DMIF, University of Udine

Hadoop and MapReduce

Andrea Brunello
andrea.brunello@uniud.it

May, 2020

Introduction

What is Hadoop ?

Hadoop is an open-source platform designed to support distributed computation in a reliable and scalable way.

Hadoop was developed by Doug Cutting and Mike Cafarella in 2005 to address a scalability problem of an open-source crawler (Nutch).

The first release in 2008 was an independent project of Apache. Nowadays, it is a collection of projects belonging to the same infrastructure for distributed computing.

Before Hadoop

Data processing on massive amounts of data were performed by means of *High Performance Computing* (HPC) and *Grid Computing*, trough APIs like *Message Passing Interface*.

HPC subdivides the work across several nodes in a cluster, each using a shared *file system* on a network.

If the work is *processor-intense*, the system performs fine. If there is the need to access a huge amount of data, delays to access the shared storage are likely to occur.

Advanced networking communications, such as Infiniband, are needed, because the size of the processed data requires high throughput and low latency.

Advantages in using Hadoop

Hadoop is easier to use than HPC solutions as the libraries are from a higher level (and since many people are now using it).

The partitioning of the data across the computing nodes is crucial in order to avoid network transfer of data (data locality).

Hadoop is reliable: designed to use (cheap) commodity hardware, it has the capability to manage hardware failures.

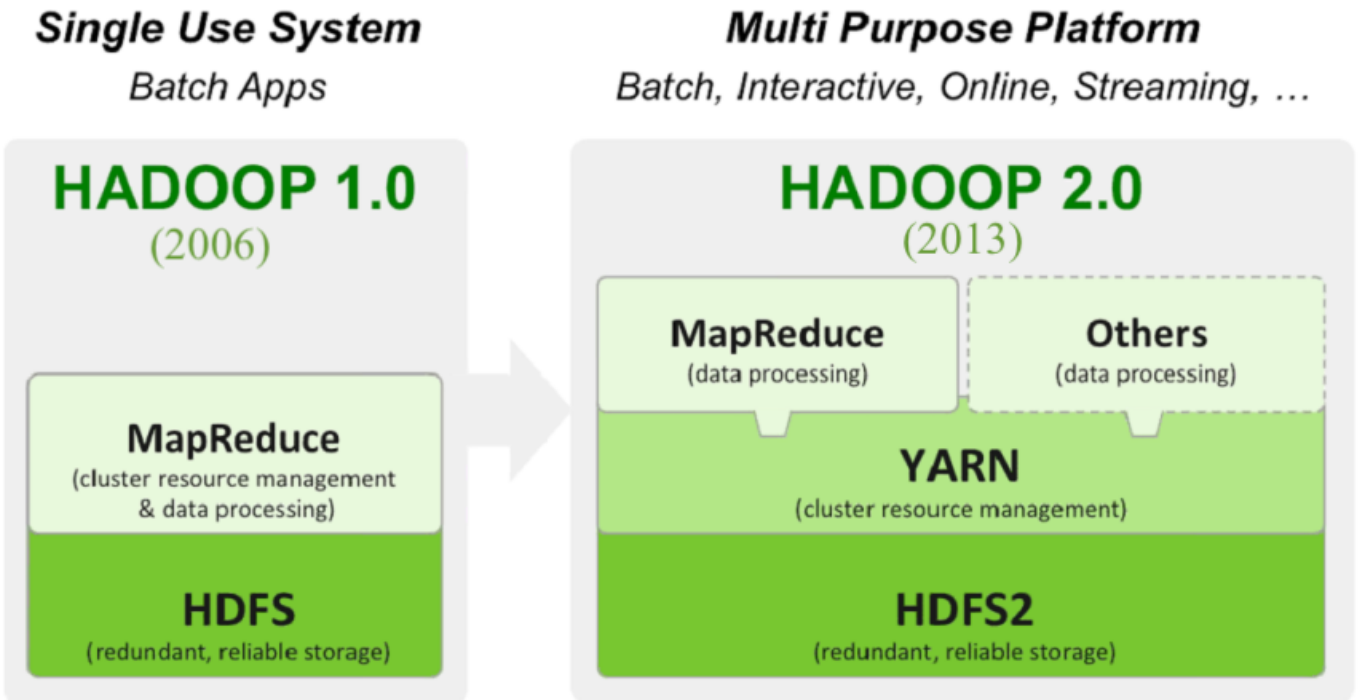
Sacalability is easy and cheap, you just need to add nodes to the cluster, there is no need of expensive and specifically designed hardware.

Hadoop Overview

Hadoop core components

- *Hadoop common:* software layer that acts as a support for the other modules, providing libraries and utilities.
- *HDFS:* distributed file system that stores data on commodity machines. It provides an effective way to access the data, guaranteeing redundancy to deal with failures. Any file format is supported, structured or not.
- *YARN:* (Yet Another Resource Negotiator) platform responsible for managing computing resources in clusters and using them to schedule users' applications.
- *MapReduce:* a parallel processing system for managing huge amounts of data, following the *divide et impera* strategy.

Hadoop 1.0 and 2.0

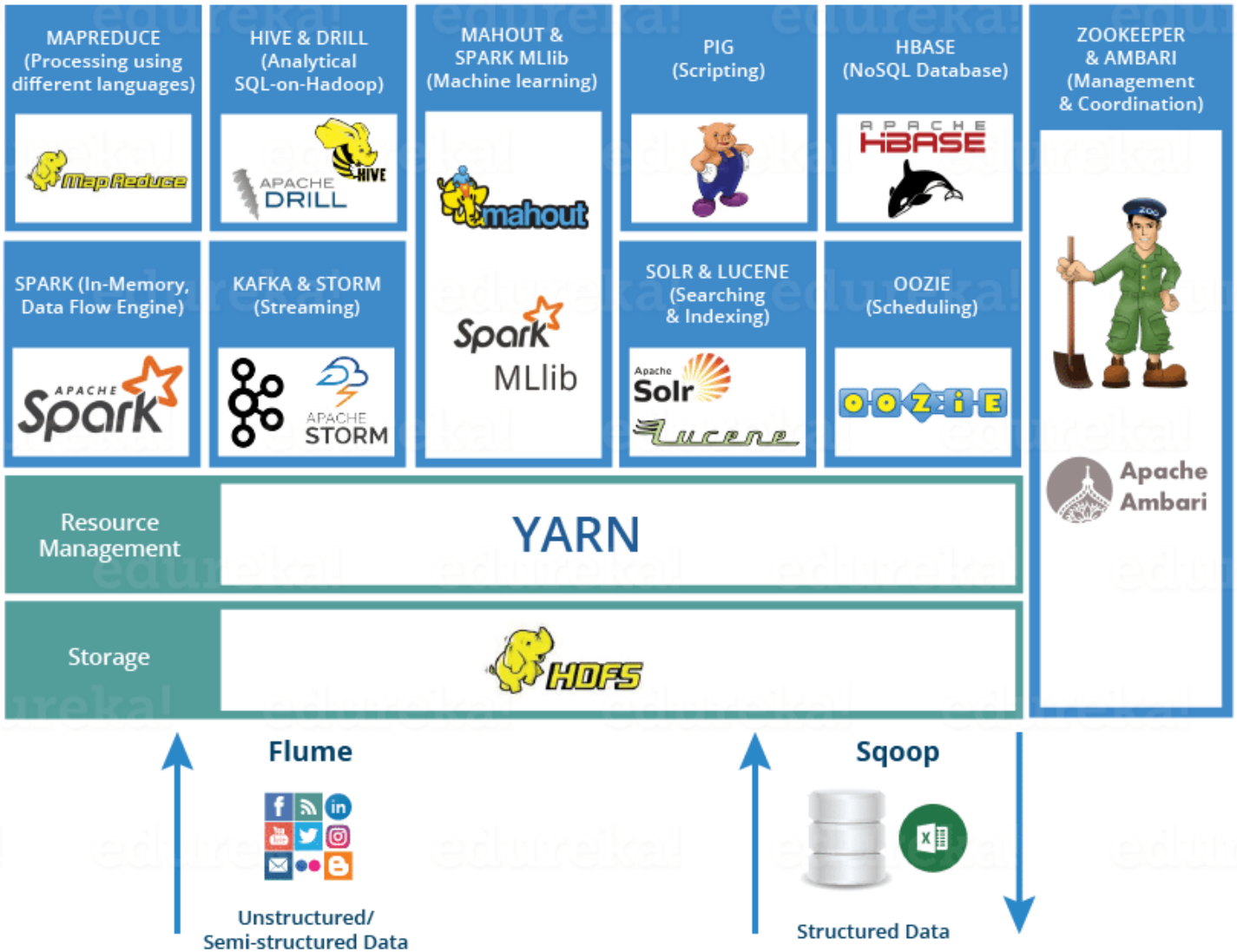


Hadoop 3.0 (2017)

Hadoop 3.0 brings thousands of bug fixes, features and enhancements over Hadoop 2.0

- Minimum Runtime Version for Hadoop 3.0 is JDK 8
- Fault tolerance via Erasure Coding in HDFS
- Support for Multiple NameNodes to maximize Fault Tolerance
- MapReduce Task Level Native Optimization (30% performance improvement for the jobs)
- Hadoop Shell Script Rewrite to fix bugs

Hadoop Extended Ecosystem



Apache Sqoop

- The name stands for Sql to Hadoop
- It is a straightforward command line tool
- Designed to transfer efficiently bulk data between Apache Hadoop and relational databases
- It supports the incremental reading of a relational table and the writing to HDFS, Hive or HBase

Apache Flume, Storm, and Kafka

They all allow to handle streaming data.

Flume is specifically designed to move unstructured or semi-structured data to Hadoop (HDFS, Hive, HBase), particularly log data.

Kafka is a more general-purpose tool. It adopts a distributed messanging system where publishers write data to topics and subscribers read from topics, providing a unified, low-latency, high-throughput platform for handling real-time data feeds. It should be used when the data destination is not (just) Hadoop.

Storm is a distributed real-time computation system not just for streaming, but it also includes other features such as real-time analytics, continuous computation, ...

HBase, Hive, and Drill

HBase is a non-relational distributed database modeled after Google's Bigtable and written in Java.

Hive is a data warehousing solution. It provides HiveQL, a language similar to SQL, that allows to run queries with MapReduce support in a transparent way.

Drill is a schema-free SQL query engine. It allows one to perform SQL queries against several NoSQL databases, and local files.

Mahout and Spark MLlib

They both provide scalable and distributed implementations of machine learning algorithms.

Mahout includes algorithms that support many tasks, such as classification, clustering, dimensionality reduction, and topic extraction. Originally based on MapReduce, today it is primarily focused on Spark.

Spark MLlib is a scalable machine learning library based on Spark. It includes all the most popular machine learning algorithms, such as random forests, gradient boosting trees, K-means, LDA, …

Oozie, Solr and Lucene

Oozie is a server-based workflow scheduling system to manage Hadoop jobs. It combines multiple jobs sequentially into one logical unit of work.

Solr and *Lucene* provide a search engine software library. Major features include full-text search, real-time indexing, dynamic clustering, database integration, NoSQL features and rich document (e.g., Word, PDF) handling.

Spark

Spark is an open-source distributed general-purpose cluster-computing framework, like MapReduce.

Differently from MapReduce, which has to read from and write to a disk while performing processing tasks, Spark can do it in-memory.

As a result, developers claim that Spark is capable of running programs up to 100x faster than MapReduce, making it suitable for real-time computation.

Nevertheless, Spark requires a lot of memory to load the processes. On the contrary, leveraging the disk, MapReduce is able to work with far larger datasets than Spark.

Ambari and Zookeeper

Ambari and Zookeeper provide support to the Hadoop administrators.

Ambari allows the provisioning, management and monitoring of Hadoop clusters.

Zookeeper is essentially a service for distributed systems offering a hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems.

HDFS

HDFS

HDFS is a distributed file system designed to work with commodity hardware.

It has been developed in order to manage a large number of files, possibly big (Gigabytes or Terabytes), and to be reliable and scalable.

It can support clusters with thousands of nodes, that may be affected by disconnections and failures.

The capability to detect failures and to perform proper automatic recovery actions is one of its main characteristics.

HDFS architecture

HDFS keeps files in an hierarchical folder structure, handled by means of these main kinds of nodes:

- *NameNode:* application on the main server that manages the file system and controls the file access (open, close, rename) and also the data distribution, replication and reallocation across the nodes.
- *DataNode:* application that runs on the other nodes of the cluster and that manages the physical storage, performing read and write operations in addition to the creation, deletion or replica of data blocks.
- *Secondary NameNode:* this is not a failover node for the NameNode, but a service that helps improve the efficiency of the NameNode. Also called CheckPointNode.

HDFS NameNode failure

For HDFS versions before the 2.X.X the Namenode was an application running on a single node.

This was a huge problem because the failure of the NameNode would lead to the *failure of the entire cluster*.

From 2.X.X onwards, two servers are configured as a NameNode. One is active while the other one remains in standby waiting to get into action in case of a failure.

Hadoop archives

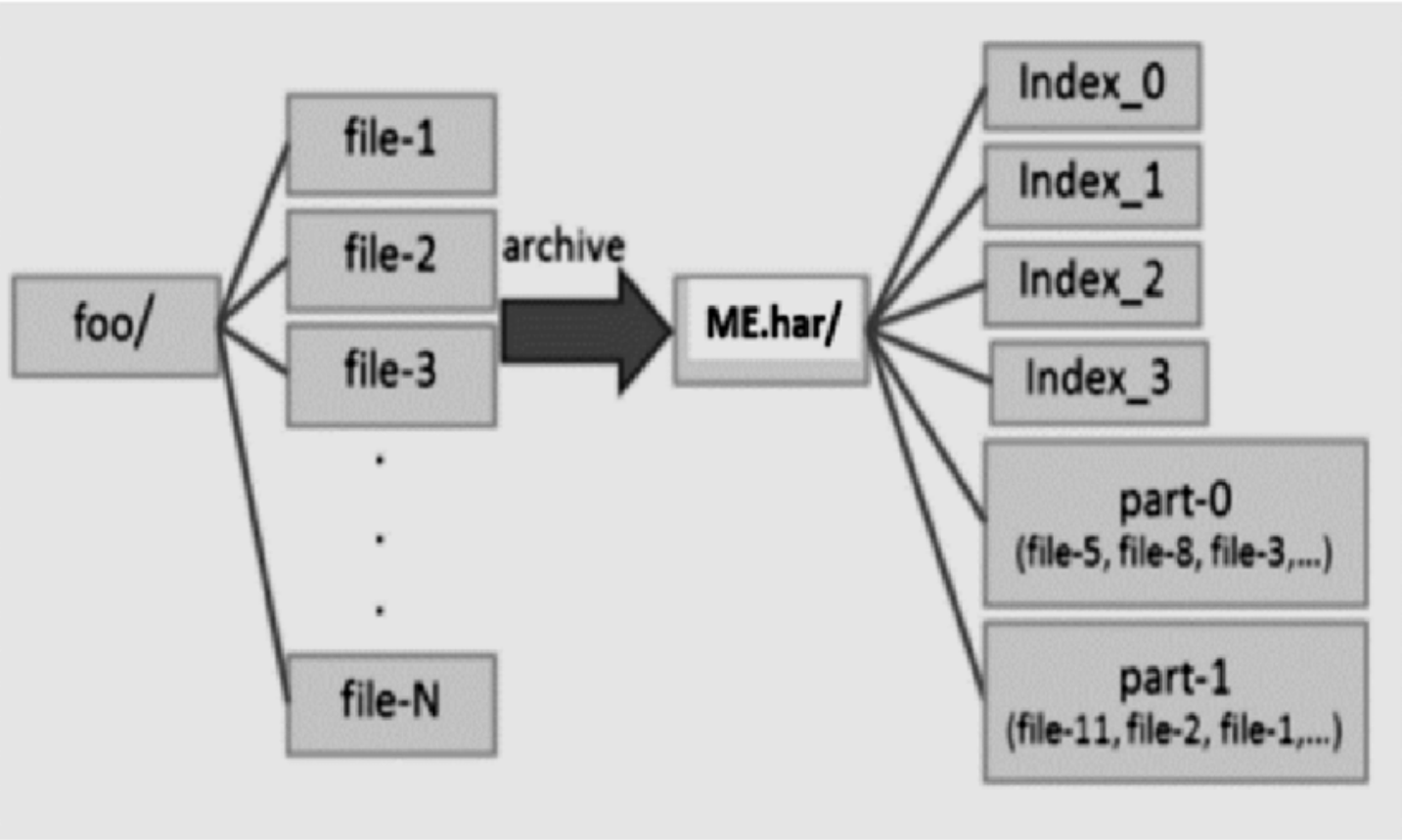
HDFS is very effective in managing big files but it is rather inefficient in handling those much smaller than the block size (64 or 128 Megabytes).

This happens because a large amount of small files occupy a lot of memory inside the namespace managed by the NameNode, which might saturate.

Also, a lot of files with a much smaller dimension than the block size leads to a partial usage of the disk space, as blocks will be mostly empty.

To overcome this issue, *archives* compact small files together allowing parallel access without the need of decompressing them.

HDFS archive example



HDFS usage - Basic commands

Access to the file system and its management can be performed using the *command shell* or the *Web API* by means of the commands:

- `cat`: copy file to the standard output
- `get`: copy from HDFS to the local system
- `put`: copy from local system to the HDFS
- `ls`: returns information about the current directory
- `mkdir`: creates a directory
- `rm`: removes a file or a directory
- `rmr`: recurrent version of rm
- `mv`: moves or renames a file
- `help`: returns the manual

HDFS usage - Security and permissions

HDFS security and permissions are based on the classical POSIX (Portable Operating System Interface) model.

Files and directories are associated to an *owner* and a *group*, and permissions are separately assigned to them.

Permission may be related to read (*r*), write (*w*) and execute (*x*) operations.

Permissions can be managed by means of the commands *chmod* (change the r/w/x permissions over a file or directory), *chgrop* (change the associated group), and *chwown* (change the owner).

There is also the concept of *superuser*, that corresponds to the NameNode process identity.

HDFS Federation

Early HDFS versions allowed a single NameNode across the entire cluster, leading to a scalability problem.

In fact, while several datanodes can be added to the cluster in order to increase storage and computing power, unfortunately the NameNode cannot scale.

From version 0.23 onward, the introduction of Federation allowed the presence of several NameNodes, independent from each other.

The set of blocks belonging to a specific NameNode is named *Block Pool*. A Namespace with its associated Blocks is called the *Namespace Volume*.

Federation also allows the separation of volumes considering different users and applications.

HDFS file formats

HDFS can store files in any format (text, images like PNG, JPG, TIFF, and any binary files), but there are also some proprietary formats. Several kinds of compression are also available.

The choice of a particular format can be driven by:

- intelligibility and interoperability
- space occupation
- capability to split the data in order to parallelize the MapReduce tasks
- metadata available to the format

HDFS file formats - Text

Traditional text files such as CSV (Comma Separated Values) and TSV (Tab Separated Values) are common in Hadoop but not particularly efficient.

They simplify the transfer to and from external data sources and are human readable.

They are naturally splittable, but do not support block compression: they are compressed only at the file level, which makes the reading phase quite costly (you always need to decompress the entire file first).

Caveat: file header should be removed, since in Hadoop each row has to be a record onto which it is possible to perform operations (loss of metadata information).

HDFS file formats - Sequence

Sequence file format is Hadoop native and allows one to store key/value pairs.

As we shall see, it is typically used for transferring data between MapReduce jobs.

It is splittable and supports block compression.

To store complex or unstructured data (such as images), it is necessary to encode the object identifier as the key, and serialize the object as the value.

HDFS file formats - Avro

Avro is both a file format and a serialization/deserialization system that is becoming quite common.

Avro stores metadata together with the data and allows to manage complex schemas through the JSON format.

Avro is splittable and it also supports block compression.

```
{
   "namespace": "customer.avro",
   "type": "record",
   "name": "customer",
   "fields": [
      {"name": "name",    "type": "string"},
      {"name": "age",     "type": "int"},
      {"name": "address", "type": "string"},
      {"name": "phone",   "type": "string"},
      {"name": "email",   "type": "string"},
   ]
}
```

HDFS file formats - RC File

Record Columnar File is a format for the serialization of tabular data.

A table is subdivided horizontally in row groups and then serialized by column.

Data is then compressed by column to save disk space.

This format does not support schema evolution, i.e., changes to the schema (such as the addition of a columm): they require the rewriting of the entire file.

HDFS NFS Gateway

NFS (Network File System) is a protocol for distributed file systems that allows one to access remote files as if they were local files.

HDFS NFS Gateway allows the access to HDFS by means of the NFS protocol, allowing:

- navigation in the file system
- file upload
- file download
- streaming data to HDFS (with some limitations)

Thus, HDFS NFS Gateway greatly simplifies the data exchange between HDFS and external file systems.

MapReduce

What is MapReduce ?

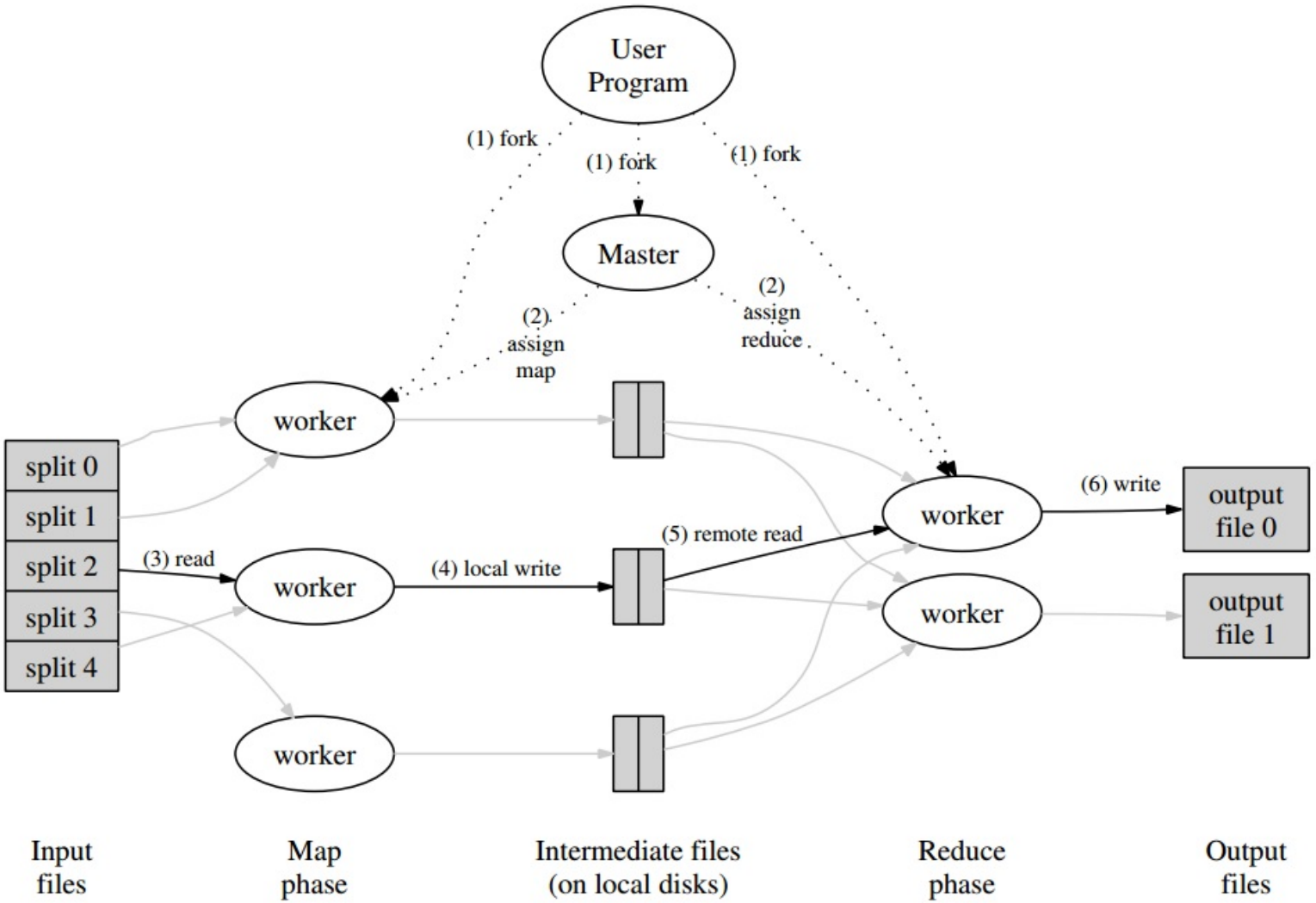
MapReduce is a framework for developing applications capable of processing huge amounts of data in parallel.

It is based on *functional programming* rather than *multithreading*. This allows to avoid the complexity of managing the access to shared data, which is instead exchanged as arguments and return values of the functions.

MapReduce works following the principle of *divide et impera*. A problem is subdivided into several parts each one processed autonomously. Once all parts are processed, the result is obtained through re-composition.

The user just needs to specify the working logic of the application. The framework then handles task monitoring, failure recovery, and node selection (according to data locality).

General MapReduce framework

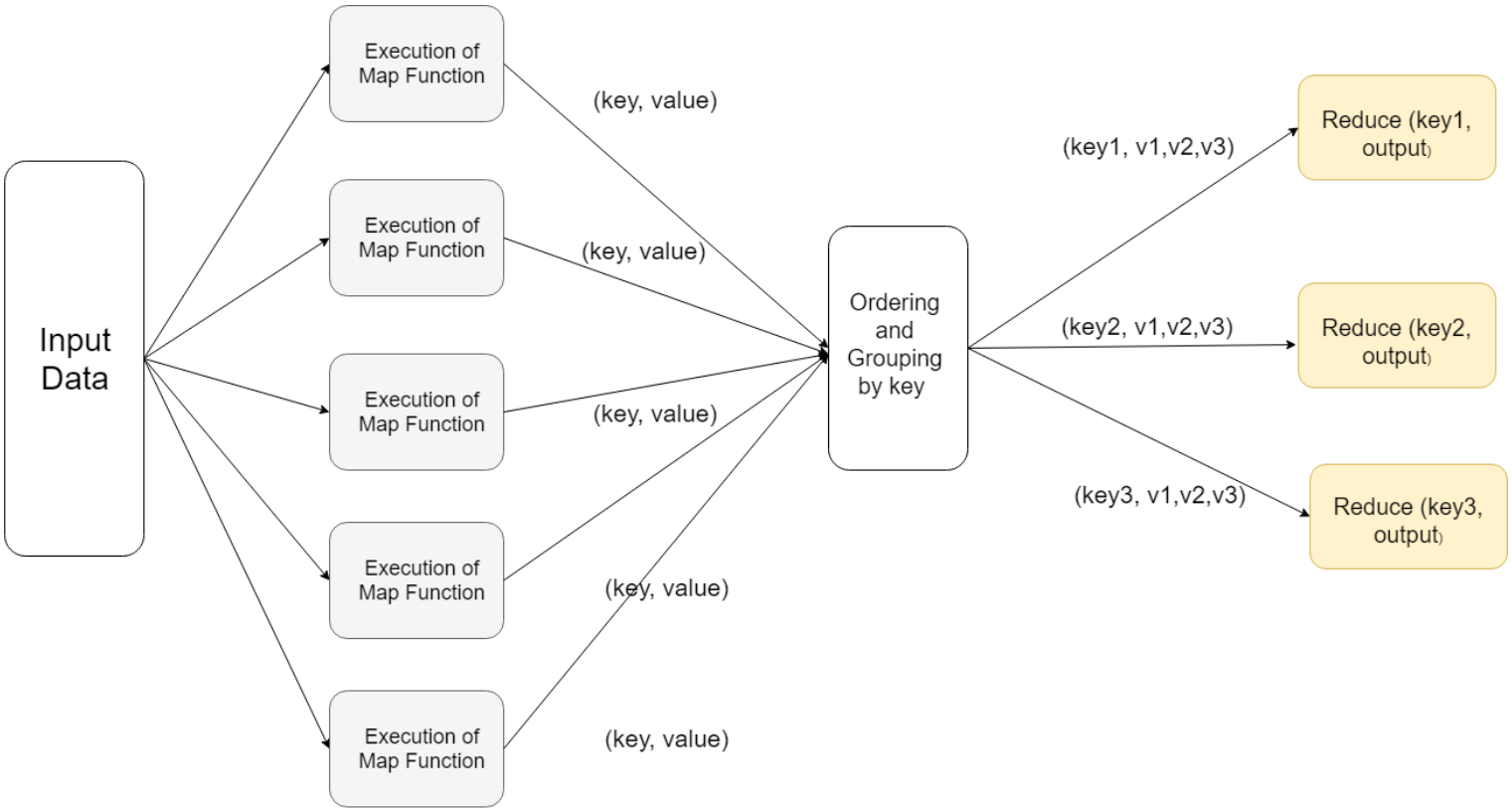


MapReduce job

A MapReduce job is composed of four elements:

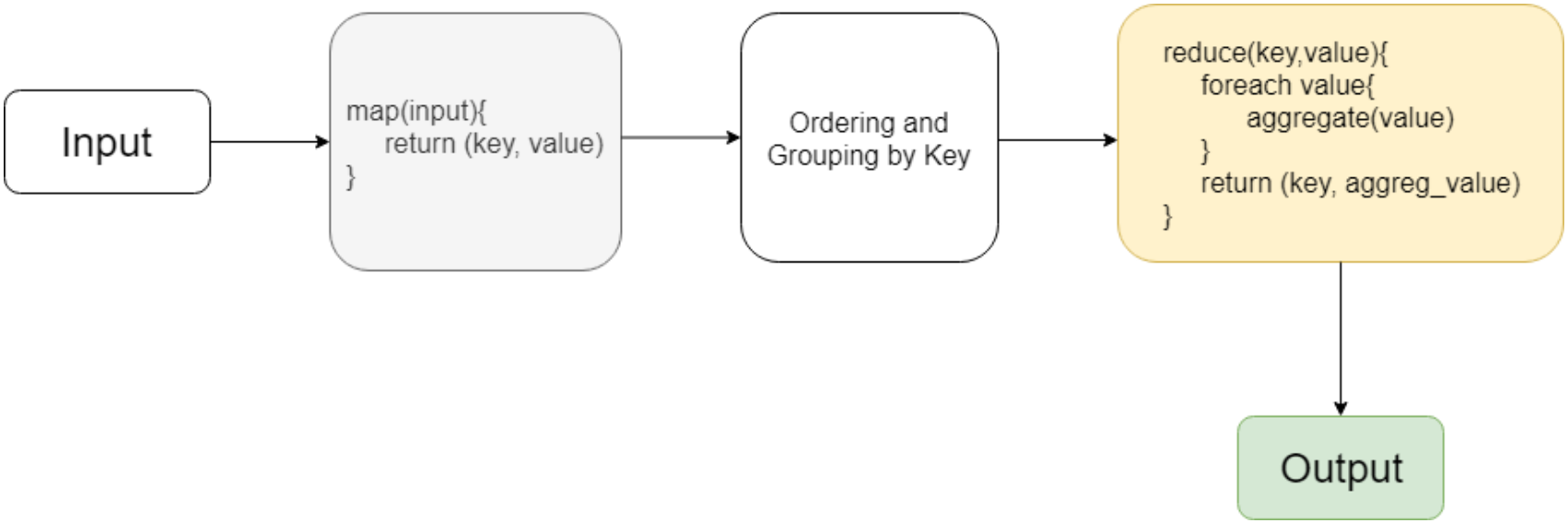
- the *input data* from HDFS
- a *map* function that transforms data in key-value couples
- a *reduce* function that, for each key, elaborates the associated values and creates one or more key-value couples as output
- the *output* result, written on a HDFS file

MapReduce job



The reduce step is preceded by a data gathering phase, in which all key-value couples produced by the map function are collected and sorted, so to bring all values sharing the same key together.

MapReduce job detail



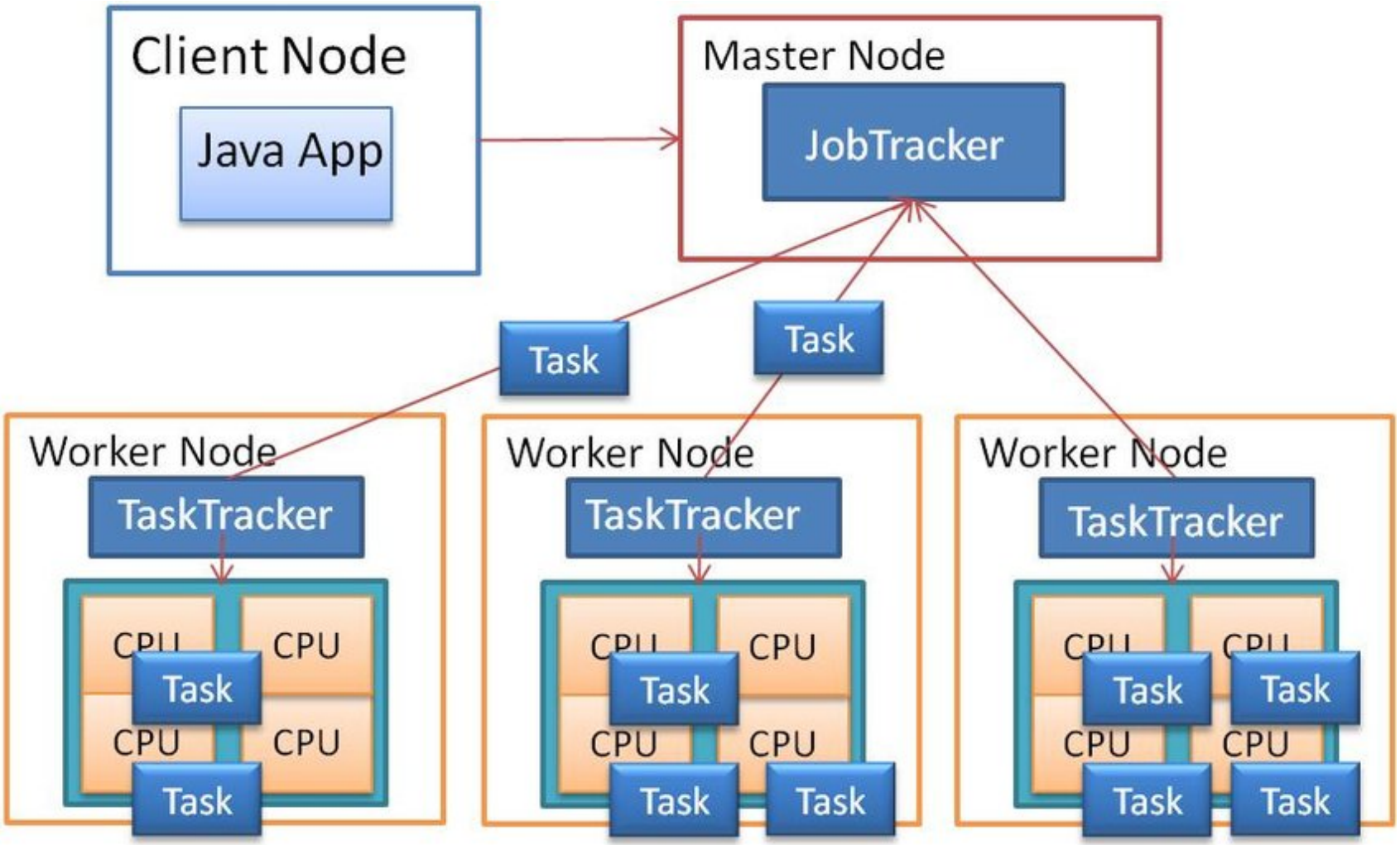
Of course, several MapReduce phases may be combined together, in order to solve complex problems.

MapReduce architecture

At the architectural level MapReduce is composed of two main components:

- *JobTracker* takes care of the resources (memory, CPU) and the life cycle of a MapReduce job. It distributes the workload giving priority to the nodes that are close to the data (e.g., node with the data, or at least in the same rack). It also takes care of scheduling and error handling policies.
- *TaskTracker* are the components running on single nodes and executing the tasks under the control of the JobTracker.

MapReduce job scheduling



MapReduce usage scenarios

- Creation of word lists from text documents, indexing and search. Examples of such category are: counting, extraction of unique values (e.g., in logs from a Web server) and data filtering applications.
- Complex data structures analysis like graphs (e.g., social network analysis), or data mining.
- Execution of distributed tasks (like complex mathematical calculations and numerical analysis).
- Correlation, union, intersection, aggregation and join operations (e.g, market analyses, descriptive analytics).

Algorithms for mapReduce - 1

Text search Searching for text rows containing a specific word (or set of words). In this case only the mapper function is needed to obtain the desired text rows, no need to write an aggregation function for the results as an additional step. This is also called a *map-only* job.

Sorting Ordering problems fit very well to the framework as the key-value couples are sorted before the reduce phase. This particular job is easy to implement as involves the usage of two simple classes: Identity Mapper and IdentityReducer that return in output what they get in input.

Algorithms for MapReduce - 2

Statistical calculations Computing minimum, maximum and counts is easy because they are commutative and associative operations. The mapper class executes the calculations locally filling a buffer and, when it is full, it gives in output the key and the computed content.

The reducer computes the global value simply applying the same function over the partial results.

Other calculations, like average, are slightly different because there is the need to obtain the components to be computed by the reducer. Considering the average, the mapper will count and sum the elements and the reducer will compute the global sum divided the global number of elements.

CountWord – Classical serial code

To count words from a single source you can simply build an iterative process like this:

- Read a word
- Lookup in word table (that keeps track of word counts)
  - If the word is not present, then add it with count 0
- Add 1 to the count for the word

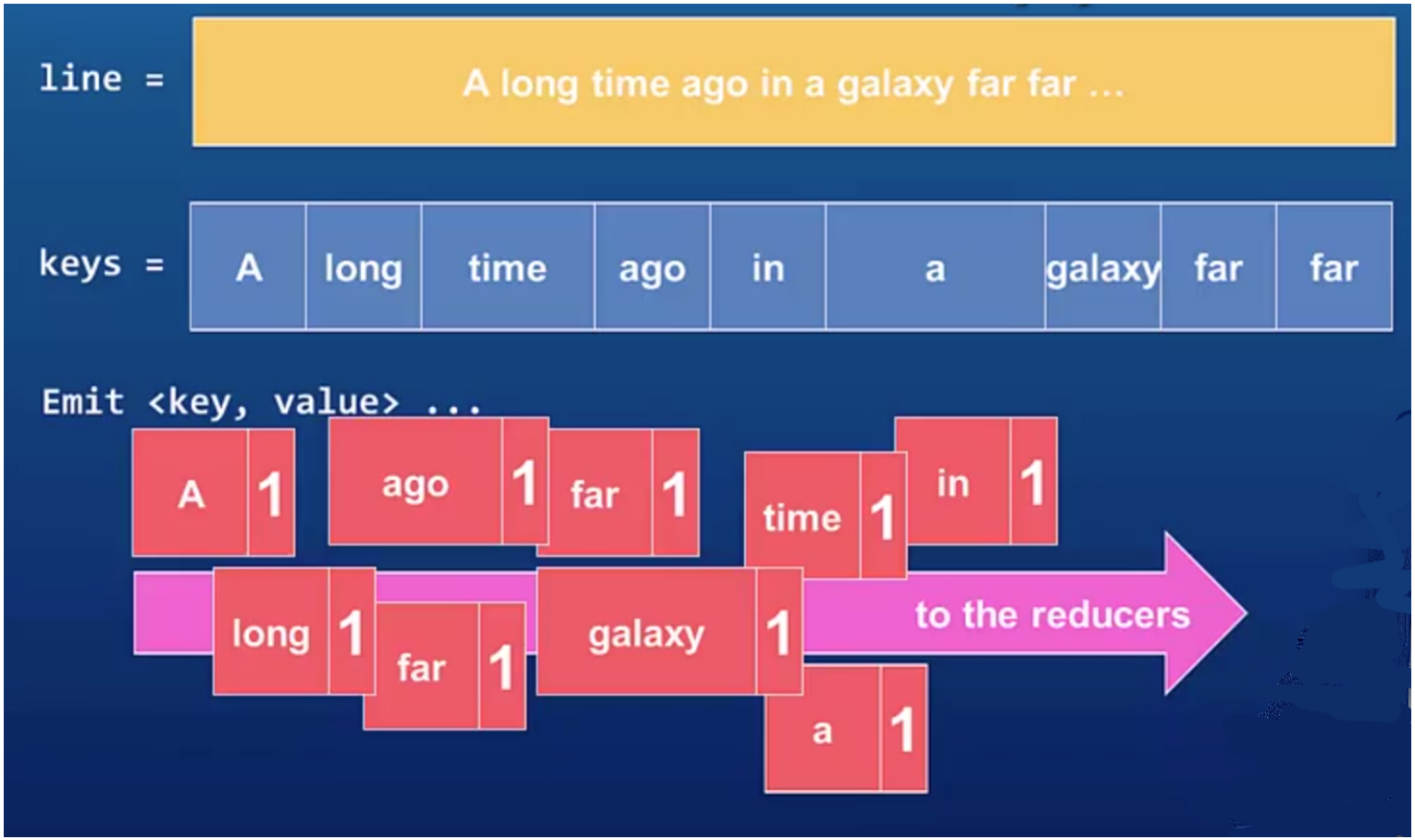
But how can you process thousand or millions of documents without considering one at a time?

CountWord – Mapper strategy

As we shall see, MapReduce is going to keep things very simple:

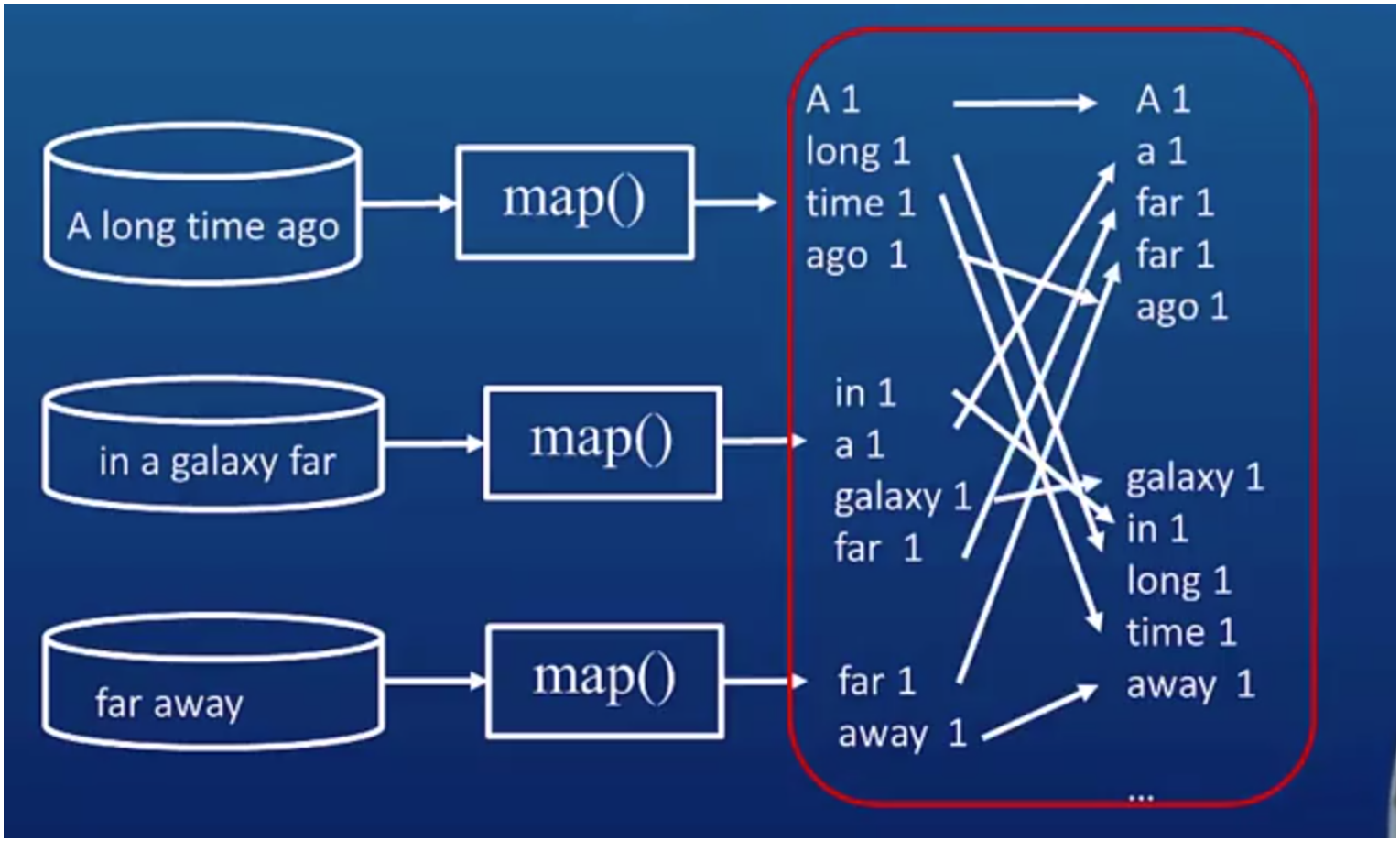
- Let <word, 1> be the <key, value>
- The mapper will perform the loop
  - Get the word
  - Emit <word,1>
- Then Hadoop is going to make the hard work

What one mapper does



Notice that the same word may appear multiple times, each with the number 1.

Mapping and sorting



Note: not necessarily a global sort at this point, just put together the same words.

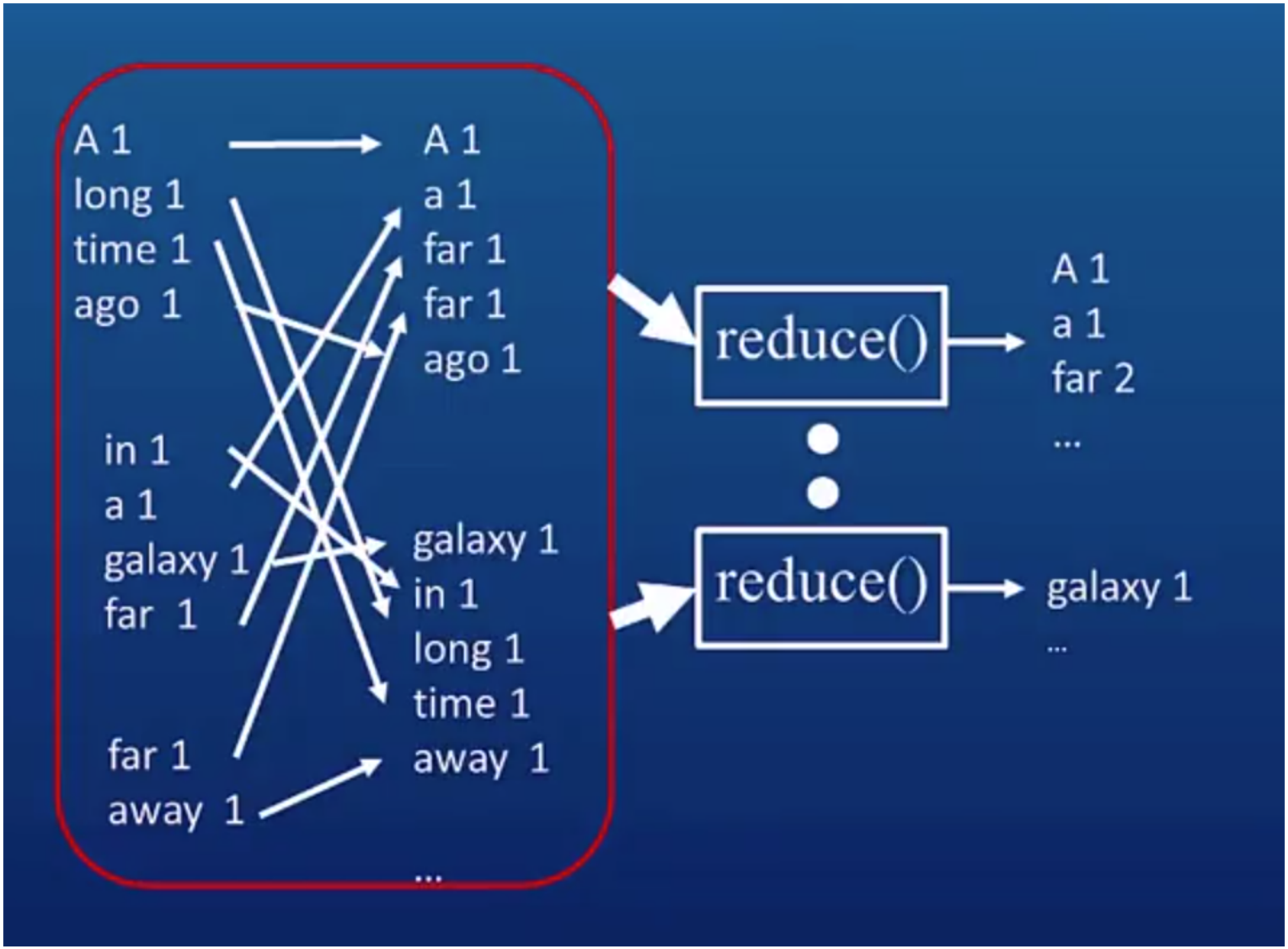
CountWord – Reducer strategy

The reducer is guaranteed to get all the same keys together so, as it reads key-value pairs, it just needs to keep a running total.

The loop for the reducer is like the following

- Get the next <word, value>
- If word is the same as the previously read one
  - add 1 to count
- Else
  - emit <word, count> and set count to 0

What the reducers do



How to write a MapReduce job

Writing a MapReduce Job can be a complex task.

It involves the creation of three Java classes:

- *mapper*
- *reducer*
- *driver*

Each of the above classes requires several libraries that refer to different Hadoop components and Java I/O functionalities.

Writing a MapReduce job - Libraries

```
1.      package org.myorg;
2.
3.      import java.io.IOException;
4.      import java.util.*;
5.
6.      import org.apache.hadoop.fs.Path;
7.      import org.apache.hadoop.conf.*;
8.      import org.apache.hadoop.io.*;
9.      import org.apache.hadoop.mapred.*;
10.     import org.apache.hadoop.util.*;
```

Writing a MapReduce job - Mapper

```
12. public class WordCount {
13.
14.  public static class Map extends MapReduceBase implements
      Mapper<LongWritable, Text, Text, IntWritable> {
15.     private final static IntWritable one = new IntWritable(1);
16.     private Text word = new Text();
17.
18.     public void map(LongWritable key, Text value,
         OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
19.       String line = value.toString();
20.       StringTokenizer tokenizer = new StringTokenizer(line);
21.       while (tokenizer.hasMoreTokens()) {
22.         word.set(tokenizer.nextToken());
23.         output.collect(word, one);
24.       }
25.     }
26.   }
```

The map method receives a key-value pair in input. In our case, these may be the offset of a text row, and its content. The OutputCollector is a collection of key-value pairs that will contain the result of the mapper (<word, 1> couples).

Key and value data types are objects that allow Hadoop to perform read and write file operations.

Writing a MapReduce job - Reducer

```
28.   public static class Reduce extends MapReduceBase implements
       Reducer<Text, IntWritable, Text, IntWritable> {
29.     public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
30.       int sum = 0;
31.       while (values.hasNext()) {
32.         sum += values.next().get();
33.       }
34.       output.collect(key, new IntWritable(sum));
35.     }
36.   }
```

The function receives key-value couples that are already sorted and partitioned (each reducer works on a key).

The object Reporter contains information regarding the task, for instance status and performance statistics.

Writing a MapReduce job - Driver

```
38.         public static void main(String[] args) throws Exception {
39.           JobConf conf = new JobConf(WordCount.class);
40.           conf.setJobName("wordcount");
41.
42.           conf.setOutputKeyClass(Text.class);
43.           conf.setOutputValueClass(IntWritable.class);
44.
45.           conf.setMapperClass(Map.class);
46.           conf.setCombinerClass(Reduce.class);
47.           conf.setReducerClass(Reduce.class);
48.
49.           conf.setInputFormat(TextInputFormat.class);
50.           conf.setOutputFormat(TextOutputFormat.class);
51.
52.           FileInputFormat.setInputPaths(conf, new Path(args[0]));
53.           FileOutputFormat.setOutputPath(conf, new Path(args[1]));
54.
55.           JobClient.runJob(conf);
57.         }
58.     }
```

Finally, the Driver initializes the job, sets the input parameters and determines how to save the output.

MapReduce usage

MapReduce represents the lowest (and most complex) level of Hadoop programming.

Any detail of data computation and storage can be managed trough the Java libraries.

It is particularly useful for dealing with unstructured data, since it is more flexible than higher level solutions.

MapReduce potential limitations

- The problem must fit <key, value> paradigm
- Accessing temporary, intermediate MapReduce results is not easy
- Requires programming/debugging skills
- Not an interactive tool

Beyond MapReduce

There are some tools that extend MapReduce to overcome some of its limitations.

Data access tools such as Pig or Hive are high level interacting tools with SQL-like syntax that greatly simplify the interaction with the user.

Also, solutions like Spark are capable of providing high interactivity with real-time tasks.

Pig

Pig

Pig is a a platform for creating MapReduce programs using Hadoop trough a high level scripting language.

Such language is called *Pig Latin*, and it excels at describing data analysis problems as data flows.

A component, known as *Pig Engine*, accepts the Pig Latin scripts as input and converts them into MapReduce jobs, in an efficient way.

Pig is complete, in the sense that you can do all the required data manipulations within Hadoop by just using Pig.

Pig is a component to build much larger, much more complex applications that can tackle some real business problems.

Pig usage benefits

Using Pig Latin, programmers can perform MapReduce tasks easily without having to type complex code in Java.

Pig Latin is a SQL-like language, so it is easy to learn it when you are already familiar with SQL.

The Pig Engine is capable of autonomously optimizing a sequence of operations when translating them to MapReduce jobs, so the user doesn't have to worry about performance.

Pig provides many built-in functions to support data operations such as joins, filters, ordering, etc. In addition, it also provides nested data types like tuples, bags, and maps that are missing from MapReduce.

Pig components

- *Parser:* Pig scripts are read by the Parser that checks their syntax. The output is a DAG (Directed Acyclic Graph). In the DAG, the logical operators of the script are represented as the nodes and the data flows are represented as edges.
- *Optimizer:* The logical plan (DAG) is passed to the logical optimizer, which carries out optimizations such as projection and pushdown.
- *Compiler:* The compiler compiles the optimized logical plan into a series of MapReduce jobs.
- *Execution engine:* MapReduce jobs are finally submitted to Hadoop, which executes them producing the desired results.

Apache Pig execution modes

You can run Apache Pig in two modes: Local Mode and HDFS mode.

Local Mode: In this mode, all the files are stored in the local file system, and the tasks are run on local host. There is no need of Hadoop or HDFS (testing purpose).

MapReduce Mode: In this mode, we load or process the data that exists in the Hadoop File System (HDFS). Whenever we execute the Pig Latin statements to process the data, a MapReduce job is invoked in the back-end to perform a particular operation on the data that exists in the HDFS.

Apache Pig execution mechanisms

Apache Pig scripts can be executed in three ways, namely, interactive mode, batch mode, and embedded mode.

- *Interactive Mode (Grunt shell):* Apache Pig can be run using the Grunt shell, that allows to execute Pig Latin and HDFS commands in an interactive way.
- *Batch Mode (Script):* Apache Pig can be run in Batch mode by writing the Pig Latin script in a single file with .pig extension.
- *Embedded Mode (UDF):* Apache Pig provides the capability of specifying User Defined Functions in programming languages such as Java, and using them in a script.

Pig Latin data model - 1

The data model is fully nested and it allows non-atomic datatypes such as maps and tuples:

- *Atom:* Any single value in Pig Latin, irrespective of their datatype is known as an Atom. It is stored as a string. A simple atomic value is also known as a field.
  'raja', or '30'
- *Tuple:* A record that is composed of an ordered list of fields is known as a tuple. The fields can be of any type. A tuple is similar to a row in a table of a RDBMS.
  (Raja, 30)

Pig Latin data model - 2

- *Bag:* An unordered multiset of (non-unique) tuples, represented by {}. It is similar to a table in a RDBMS, but it is not necessary for every tuple to contain the same number of fields or for the fields in the same position (column) to have the same type: {(Raja, 30) , (Mohammad, 45)}. Bags can be nested (inner bags): {Raja, 30, {9848022338, raja@gmail.com}}
- *Map:* A set of key-value pairs. The key needs to be of type chararray and should be unique. The value may be of any type. It is represented by [].
- *Relation:* is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

Pig Latin data types

Simple Types	**Description**	**Example**
int	Signed 32-bit integer	10
long	Signed 64-bit integer	Data: 10L o
float	32-bit floating point	Data: 10.5F 10.5E2F
double	64-bit floating point	Data: 10.5 or 10.5e2
chararray	string UTF-8	hello world
bytearray	Byte array (blob)	
boolean	boolean	true/false (case ins)
datetime	datetime	1970-01-01T00: ...
biginteger	Java BigInteger	200000000000
bigdecimal	Java BigDecimal	33.456783321323 ...
Complex Types		
tuple	An ordered set of fields.	(19,2)
bag	An collection of tuples.	{(19,2), (18,1)}
map	A set of key value pairs.	[open#apache]

Pig basics - Loading

A job written in Pig Latin consist of a series of steps, each generating a *relation*, basically a tabular dataset.

Every command or *statement* ends with a ;

The function PigStorage is in charge of loading data:

Stocks = LOAD '/data/finance/stocks' using PigStorage(',') AS (ticker:chararray, date:chararray, open:double, close:double, low:double, high:double, volume:int);

The LOAD function allows to specify the data delimiter, and the relation schema through the name and the data type.

Pig basics - Filtering

Dataset names can be reused and assigned to a new relation.

With this command we extract a particular stock data from the relation:

```
Stocks = FILTER Stocks by ticker eq 'FCA.MI';
```

The result can be checked with the command DUMP, that shows the dataset on the shell. Also, it provides some log information regarding the execution of the MapReduce job.

Pig basics - Storage

The output can be saved using the command STORE.

With this command we save the output of the previous command, using the specified delimiter:

STORE Stocks INTO '/data/finance/prices_fca' USING PigStorage (',');

This is the last step of a simple process of extracting, filtering and storing data with Pig.

Pig basics - Data management

LOAD and STORE rely on function which defines a structure on data.

The default function is PigStorage, that is designed for managing structured text files on HDFS.

But there are several built in functions for reading and writing data in different formats:

- AvroStorage
- AccumuloStorage
- BinStorage
- HBaseStorage
- JsonLoader/JsonStorage
- OrcStorage
- TextLoader

Pig basics - Functions

PigLatin comes with a series of predefined mathematical functions to process the data ABS, ROUND, etc., while others are for strings TRIM, UPPER, LOWER, RegEx, and dates CurrentTime, GetDay, GetYear, etc.

There are also aggregating functions similar to the ones in SQL:

- AVG
- CONCAT
- COUNT
- SUM
- MIN, MAX
- SIZE
- TOKENIZE

Pig commands - 1

FILTER
Selects tuples from a relation based on some condition.
Syntax
alias = FILTER alias BY expression;
Example
Stocks = FILTER Stocks BY ticker == 'FCA.MI';

FOREACH
Generates data transformations on the rows of a relation.
Syntax
alias = FOREACH alias GENERATE expression [AS schema];
Example
A = FOREACH Stocks GENERATE high-low AS diff:double;

Pig commands - 2

GROUP
Groups the data, similarly to the SQL GROUP BY operation.
Syntax
alias = GROUP alias {ALL | BY expression} [USING 'collected' | 'merge'] [PARTITION BY partitioner] [PARALLEL n];
Example
A = GROUP Stocks BY ticker;
A = FOREACH A GENERATE group, AVG(open);

ALL generates a single group for the whole relation.

USING allows to fine-tune the job performance.

PARTITION BY allows to specifiy a Hadoop partitioner.

PARALLEL defines the number of reduce tasks.

Pig commands - 3

JOIN

Performs a join of two relations based on common field values.

Syntax

alias = JOIN left-alias BY left-alias-column
[LEFT | RIGHT | FULL] [OUTER], right-alias BY
right-alias-column [USING 'replicated' | 'bloom' | 'skewed' |
'merge'] [PARTITION BY partitioner] [PARALLEL n];

Example

```
FCA = FILTER Stocks BY ticker == 'FCA.MI';
FCA = FOREACH FCA GENERATE date, close;
ENI = FILTER Stocks BY ticker == 'ENI.MI';
ENI = FOREACH ENI GENERATE date, close;
J = JOIN FCA BY date, ENI BY date;
```

Pig commands - 4

ORDER BY
Sorts a relation based on one or more fields.
Syntax
alias = ORDER alias BY {field_alias [ASC | DESC] [, field_alias [ASC | DESC] . . .]}[PARALLEL n];
Example
Sorted = ORDER Stocks BY ticker ASC, date ASC;

UNION
Computes the union (multiset) of two or more relations.
Syntax
alias = UNION [ONSCHEMA] alias, alias [, alias . . .] [PARALLEL n];
Notes
ONSCHEMA allows to perform the UNION operation based on the field names, instead of considering their position.

Pig commands - 5

MAPREDUCE
Executes a native MapReduce job inside a Pig Latin script.
Syntax
alias1 = MAPREDUCE 'mr.jar' STORE alias2 INTO 'inputLocation' [USING storeFunc] LOAD 'outputLocation' [USING loadFunc] AS schema ['params, . . .];
Example
A = LOAD 'textfile.txt';
B = MAPREDUCE 'wordcount.jar' STORE A INTO '/data/input' LOAD '/data/output' AS (word:chararray, count:int) 'org.myorg.WordCount inputDir outputDir';

'params' encodes the parameters for MapReduce job.

Pig commands - 6

CUBE AND ROLLUP

Syntax

alias = CUBE alias BY CUBE expression | ROLLUP expression
, [CUBE expression | ROLLUP expression] [PARALLEL n];

CUBE computes aggregates for all possible combinations of specified group by dimensions. The number of group by combinations generated by CUBE for n dimensions is 2^n.

Example

Cubedstocks = CUBE Stocks BY CUBE (ticker, date);
Result = FOREACH Cubedstocks GENERATE
FLATTEN(group), AVG(open);

For a sample input tuple ('FCA.MI', '2020-04-16', 7.18 , 7.11, 7.06, 7.3', 7861000), it outputs: ('FCA.MI', '2020-04-16', 7.18), ('FCA.MI', , 7.18), (, '2020-04-16', 7.18), (, , 7.18)

Pig commands - 7

ROLLUP operation computes multiple levels of aggregates based on hierarchical ordering of specified group by dimensions. ROLLUP is useful when there is hierarchical ordering on the dimensions. The number of group by combinations generated by rollup for n dimensions is $n + 1$.

Considering a sample input tuple (car, 2012, midwest, ohio, columbus, 4000), the following query

```
salesinp = LOAD '/pig/data/salesdata' USING PigStorage(',') AS
    (product:chararray, year:int, region:chararray, state:chararray, city:chararray, sales:long);
rolledup = CUBE salesinp BY ROLLUP(region,state,city);
result = FOREACH rolledup GENERATE FLATTEN(group), SUM(cube.sales) AS totalsales;
```

will produce:

```
(midwest,ohio,columbus,4000)
(midwest,ohio,,4000)
(midwest,,,4000)
(,,,4000)
```

CountWord with Pig

```
lines = LOAD '/user/hadoop/HDFS_File.txt' AS
    (line:chararray);
words = FOREACH lines GENERATE
    FLATTEN(TOKENIZE(line)) as word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped
    GENERATE group, COUNT(words);
DUMP wordcount;
```

CountWord with Pig

The considered Pig script first splits each line into words using the TOKENIZE operator, that creates a bag of words. Using the FLATTEN function, the bag is then converted into a tuple.

In the third statement, the words are grouped together so that the count can be computed, which is done in fourth statement.

So, with just 5 lines in Pig Latin language, we have solved the word count problem very easily.