(key, value) {
  tokens = value.split(",")
  gene_id = tokens[0]
  gene_value = tokens[1]
  status = tokens[2]
  if (status == "cancer" ) {
     # revised mapper output
     emit(gene_id, (gene_value, 1))
  }
}
```

- Combiner function

```
# key: a unique gene_id
# values: Iterable<(double, Integer)>
combine(key, values) {
  total = 0
  count = 0
  for v in values {
     # v = (double, integer)
     # v = (sum, count)
     total += v[0]
     count += v[1]
  }
  # note the combiner does not calculate avg
  emit(key, (total, count))
}
```

- Reducer function

```
# key: a unique gene_id
# values: Iterable<(double, Integer)>
combine(key, values) {
  total = 0
  count = 0
  for v in values {
     # v = (double, integer)
     # v = (sum, count)
     total += v[0]
     count += v[1]
  }
  # calculate avg
  avg = total / count
  emit(key, avg)
}
```

What is an Associative Law

An associative operation:

```
f: X x X -> X
```

is a binary operation such that for all `a, b, c` in `X`:

```
f(a, f(b, c)) = f(f(a, b), c)
```

For example, + (addition) is an associative function because

```
(a + (b + c)) = ((a + b) + c)
```

For example, * (multiplication) is an associative function because

```
(a * (b * c)) = ((a * b) * c)
```

While, - (subtraction) is not an associative function because

```
    (4 - (6 - 3) != ((4 - 6) - 3)
         (4 - 3) != (-2 - 3)
               1 != -5
```

While average operation is not an associative function.

```
FACT: avg(1, 2, 3) = 2

avg(1, avg(2, 3)) != avg(avg(1, 2), 3)
    avg(1, 2.5) != avg(1.5, 3)
           1.75 != 2.25
```

What is a Commutative Law

A commutative function `f` is a function that takes multiple inputs from a set X and produces an output that does not depend on the ordering of the inputs. For example, the binary operation `+` is commutative, because `2 + 5 = 5 + 2`. Function `f` is commutative if the following property holds:

```
    f(a, b) = f(b, a)
```

While, - (subtraction) is not an commutative function because

```
    2 - 4 != 4 - 2
       -2 != 2
```

Monoid

Monoids are algebraic structures. A monoid M is a triplet `(X, f, i)`, where

- `X` is a set
- `f` is an associative binary operator
- `i` is an identity element in `X`

The monoid axioms (which govern the behavior of `f`) are as follows.

1. (Closure) For all a, b in X, f(a, b) and f(b, a) is also in X.

2. (Associativity) For all a, b, c in X:

```
f(a, f(b, c)) = f(f(a, b), c)
```

3. (Identity) There is an `i` in `X` such that, for all `a` in `X`:

```
f(a, i) = f(i, a) = a
```

Monoid Examples

Example-1

Let X denotes non-negative integer numbers.

- Let `+` be an addition function, then `M(X, +, 0)` is a monoid.
- Let `*` be an multiplication function, then `M(X, *, 1)` is a monoid.

Example-2

Let `S` denote a set of strings including an empty string (`""`) of length zero, and `||` denote a concatenation operator,

Then `M(S, ||, "")` is a monoid.

Non Monoid Examples

Then `M(X, -, 0)` is not a monoid, since binary subtraction function is not an associative function.

Then `M(X, /, 1)` is not a monoid, since binary division function is not an associative function.

Then `M(X, AVG, 0)` is not a monoid, since `AVG` (an averge function) is not an associative function.

Monoids as a Design Principle for Efficient MapReduce Algorithms

According to Jimmy Lin: "it is well known that since the sort/shuffle stage in MapReduce is costly, local aggregation is one important principle to designing efficient algorithms. This short paper represents an attempt to more clearly articulate this design principle in terms of monoids, which generalizes the use of combiners and the in-mapper combining pattern.

For example, in Spark (using PySpark), in a distributed computing environment, we can not write the following transformation to find average of integer numbers per key:

```
# rdd: RDD[(String, Integer)] : RDD[(key, value)]
# The Following Transformation is WRONG
avg_per_key = rdd.reduceByKey(lambda x, y: (x+y) / 2)
```

This will not work, because averge of average is not an average. In Spark, `RDD.reduceByKey()` merges the values for each key using an **associative** and **commutative** reduce function. Average function is not an associative function.

How to fix this problem? Make it a Monoid:

```
# rdd: RDD[(String, Integer)] : RDD[(key, value)]
# convert (key, value) into (key, (value, 1))
# rdd2 elements will be monoidic structures for addition +
rdd2 = rdd.mapValues(lambda v: (v, 1))
# rdd2: RDD[(String, (Integer, Integer))] : RDD[(key, (sum, count))]

# find (sum, count) per key: a Monoid
sum_count_per_key = rdd2.reduceByKey(
  lambda x, y: (x[0]+y[0], x[1]+y[1])
)

# find average per key
# v : (sum, count)
avg_per_key = sum_count_per_key.mapValues(
   lambda v: float(v[0]) / v[1]
)
```

Note that by mapping `(key, value)` to `(key, (value, 1))` we make addition of values such as (sum, count) to be a monoid. Consider the follwing two partitions:

```
Partition-1       Partition-2
(A, 1)            (A, 3)
(A, 2)
```

By mapping `(key, value)` to `(key, (value, 1))`, we will have (as `rdd2`):

```
Partition-1       Partition-2
(A, (1, 1))        (A, (3, 1))
(A, (2, 1))
```

Then `sum_count_per_key` RDD will hold:

```
Partition-1       Partition-2
(A, (3, 2))        (A, (3, 1))
```

Finally, `avg_per_key` RDD will produce the final value per key: `(A, 2)`.

What Does it Mean that "Average of Average is Not an Average"

In distributed computing environments (such as MapReduce, Hadoop, Spark, ...) correctness of algorithms are very very important. Let's say, we have only 2 partitions:

```
Partition-1       Partition-2
(A, 1)            (A, 3)
(A, 2)
```

and we want to calculate the average per key. Looking at these partitions, the average of (1, 2, 3) will be exactly 2.0. But since we are ina distributed environment, then the average will be calculated per partition:

```
Partition-1: avg(1, 2) = 1.5
Partition-2: avg(3) = 3.0

avg(Partition-1, Partition-2) = (1.5 + 3.0) / 2 = 2.25

===> which is NOT the correct average we were expecting.
```

To fix this problem, we can change the output of mappers: new revised output is as: `(key, (sum, count))`:

```
Partition-1       Partition-2
(A, (1, 1))       (A, (3, 1))
(A, (2, 1))
```

Now, let's calculate average:

```
Partition-1: avg((1, 1), (2, 1)) = (1+2, 1+1) = (3, 2)
Partition-2: avg((3, 1)) = (3, 1)
avg(Partition-1, Partition-2) = avg((3,2), (3, 1))
                              = avg(3+3, 2+1)
                              = avg(6, 3)
                              = 6 / 3
                              = 2.0
                              ===> CORRECT AVERAGE
```

Advantages of MapReduce

Is there any benefit in using MapReduce paradigm? With MapReduce, developers do not need to write code for parallelism, distributing data, or other complex coding tasks because those are already built into the model. This alone shortens analytical programming time.

The following are advantages of MapReduce:

- Scalability
- Flexibility
- Security and authentication
- Faster processing of data
- Very simple programming model
- Availability and resilient nature

- Fault tolerance

What is a MapReduce Job

Job – A program is an execution of a Mapper and Reducer across a dataset. A MapReduce job will have the following components:

- Input path: identifies input directories and files
- Output path: identifies a directory where the outputs will be written
- Mapper: a `map()` function
- Reducer: a `reduce()` function
- Combiner: a `combine()` function [optional]

Disadvantages of MapReduce

- Rigid Map and Reduce programming paradigm
  - low level API
  - must use `map()`, `reduce()` many times to solve a problem
  - join operation is not supported

- Disk I/O (makes it slow)
- Read/Write Intensive (does not utilize in-memory computing)
- Java Focused
  - have to write lots of lines of code to do some simple map and reduce functions
  - API is a low level

What the MapReduce's Job Flow

1-InputFormat: Splits input into `(key_1, value_1)` pairs and passes them to mappers

2-Mapper: `map(key_1, value_1)` emits a set of `(key_2, value_2)` pairs. If a mapper does not emit any `(key, value)` pairs, then it means that `(key_1, value_1)` is filtered out (for example, tossing out the invalid/bad records).

3-Combiner: [optional] `combine(key_2, [value-2, ...])` emits `(key_2, value_22)`. The combiner might emit no (key, value) pair if there is a filtering algorithm (based on the key (i.e., `key_2` and its associated values)).

Note that `value_22` is an aggregated value for `[value-2, ...]`

4-Sort & Shuffle: Group by keys of mappers with their associated values. If output of all mappers/combiners are:

```
(K_1, v_1), (K_1, v_2), (K_1, v_3), ...,
(K_2, t_1), (K_2, t_2), (K_2, t_3), ...,
...
(K_n, a_1), (K_n, a_2), (K_n, a_3), ...
```

Then output of Sort & Shuffle will be (which will be fed as an inut to reducers as `(key, values)`:

```
(K_1, [v_1, v_2, v_3, ...])
(K_2, [t_1, t_2, t_3, ...])
...
(K_n, [a_1, a_2, a_3, ...])
```

5-Reducer: We will have `n` reducers, sicnce we have `n` unique keys. All these reducers can run in parallel (if we have enough resources).

`reduce(key, values)` will emit a set of `(key_3, value_3)` pairs and eventually thay are written to output. Note that reducer key will be one of `{K_1, K_2, ..., K_n}`.

6-OutputForamt: Reponsible for writing `(key_3, value_3)` pairs to output medium. Note that some of the reducers might not emit any `(key_3, value_3)` pairs: this means that the reducer is filtering out some keys based of the associated values (for example, if the median of the values is less than 10, then filter out).

Hadoop vs. Spark

Feature	Haoop	Spark
Data Processing	Provides batch processing	Provides both batch processing and stream processing
Memory usage	Disk-bound	Uses large amounts of RAM
Security	Better security features	Basic security is provided
Fault	Replication is used for fault	RDD and various data storage models

Tolerance	tolerance	are used for fault tolerance.
Graph Processing	Must develop custom algorithms	Comes with a graph computation library called GraphX and external library as GraphFrames
Ease of Use	Difficult to use	Easier to use
Powerful API	Low level API	High level API
Real-time	Batch only	Batch and Interactive and Stream
Interactive data processing	Not supported	Supported by PySpark, ...
Speed	SLOW: Hadoop's MapReduce model reads and writes from a disk, thus it slows down the processing speed.	FAST: Spark reduces the number of read/write cycles to disk and store intermediate data in memory, hence faster-processing speed.
Latency	It is high latency computing framework.	It is a low latency computing and can process data interactively
Machine Learing API	Not supported	Supported by ML Library
Data Source Support	Limited	Extensive
Storage	Has HDFS (Hadoop Distributed File System)	Does not have a storage system, but may use S3 and HDFS and many other data sources and storages
MapReduce	Implements MapReduce	Implements superset of MapReduce and beyond
Join Operation	Does not support Join directly	Has extensive API for Join

Spark

Apache Spark is an engine for large-scale data analytics. Spark is a multi-language (Java, Scala, Python, R, SQL) engine for executing data engineering, data science, and machine learning on single-node machines or clusters. Spark implements superset of MapReduce paradigm and uses memory/RAM as much as possible and can run up to 100 times faster than Hadoop. Spark is considered the successor of Hadoop/Mapreduce and has addressed many problems of Hadoop.

With using Spark, developers do not need to write code for parallelism, distributing data, or other complex coding tasks because those are already built into the spark engine. This alone shortens analytical programming time.

Apache Spark is one of the best alternatives to Hadoop and currently is the defacto standard for big data analytics. Spark offers simple API and provides high-level mappers, filters, and reducers.

Spark's architecture consists of two main components:

- Drivers - convert the user's code into tasks to be distributed across worker nodes
- Executors - run on those nodes and carry out the tasks assigned to them

PySpark is an interface for Spark in Python. PySpark has two main data abstractions:

- RDD (Resilient Distributed Dataset)
    - low-level
    - immutable
    - partitioned
    - can represent billions of data points
    - for unstructured and semi-structured data

- DataFrame
    - high-level
    - immutable
    - partitioned
    - can represent billions of rows with named columns
    - for structured and semi-structured data

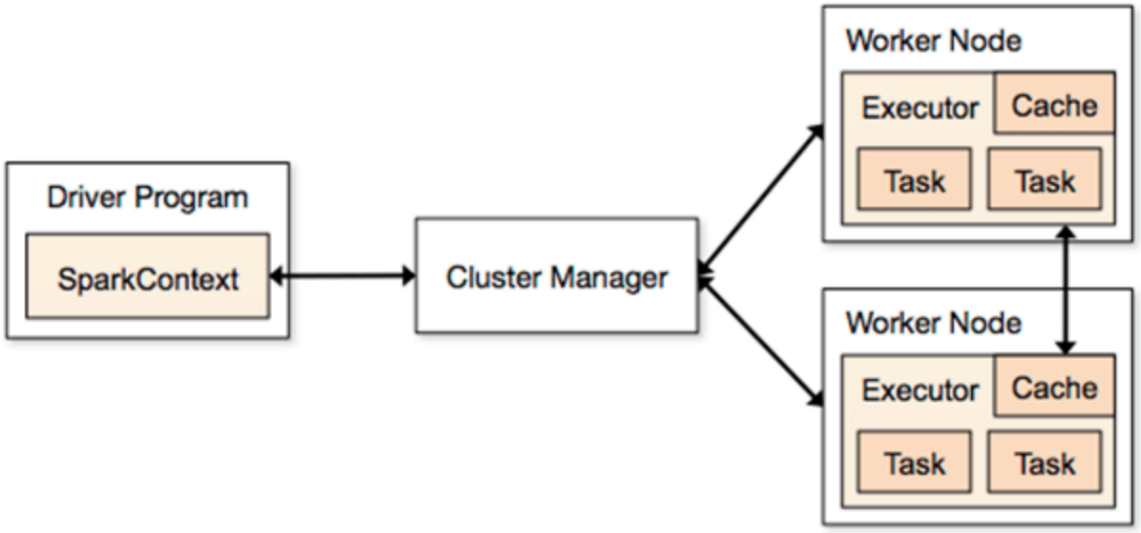
Spark addresses many problems of hadoop:

- provides in-memory computing

- provides simple, powerful, and high-level transformations
- provides join operations for RDDs and DataFrames
- You do not need to write too many lines of a code to solve a big data problem

Spark Concepts and Key Terms

- Spark Architecture



- Spark Cluster: a collection of machines or nodes in the public cloud or on-premise in a private data center on which Spark is installed. Among those machines are Spark workers, a Spark Master (also a cluster manager in a Standalone mode), and at least one Spark Driver.

- Spark Master: As the name suggests, a Spark Master JVM acts as a cluster manager in a Standalone deployment mode to which Spark workers register themselves as part of a quorum. Depending on the deployment mode, it acts as a resource manager and decides where and how many Executors to launch, and on what Spark workers in the cluster.

- Spark Worker: Upon receiving instructions from Spark Master, the Spark worker JVM launches Executors on the worker on behalf of the Spark Driver. Spark applications, decomposed into units of tasks, are executed on each worker's Executor. In short, the worker's job is to only launch an Executor on behalf of the master.

- Spark Executor: A Spark Executor is a JVM container with an allocated amount of cores and memory on which Spark runs its tasks. Each worker node launches its own Spark Executor, with a configurable number of cores (or threads). Besides executing Spark tasks, an Executor also stores and caches all data partitions in its memory.

- Spark Driver: Once it gets information from the Spark Master of all the workers in the cluster and where they are, the driver program distributes Spark tasks to each worker's Executor. The driver also receives computed results from each Executor's tasks.

Spark as a superset of MapReduce

Spark is a true successor of MapReduce and maintains MapReduce's linear scalability and fault tolerance, but extends it in 7 important ways:

1. Spark does not rely on a low-level and rigid `map-then-reduce` workflow. Spark's engine can execute a more general Directed Acyclic Graph (DAG) of operators. This means that in situations where MapReduce must write out intermediate results to the distributed file system (such as HDFS and S3), Spark can pass them directly to the next step in the pipeline. Rather than writing many `map-then-reduce` jobs, in Spark, you can use transformations in any order to have an optimized solution.

2. Spark complements its computational capability with a simple and rich set of transformations and actions that enable users to express computation more naturally. Powerful and simple API (as a set of functions) are provided for various tasks including numerical computation, datetime processing and string manipulation.

3. Spark extends its predecessors (such as Hadoop) with in-memory processing. MapReduce uses disk I/O (which is slow), but Spark uses in-memory computing as much as possible and it can be up to 100 times faster than MapReduce implementations. This means that future steps that want to deal with the same data set need not recompute it or reload it from disk. Spark is well suited for highly iterative algorithms as well as adhoc queries.

4. Spark offers interactive environment (for example using PySpark interactively) for testing and debugging data transformations.

5. Spark offers extensive Machine Learning libraries (Hadoop/MapReduce does not have this capability)

6. Spark offers extensive graph API by GraphX (built-in) and GraphFrames (as an external library).

7. Spark Streaming is an extension of the core Spark API that allows data engineers and data scientists to process real-time data from various sources including (but not limited to) Kafka, Flume, and Amazon Kinesis. This processed data can be pushed out to file systems, databases, and live dashboards.

What is an Spark RDD

Spark's RDD (full name in PySpark as: `pyspark.RDD`) is a Resilient Distributed Dataset (`RDD`), the basic abstraction in Spark. RDD represents an immutable, partitioned collection of elements that can be operated on in parallel. Basically, an RDD represents your data (as a collection, text files, databases, Parquet files, JSON, CSV files, ...). Once your data is represented as an RDD, then you call apply transformations (such as filters, mappers, and reducers) to your RDD and create new RDDs.

An RDD can be created from many data sources such as Python collections, text files, CSV files, JSON, ...

An RDD is more suitable to unstructured and semi-structured data (while a DataFrame is more suitable to structured and semi-structured data.

What is Lineage In Spark?

Spark RDDs are immutable (READ-ONLY) distributed collection of elements of your data that can be stored in memory or disk across a cluster of machines. The data is partitioned across machines in your cluster that can be operated in parallel with a low-level API that offers transformations and actions. RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure.

What are Spark operations/functions?

Two types of Spark RDD operations are: Transformations and Actions.

- Transformation: a transformation is a function that produces new/target RDDs from the source/existing RDDs
    - `Transformation: source_rdd --> target_rdd`
    - `map()`, `filter()`, `flatMap()`, `mapPartitions()`
    - `groupByKey()`, `reduceByKey()`, `combineByKey()`
    - ...
- Action: when we want to work with the actual dataset, at that point Action is performed. For RDD, anction is defined as the Spark operations that return raw values. In other words, any of the RDD functions that return other than the `RDD[T]` is considered an action in the spark programming.

- `Action: source_rdd --> NONE_rdd`
- `collect()`
- `count()`
- ...

The Spark Programming Model

Spark programming starts with a data set (which can be represented as an RDD or a DataFame), usually residing in some form of distributed, persistent storage like Amazon S3 or Hadoop HDFS. Writing a Spark program typically consists of a few related steps:

1. Define a set of transformations on the input data set.
2. Invoke actions that output the transformed data sets to persistent storage or return results to the driver's local memory.
3. Run local computations that operate on the results computed in a distributed fashion. These can help you decide what transformations and actions to undertake next.

What is Lazy Binding In Spark?

Lazy binding/evaluation in Spark means that the execution of **transformations** will not start until an **action** is triggered.

In programming language theory, lazy evaluation, or call-by-need, is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing).

Difference between `reduceByKey()` and `combineByKey()`

- `reduceByKey()`

`RDD.reduceByKey()` merges the values for each key using an **associative** and **commutative** reduce function. This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a "combiner" in MapReduce.

This can be expressed as:

```
    reduceByKey: RDD[(K, V)] --> RDD[(K, V)]
```

- `combineByKey()`

`RDD.combineByKey()` is a generic function to combine the elements for each key using a custom set of aggregation functions. `RDD.combineByKey()` turns an `RDD[(K, V)]` into a result of type `RDD[(K, C)]`, for a "combined type" `C`.

For `combineByKey()`, users provide three functions:

- createCombiner, which turns a `V` into a `C` (e.g., creates a one-element list)

  ```
  createCombiner: V --> C
  ```

- mergeValue, to merge a `V` into a `C` (e.g., adds it to the end of a list)

  ```
  mergeValue: C x V --> C
  ```

- mergeCombiners, to combine two `C's` into a single one (e.g., merges the lists)

  ```
  mergeCombiners: C x C --> C
  ```

This can be expressed as:

```
    combineByKey: RDD[(K, V)] --> RDD[(K, C)]

    where V and C can be the same or different
```

What is an example of `RDD.combineByKey()`?

Combine all of values per key.

```
# combineByKey: RDD[(String, Integer)] --> RDD[(String, [Integer])]

rdd = sc.parallelize([("a", 1), ("b", 7), ("a", 2), ("a", 3), ("b", 8), ("z", 5)])

# V --> C
def to_list(a):
    return [a]

# C x V --> C
def append(a, b):
    a.append(b)
    return a

# C x C --> C
def extend(a, b):
    a.extend(b)
    return a

# rdd: RDD[(String, Integer)]
# rdd2: RDD[(String, [Integer])]
rdd2 = rdd.combineByKey(to_list, append, extend)
rdd2.collect()

[
 ('z', [5]),
 ('a', [1, 2, 3]),
 ('b', [7, 8])
]

# Note that values of keys does not need to be sorted
```

What is an example of `RDD.reduceByKey()`?

Find maximum of values per key.

```
# reduceByKey: RDD[(String, Integer)] --> RDD[(String, Integer)]

rdd = sc.parallelize([("a", 1), ("b", 7), ("a", 2), ("a", 3), ("b", 8), ("z", 5)])

# rdd: RDD[(String, Integer)]
# rdd2: RDD[(String, Integer)]
rdd2 = rdd.reduceByKey(lambda x, y: max(x, y))
rdd2.collect()

[
 ('z', 5),
 ('a', 3),
 ('b', 8)
]
```

What is an example of `RDD.groupByKey()`?

Combine/Group values per key.

```
# reduceByKey: RDD[(String, Integer)] --> RDD[(String, [Integer])]

rdd = sc.parallelize([("a", 1), ("b", 7), ("a", 2), ("a", 3), ("b", 8), ("z", 5)])

# rdd: RDD[(String, Integer)]
# rdd2: RDD[(String, [Integer])]
rdd2 = rdd.groupByKey()
rdd2.collect()

[
 ('z', [5]),
 ('a', [1, 2, 3]),
 ('b', [7, 8])
]
```

What is a DataFrame?

A DataFrame is a data structure that organizes data into a 2-dimensional table of rows and

columns, much like a spreadsheet or a relational table. DataFrames are one of the most common data structures used in modern data analytics because they are a flexible and intuitive way of storing and working with data.

Python DataFrame Example

DataFrame is a 2-dimensional mutable labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used `Pandas` object. A `Pandas` DataFrame is a 2-dimensional data structure, like a 2-dimensional array, or a table with rows and columns. The number of rows for `Pandas` DataFrame is mutable and limited to the computer and memory where it resides.

```
import pandas as pd

data = {
  "calories": [100, 200, 300],
  "duration": [50, 60, 70]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)

print(df)

# Result:

     calories  duration
  0       100        50
  1       200        60
  2       300        70
```

Spark DataFrame Example

A distributed collection of data grouped into named columns. Spark's DataFrame is immutable and can have billions of rows. A DataFrame is equivalent to a relational table in Spark SQL, and can be created using various functions in `SparkSession` :

```
# PySpark code:

input_path = "..."
# spark: as a SparkSession object
people = spark.read.parquet(input_path)
```

Once created, it can be manipulated using the various domain-specific-language (DSL) functions or you may use `SQL` to execute queries against DataFrame (registered as a table).

A more concrete example:

```
# PySpark code:

# To create DataFrame using SparkSession
input_path_people = "..."
people = spark.read.parquet(input_path_people)
input_path_dept = "..."
department = spark.read.parquet(input_path_dept)

result = people.filter(people.age > 30)\
               .join(department, people.deptId == department.id)\
               .groupBy(department.name, "gender")\
               .agg({"salary": "avg", "age": "max"})
```

What is an Spark DataFrame?

Spark's DataFrame (full name as: `pyspark.sql.DataFrame`) is an immutable and distributed collection of data grouped into named columns. Once your DataFrame is created, then your DataFrame can be manipulated and transformed into another DataFrame by DataFrame's native API and SQL.

A DataFrame can be created from Python collections, relational databases, Parquet files, JSON, CSV files, ...).

DataFrame is more suitable to structured and semi-structured data (while an RDD is more suitable to unstructured and semi-structured data).

Spark RDD Example

An Spark RDD can represent billions of elements.

```
>>> sc
<SparkContext master=local[*] appName=PySparkShell>
>>> sc.version
'3.3.1'
>>> numbers = sc.parallelize(range(0,1000))
>>> numbers
PythonRDD[1] at RDD at PythonRDD.scala:53
>>> numbers.count()
1000
>>> numbers.take(5)
[0, 1, 2, 3, 4]
>>> numbers.getNumPartitions()
16
>>> total = numbers.reduce(lambda x, y: x+y)
>>> total
499500
```

Spark DataFrame Example

A Spark DataFrame can represent billions of rows of named columns.

```
>>> records = [("alex", 23), ("jane", 24), ("mia", 33)]
>>> spark
<pyspark.sql.session.SparkSession object at 0x12469e6e0>
>>> spark.version
'3.3.1'
>>> df = spark.createDataFrame(records, ["name", "age"])
>>> df.show()
+----+---+
|name|age|
+----+---+
|alex| 23|
|jane| 24|
| mia| 33|
+----+---+

>>> df.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
```

Spark partitioning

A partition in spark is an atomic chunk of data (logical division of data) stored on a node in the cluster. Partitions are basic units of parallelism in Apache Spark. RDDs in Apache Spark are collection of partitions.

For example, in PySpark you can get the current number/length/size of partitions by running `RDD.getNumPartitions()`.

GraphFrames

GraphFrames is an external package for Apache Spark which provides DataFrame-based Graphs. It provides high-level APIs in Scala, Java, and Python. It aims to provide both the functionality of GraphX (included in Spark API) and extended functionality taking advantage of Spark DataFrames. This extended functionality includes motif finding, DataFrame-based serialization, and highly expressive graph queries.

> GraphFrames are to DataFrames as GraphX is to RDDs.

To build a graph, you build 2 DataFrames (one for vertices and another one for the edges) and

then glue them together to create a graph:

```
# each node is identified by "id" and an optional attributes
# vertices: DataFrame(id, ...)

# each edge is identified by (src, dst) and an optional attributes
# where src and dst are node ids
# edges: DataFrame(src, dst, ...)

# import required GraphFrame library
from graphframes import GraphFrame

# create a new directed graph
graph = GraphFrame(vertices, edges)
```

Example of a GraphFrame

This example shows how to build a directed graph using graphframes API.

To invoke PySpark with GraphFrames:

```
% # define the home directory for Spark
% export SPARK_HOME=/home/spark-3.2.0
% # import graphframes library into PySpark and invoke interactive PySpark:
% $SPARK_HOME/bin/pyspark --packages graphframes:graphframes:0.8.2-spark3.2-s_2.12
Python 3.8.9 (default, Mar 30 2022, 13:51:17)
...
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Python version 3.8.9 (default, Mar 30 2022 13:51:17)
Spark context Web UI available at http://10.0.0.234:4040
Spark context available as 'sc' (master = local[*], app id = local-1650670391027).
SparkSession available as 'spark'.
>>>
```

Then PySpark is ready to use GraphFrames API:

```
>>># create list of nodes
>>> vert_list = [("a", "Alice", 34),
...              ("b", "Bob", 36),
...              ("c", "Charlie", 30)]
>>>
>>># define column names for a node
>>> column_names_nodes = ["id", "name", "age"]
>>>
>>># create vertices_df as a Spark DataFrame
>>> vertices_df = spark.createDataFrame(
...      vert_list,
...      column_names_nodes
... )

>>>
>>># create list of edges
>>> edge_list = [("a", "b", "friend"),
...              ("b", "c", "follow"),
...              ("c", "b", "follow")]
>>>
>>># define column names for an edge
>>> column_names_edges = ["src", "dst", "relationship"]
>>>
>>># create edges_df as a Spark DataFrame
>>> edges_df = spark.createDataFrame(
...           edge_list,
...           column_names_edges
... )
>>>
>>># import required libriaries
>>> from graphframes import GraphFrame
>>>
>>># build a graph using GraphFrame library
>>> graph = GraphFrame(vertices_df, edges_df)
>>>
>>># examine built graph
>>> graph
GraphFrame(
  v:[id: string, name: string ... 1 more field],
  e:[src: string, dst: string ... 1 more field]
)
>>>
>>># access vertices of a graph
>>> graph.vertices.show()
```

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
+---+-------+---+

>>># access edges of a graph
>>> graph.edges.show()
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  b|  c|      follow|
|  c|  b|      follow|
+---+---+------------+
```

GraphX

GraphX is Apache Spark's API (RDD-based) for graphs and graph-parallel computation, with a built-in library of common algorithms. GraphX has API for Java and Scala, but does not have an API for Python (therefore, PySpark does not support GraphX, but PySpark supports GraphFrames).

Cluster

Cluster is a group of servers on a network that are configured to work together. A server is either a master node or a worker node. A cluster may have a master node and many worker nodes. In a nutshell, a master node acts as a cluster manager.

A cluster may have one (or two) master nodes and many worker nodes. For example, a cluster of 15 nodes: one master and 14 worker nodes. Another example: a cluster of 101 nodes: one master and 100 worker nodes.

A cluster may be used for running many jobs (Spark and MapReduce jobs) at the same time.

Cluster computing

Cluster computing is a collection of tightly or loosely connected computers that work together so

that they act as a single entity. The connected computers execute operations all together thus creating the idea of a single system. The clusters are generally connected through fast local area networks (LANs). A cluster computing is comprised of a one or more masters (manager for the whole cluster) and many worker nodes. For example, a cluster computer may have a single master node (which might not participate in tasks such as mappers and reducers) and 100 worker nodes (which actively participate in carrying tasks such as mappers and reducers). A small cluster might have one master node and 5 worker nodes. Large clusters might have hundreds or thousands of worker nodes.

Concurrency

Performing and executing multiple tasks and processes at the same time. Let's define 5 tasks {T1, T2, T3, T4, T5} where each will take 10 seconds. If we execute these 5 tasks in sequence, then it will take about 50 seconds, while if we execute all of them in parallel, then the whole thing will take about 10 seconds. Cluster computing enables concurrency and parallelism.

Histogram

A graphical representation of the distribution of a set of numeric data, usually a vertical bar graph

Structured data

Structured data — typically categorized as quantitative data — is highly organized and easily decipherable by machine learning algorithms. Developed by IBM in 1974, structured query language (SQL) is the programming language used to manage structured data. By using a relational (SQL) database, business users can quickly input, search and manipulate structured data. In structured data, each record has a precise record format. Structured data is identifiable as it is organized in structure like rows and columns.

Unstructured data

n the modern world of big data, unstructured data is the most abundant. It’s so prolific because unstructured data could be anything: media, imaging, audio, sensor data, log data, text data, and much more. Unstructured simply means that it is datasets (typical large collections of files) that aren’t stored in a structured database format. Unstructured data has an internal structure, but it’s not predefined through data models. It might be human generated, or machine generated

in a textual or a non-textual format. Unstructured data is regarded as data that is in general text heavy, but may also contain dates, numbers and facts.

Correlation analysis

The analysis of data to determine a relationship between variables and whether that relationship is negative `(- 1.00)` or positive `(+1.00)`.

Data aggregation tools

The process of transforming scattered data from numerous sources into a single new one.

Data analyst

Someone analysing, modelling, cleaning or processing data

Database

A digital collection of data stored via a certain technique. In computing, a database is an organized collection of data (rows or objects) stored and accessed electronically.

Database Management System

Collecting, storing and providing access of data.

Data cleansing

The process of reviewing and revising data in order to delete duplicates, correct errors and provide consistency

Data mining

The process of finding certain patterns or information from data sets

Data virtualization

A data integration process in order to gain more insights. Usually it involves databases, applications, file systems, websites, big data techniques, etc.)

De-identification

Same as anonymization; ensuring a person cannot be identified through the data

ETL (Extract, Transform and Load)

ETL is a process in a database and data warehousing meaning extracting the data from various sources, transforming it to fit operational needs and loading it into the database or some storage. For example, processing DNA data, creating output records in specific Parquet format and loading it to Amazon S3 is an ETL process.

- Extract: the process of reading data from a database or data sources
- Transform: the process of conversion of extracted data in the desired form so that it can be put into another database.
- Load: the process of writing data into the target database to data source

Failover

Switching automatically to a different server or node should one fail Fault-tolerant design – a system designed to continue working even if certain parts fail Feature - a piece of measurable information about something, for example features you might store about a set of people, are age, gender and income.

Graph Databases

Graph databases are purpose-built to store and navigate relationships. Relationships are first-class citizens in graph databases, and most of the value of graph databases is derived from these relationships. Graph databases use nodes to store data entities, and edges to store relationships between entities. An edge always has a start node, end node, type, and direction, and an edge can describe parent-child relationships, actions, ownership, and the like. There is no limit to the number and kind of relationships a node can have.

Grid computing

Connecting different computer systems from various location, often via the cloud, to reach a common goal

Key-Value Databases

Key-Value Databases store data with a primary key, a uniquely identifiable record, which makes easy and fast to look up. The data stored in a Key-Value is normally some kind of primitive of the programming language. As a dictionary, for example, Redis allows you to set and retrieve pairs of keys and values. Think of a “key” as a unique identifier (string, integer, etc.) and a “value” as whatever data you want to associate with that key. Values can be strings, integers, floats, booleans, binary, lists, arrays, dates, and more.

(key, value)

The `(key, value)` notation is used in many places (such as Spark) and in MapReduce Paradigm. In MapReduce paradigm everything works as a `(key, value)`. Note that the `key` and `value` can be

- simple data type: such as String, Integer, Double, ...
- combined data types: tuples, structures, arrays, lists, ...

In MapReduce, `map()` and `reduce()` use `(key, value)` pairs:

The Map output types should match the input types of the Reduce as shown below:

```
# mapper can emit 0, 1, 2, ... of (K2, V2)
map(K1, V1) -> { (K2, V2) }

# reducer can emit 0, 1, 2, ... of (K3, V3)
# K2 is a unique key from mapper's outputs
# [V2, ...] are all values associated with key K2
reduce(K2, [V2, ...]) -> { (K3, V3) }
```

In Spark, using RDDs, a source RDD must be in `(key, value)` form before we can apply reduction transformations such as `groupByKey()`, `reduceByKey()`, and `combineByKey()`.

Java

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It has evolved from humble beginnings to power a large share of today's digital world, by providing the reliable platform upon which many services and applications are built. New, innovative products and digital services designed for the future continue to rely on Java, as well.

Python

Python is a programming language that lets you work quickly and integrate systems more effectively. Python is an interpreted, object-oriented (not fully) programming language that's gained popularity for big data professionals due to its readability and clarity of syntax. Python is relatively easy to learn and highly portable, as its statements can be interpreted in several operating systems.

JavaScript

A scripting language designed in the mid-1990s for embedding logic in web pages, but which later evolved into a more general-purpose development language.

In-memory

A database management system stores data on the main memory instead of the disk, resulting is very fast processing, storing and loading of the data Internet of Things – ordinary devices that are connected to the internet at any time anywhere via sensors

Latency

A measure of time delayed in a system

Location data

GPS data describing a geographical location

Machine learning

Part of artificial intelligence where machines learn from what they are doing and become better over time. Apache Spark offers a comprehensive Maching Learning library.

Metadata

Data about data; gives information about what the data is about.

Natural Language Processing

A field of computer science involved with interactions between computers and human languages

Network analysis

Viewing relationships among the nodes in terms of the network or graph theory, meaning analysing connections between nodes in a network and the strength of the ties.

Workflow

A graphical representation of a set of events, tasks, and decisions that define a business process (example: vacation approval process in a company; purchase approval process). You use the developer tool to add objects to a workflow and to connect the objects with sequence flows. The Data Integration Service uses the instructions configured in the workflow to run the objects.

Schema

In computer programming, a schema (pronounced `SKEE-mah`) is the organization or structure for a database, while in artificial intelligence (AI) a schema is a formal expression of an inference rule. For the former, the activity of data modeling leads to a schema.

- Example, Database schema:

```
CREATE TABLE product (
    id INT AUTO_INCREMENT PRIMARY KEY,
    product_name VARCHAR(50) NOT NULL,
    price VARCHAR(7) NOT NULL,
    quantity INT NOT NULL
)
```

- Example, DataFrame schema in PySpark

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import StringType, IntegerType

schema = StructType([ \
    StructField("firs_tname", StringType(),True), \
    StructField("last_name", StringType(),True), \
    StructField("emp_id", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True)
])
```

Difference between Tuple and List in Python

The primary difference between tuples and lists is that tuples are **immutable** as opposed to lists which are **mutable**. Therefore, it is possible to change a list but not a tuple. The contents of a tuple cannot change once they have been created in Python due to the immutability of tuples.

Examples in Python3:

```
# create a tuple
>>> t3 = (10, 20, 40)
>>> t3
(10, 20, 40)

# create a list
>>> l3 = [10, 20, 40]
>>> l3
[10, 20, 40]

# add an element to a list
>>> l3.append(500)
>>> l3
[10, 20, 40, 500]

# add an element to a tuple
>>> t3.append(600)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Object Databases

An object database store data in the form of objects, as used by object-oriented programming. They are different from relational or graph databases and most of them offer a query language that allows object to be found with a declarative programming approach.

Pattern Recognition

Pattern Recognition identifies patterns in data via algorithms to make predictions of new data coming from the same source.

Predictive analysis

Analysis within big data to help predict how someone will behave in the (near) future. It uses a variety of different data sets such as historical, transactional, or social profile data to identify risks and opportunities.

Privacy

To seclude certain data / information about oneself that is deemed personal Public data – public information or data sets that were created with public funding

Query

Asking for information to answer a certain question

Regression analysis

To define the dependency between variables. It assumes a one-way causal effect from one variable to the response of another variable.

Real-time data

Data that is created, processed, stored, analysed and visualized within milliseconds

Scripting

The use of a computer language where your program, or script, can be run directly with no need to first compile it to binary code. Semi-structured data - a form a structured data that does not have a formal structure like structured data. It does however have tags or other markers to enforce hierarchy of records.

Sentiment Analysis

Using algorithms to find out how people feel about certain topics or events

SQL

A programming language for retrieving data from a relational database. Also, SQL is used to retrieve data from big data by translating query into mappers, filters, and reducers.

Time series analysis

Analysing well-defined data obtained through repeated measurements of time. The data has to be well defined and measured at successive points in time spaced at identical time intervals.

Variability

It means that the meaning of the data can change (rapidly). In (almost) the same tweets for example a word can have a totally different meaning

Variety

Data today comes in many different formats: structured data, semi-structured data, unstructured data and even complex structured data

Velocity

The speed at which the data is created, stored, analysed and visualized

Veracity

Ensuring that the data is correct as well as the analyses performed on the data are correct.

Volume

The amount of data, ranging from megabytes to gigabytes to petabytes to ...

XML Databases

XML Databases allow data to be stored in XML format. The data stored in an XML database can be queried, exported and serialized into any format needed.

Big Data Scientist

Someone who is able to develop the distributed algorithms to make sense out of big data

Classification analysis

A systematic process for obtaining important and relevant information about data, also meta data called; data about data.

Cloud computing

A distributed computing system over a network used for storing data off-premises. This can include ETL, data storage, application development, and data analytics. Examples: Amazon Cloud and Google Cloud.

Cloud computing is one of the must-known big data terms. It is a new paradigm computing system which offers visualization of computing resources to run over the standard remote server for storing data and provides IaaS, PaaS, and SaaS. Cloud Computing provides IT resources such as Infrastructure, software, platform, database, storage and so on as services. Flexible scaling, rapid elasticity, resource pooling, on-demand self-service are some of its services.

Clustering analysis

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters).

Database-as-a-Service

A database hosted in the cloud on a pay per use basis, for example Amazon Web Services

Database Management System (DBMS)

Database Management System is software that collects data and provides access to it in an organized layout. It creates and manages the database. DBMS provides programmers and users a well-organized process to create, update, retrieve, and manage data.

Distributed File System

Systems that offer simplified, highly available access to storing, analysing and processing data; examples are:

- Hadoop Distributed File System (HDFS)
- Amazon S3 (a distributed object storage system)

Document Store Databases

A document-oriented database that is especially designed to store, manage and retrieve documents, also known as semi structured data.

NoSQL

NoSQL sometimes referred to as ‘Not only SQL' as it is a database that doesn't adhere to traditional relational database structures. It is more consistent and can achieve higher availability and horizontal scaling. NoSQL is an approach to database design that can accommodate a wide variety of data models, including key-value, document, columnar and graph formats. NoSQL, which stands for "not only SQL," is an alternative to traditional relational databases in which data is placed in tables and data schemais carefully designed before the database is built. NoSQL databases are especially useful for working with large sets of distributed data.

Scala

A software programming language that blends object-oriented methods with functional programming capabilities. This allows it to support a more concise programming style which

reduces the amount of code that developers need to write. Another benefit is that Scala features, which operate well in smaller programs, also scale up effectively when introduced into more complex environments.

Columnar Database

A database that stores data column by column instead of the row is known as the column-oriented database.

Data Analyst

The data analyst is responsible for collecting, processing, and performing statistical analysis of data. A data analyst discovers the ways how this data can be used to help the organization in making better business decisions. It is one of the big data terms that define a big data career. Data analyst works with end business users to define the types of the analytical report required in business.

Data Scientist

Data Scientist is also a big data term that defines a big data career. A data scientist is a practitioner of data science. He is proficient in mathematics, statistics, computer science, and/or data visualization who establish data models and algorithms for complex problems to solve them.

Data Model and Data Modelling

Data Model is a starting phase of a database designing and usually consists of attributes, entity types, integrity rules, relationships and definitions of objects.

Data modeling is the process of creating a data model for an information system by using certain formal techniques. Data modeling is used to define and analyze the requirement of data for supporting business processes.

Hive

Hive is an open source Hadoop-based data warehouse software project for providing data summarization, analysis, and query. Users can write queries in the SQL-like language known as

HiveQL. Hadoop is a framework which handles large datasets in the distributed computing environment.

Load Balancing

Load balancing is a tool which distributes the amount of workload between two or more computers over a computer network so that work gets completed in small time as all users desire to be served faster. It is the main reason for computer server clustering and it can be applied with software or hardware or with the combination of both.

Load balancing refers to distributing workload across multiple computers or servers in order to achieve optimal results and utilization of the system

Log File

A log file is the special type of file that allows users keeping the record of events occurred or the operating system or conversation between the users or any running software.

Log file is a file automatically created by a computer program to record events that occur while operational.

Parallel Processing

It is the capability of a system to perform the execution of multiple tasks simultaneously (at the same time)

Server (or node)

The server is a virtual or physical computer that receives requests related to the software application and thus sends these requests over a network. It is the common big data term used almost in all the big data technologies.

Abstraction layer

A translation layer that transforms high-level requests into low-level functions and actions. Data abstraction sees the essential details needed to perform a function removed, leaving behind the complex, unnecessary data in the system. The complex, unneeded data is hidden from the client, and a simplified representation is presented. A typical example of an abstraction layer is

an API (application programming interface) between an application and an operating system.

Cloud

Cloud technology, or The Cloud as it is often referred to, is a network of servers that users access via the internet and the applications and software that run on those servers. Cloud computing has removed the need for companies to manage physical data servers or run software applications on their own devices - meaning that users can now access files from almost any location or device.

The cloud is made possible through virtualisation - a technology that mimics a physical server but in virtual, digital form, A.K.A virtual machine.

Data ingestion

Data ingestion is the process of moving data from various sources into a central repository such as a data warehouse where it can be stored, accessed, analysed, and used by an organisation.

Data warehouse

A centralised repository of information that enterprises can use to support business intelligence (BI) activities such as analytics. Data warehouses typically integrate historical data from various sources.

Open-source

Open-source refers to the availability of certain types of code to be used, redistributed and even modified for free by other developers. This decentralised software development model encourages collaboration and peer production.

Relational database

A relational database exists to house and identify data items that have pre-defined relationships with one another. Relational databases can be used to gain insights into data in relation to other data via sets of tables with columns and rows. In a relational database, each row in the table has a unique ID referred to as a key.

What do you mean by relational database? a relational database is a collection of information

(stored as rows) that organizes data in predefined relationships where data is stored in one or more tables (or "relations") of columns and rows, making it easy to see and understand how different data structures relate to each other.

There are 3 different types of relations in the database:

- one-to-one
- one-to-many
- many-to-many

How does Hadoop perform input splits?

The Hadoop's `InputFormat<K, V>` is responsible to provide the splits. The `InputFormat<K,V>` describes the input-specification for a Map-Reduce job. The interface `InputFormat` 's full name is `org.apache.hadoop.mapred.InputFormat<K,V>` .

According to Hadoop: the Map-Reduce framework relies on the `InputFormat` of the job to:

1. Validate the input-specification of the job.
2. Split-up the input file(s) into logical InputSplits, each of which is then assigned to an individual Mapper.
3. Provide the `RecordReader` implementation to be used to glean input records from the logical InputSplit for processing by the Mapper.

In general, if you have `N` nodes, the HDFS will distribute the input file(s) over all these `N` nodes. If you start a job, there will be `N` mappers by default. The mapper on a machine will process the part of the data that is stored on this node.

MapReduce/Hadoop data processing is driven by this concept of input splits. The number of input splits that are calculated for a specific application determines the number of mapper tasks.

The number of maps is usually driven by the number of DFS blocks in the input files. Each of these mapper tasks is assigned, where possible, to a worker node where the input split is stored. The Resource Manager does its best to ensure that input splits are processed locally (for optimization purposes).

How does Sort & Shuffle work in MapReduce/Hadoop

Shuffle phase in Hadoop transfers the map output (in the form of (key, value) pairs) from Mapper

to a Reducer in MapReduce. Sort phase in MapReduce covers the merging and sorting of mappers outputs. Data from the mapper are grouped by the key, split among reducers and sorted by the key. Every reducer obtains all values associated with the same key.

For example, if there were 3 input chunks/splits, then mappers create (key, value) pairs per split (i call them partitions), consider all of the output from all of the mappers:

```
Partition-1  Partition-2  Partition-3
(A, 1)       (A, 5)       (A, 9)
(A, 3)       (B, 6)       (C, 20)
(B, 4)       (C, 10)      (C, 30)
(B, 7)
```

Then the output of Sort & Shuffle phase will be (note that the values of keys are not sorted):

```
(A, [1, 3, 9, 5])
(B, [4, 7, 6])
(C, [10, 20, 30])
```

Output of Sort & Shuffle phase will be input to reducers.

NoSQL Database

NoSQL databases (aka "not only SQL") are non-tabular databases and store data differently than relational tables. NoSQL databases come in a variety of types. Rdis, HBase, CouchDB and ongoDB, ... are examples of NoSQL databases.

References

1. Data Algorithms with Spark by Mahmoud Parsian
2. Data Algorithms by Mahmoud Parsian
3. Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms by Jimmy Lin
4. Google's MapReduce Programming Model — Revisited by Ralf Lammel
5. MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat

6. Data-Intensive Text Processing with MapReduce by Jimmy Lin and Chris Dyer
7. MapReduce Design Patterns by Donald Miner, Adam Shook
8. Hadoop: The Definitive Guide, 4th Edition by Tom White
9. Learning Spark, 2nd Edition by Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee
10. Mining of Massive Datasets by Jure Leskovec, Anand Rajaraman, Jeff Ullman
11. Chapter 2, MapReduce and the New Software Stack by Jeff Ullman
12. A Very Brief Introduction to MapReduce by Diana MacLean
13. Apache Hadoop MapReduce Tutorial, 2022-07-29
14. Big Data Glossary by Pete Warden, 2011, O'Reilly
15. What is Lineage In Spark?
16. RDD lineage in Spark: ToDebugString Method
17. Lazy Evaluation in Apache Spark
18. Advanced Analytics with PySpark by Akash Tandon, Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills
19. 8 Steps for a Developer to Learn Apache Spark with Delta Lake by Databricks
20. Apache Spark Key Terms, Explained
21. How Data Partitioning in Spark helps achieve more parallelism?