Distributed architectures for big data processing and analytics

4/2/23

Table of contents

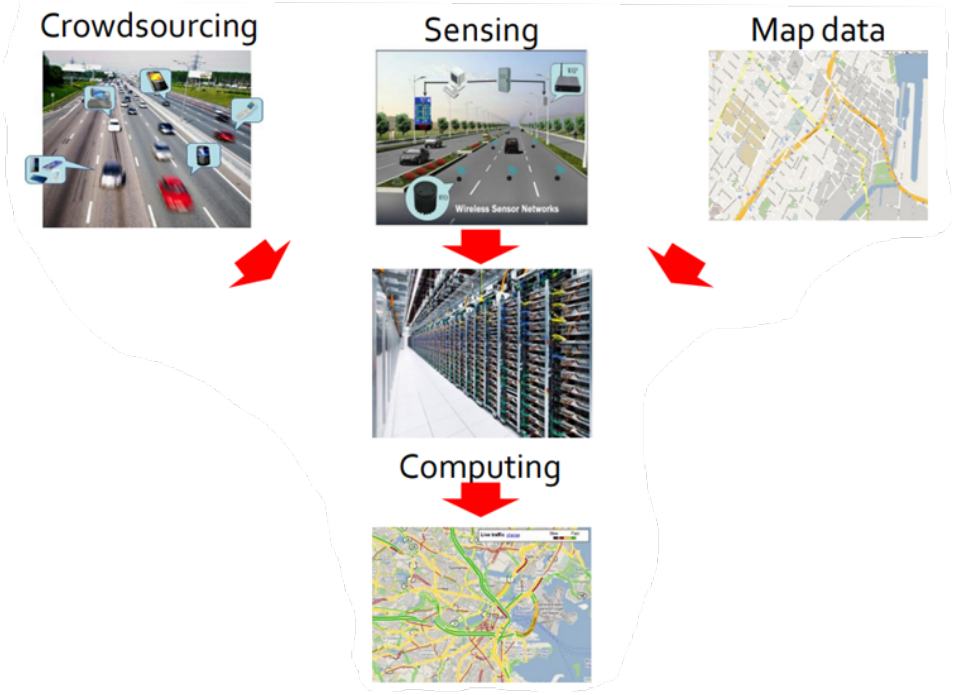
1 Index

2 Introduction to Big data

- User generated content: social networks (web and mobile)
- Health and scientific computing
- Log files: web server log files, machine system log files
- Internet of Things (IoT): sensor networks, RFIDs, smart meters

2.1 Example of Big Data at work

Figure 2.1: Bigdata example



3 The five Vs of Big Data

- Volume: scale of data
- Variety: different forms of data
- Velocity: analysis of streaming data
- Veracity: uncertainty of data
- Value: exploit information provided by data

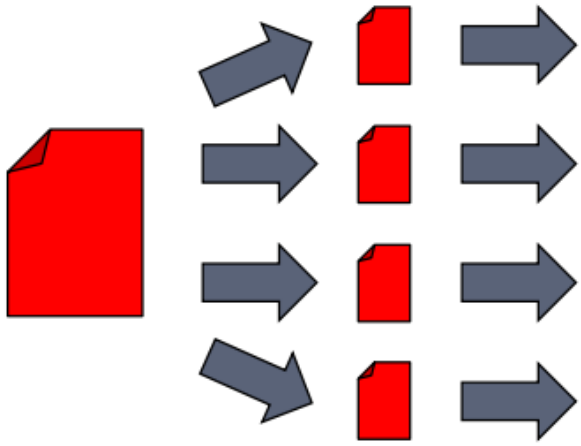
4 The bottleneck and the solution

4.1 Bottleneck

- Processors process data
- Hard drives store data
- We need to transfer data from the disk to the processor

4.2 Solution

- Transfer the processing power to the data
- Multiple distributed disks: each one holding a portion of a large dataset



- Process in parallel different file portions from different disks

5 Big data architectures

From Data Architecture Guide in Microsoft Learn

A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems…
Big data solutions typically involve one or more of the following types of workload:

- Batch processing of big data sources at rest
- Real-time processing of big data in motion
- Interactive exploration of big data
- Predictive analytics and machine learning

Consider big data architectures when you need to:

- Store and process data in volumes too large for a traditional database
- Transform unstructured data for analysis and reporting
- Capture, process, and analyze unbounded streams of data in real time, or with low latency

6 Lambda architecture

The most frequently used big data architecture is the Lambda Architecture. The lambda architecture was proposed by Nathan Marz in 2011.

6.1 Definitions

From Nathan Marz

The past decade has seen a huge amount of innovation in scalable data systems. These include large-scale computation systems like Hadoop and databases such as Cassandra and Riak. These systems can handle very large amounts of data, but with serious trade-offs.Hadoop, for example, can parallelize large-scale batch computations on very large amounts of data, but the computations have high latency. You don't use Hadoop for anything where you need low-latency results.
NoSQL databases like Cassandra achieve their scalability by offering you a much more limited data model than you're used to with something like SQL. Squeezing your application into these limited data models can be very complex. And because the databases are mutable, they're not human-fault tolerant.
These tools on their own are not a panacea. But when intelligently used in conjunction with one another, you can produce scalable systems for arbitrary data problems with human-fault tolerance and a minimum of complexity. This is the Lambda Architecture you'll learn throughout the book.

From What is Lambda Architecture? article in Databricks website

Lambda architecture is a way of processing massive quantities of data (i.e. "Big Data") that provides access to batch-processing and stream-processing methods with a hybrid approach.
Lambda architecture is used to solve the problem of computing arbitrary functions.

From Lambda architecture article in Wikipedia

Lambda architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods.
This approach to architecture attempts to balance latency, throughput, and fault tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data. The two view outputs may be joined before presentation.
Lambda architecture depends on a data model with an append-only, immutable data source that serves as a system of record. It is intended for ingesting and processing timestamped events that are appended to existing events rather than overwriting them. State is determined from the natural time-based ordering of the data.

6.2 Requirements

Fault-tolerant against both hardware failures and human errors Support variety of use cases that include low latency querying as well as updates Linear scale-out capabilities Extensible, so that the system is manageable and can accommodate newer features easily

6.3 Queries

$$\textbf{query} = \textbf{function}(\textbf{all data})$$

Some query properties

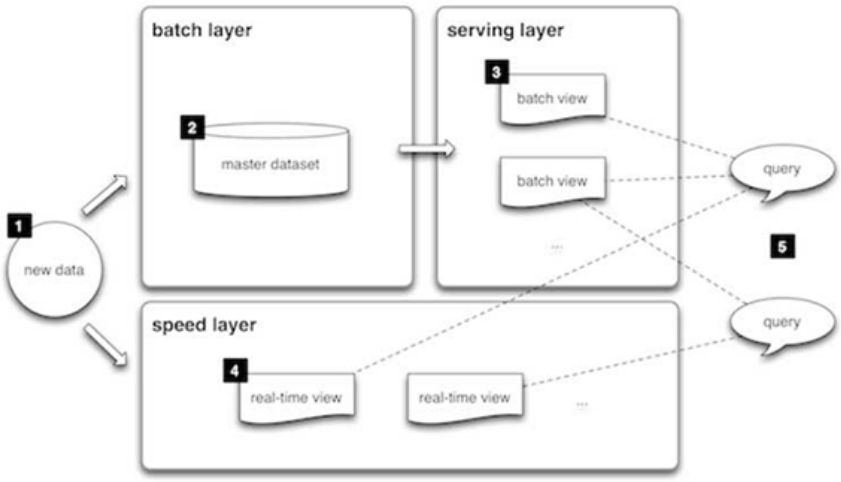
- Latency: the time it takes to run a query
- Timeliness: how up to date the query results are (freshness and consistency)
- Accuracy: tradeoff between performance and scalability (approximations)

It is based on two data paths:

- Cold path (batch layer)
  - It stores all of the incoming data in its raw form and performs batch processing on the data
  - The result of this processing is stored as batch views
- Hot path (speed layer)
  - It analyzes data in real time
  - This path is designed for low latency, at the expense of accuracy

6.4 Basic structure

Figure 6.1: General Lambda architecture



1. All data entering the system is dispatched to both the batch layer and the speed layer for processing
2. The batch layer has two functions:

   i) managing the master dataset(an immutable, append-only set of raw data), and
   ii) to pre-compute the batch views

3. The serving layer indexes the batch views so that they can be queried in low-latency, ad-hoc way
4. The speed layer compensates for the high latency of updates to the serving layer and deals with recent data only
5. Any incoming query can be answered by merging results from batch views and real-time views (e.g., the query looks at the serving layer for days until today, and looks at the speed layer for today's data).

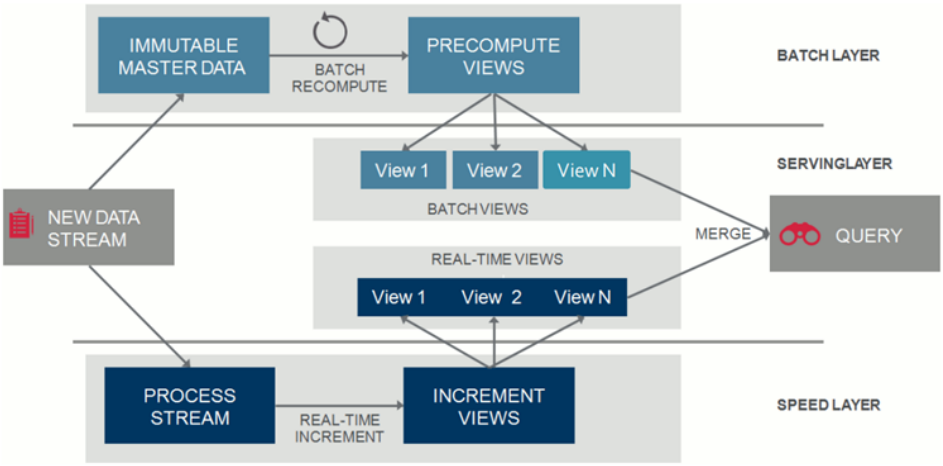
6.5 Detailed view

Structure similar to the one described before

0. Data stream
1. Batch layer

   - immutable data

Figure 6.2: More detailed of Lambda architecture

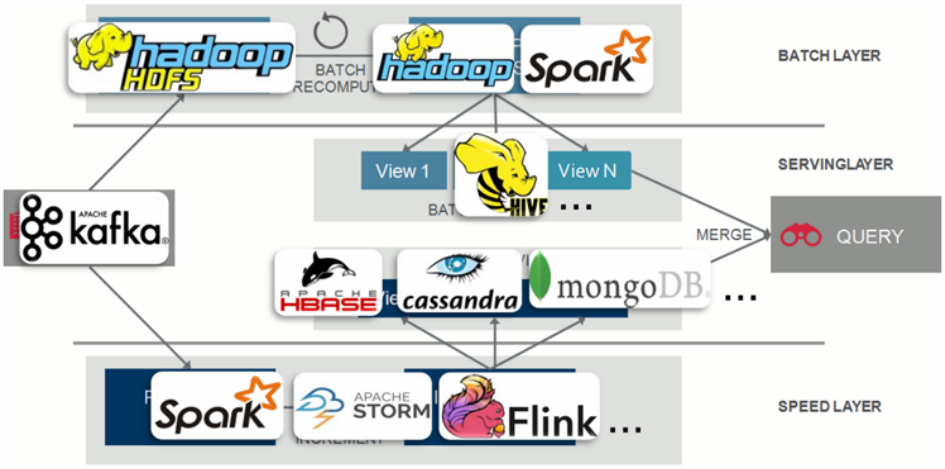


- precompute views

2. Real-time layer
   - process stream
   - increment views
3. Serving layer

6.5.1 Possible instances

Figure 6.3: More detailed of Lambda architecture



In general, the technologies used are

0. Data stream: Kafka
1. Batch layer:
    - immutable data: Hadoop HDFS
    - precompute views: Hadoop, Spark
    - views: Hive (it is a distributed relational database; SQL-like query language can be used)
2. Real-time layer:
    - process stream and increment views:
        - Spark (it has a module available for managing stream data)
        - Apache Storm (pros: true real-time; cons: sometimes it approximates)
        - Flink (used stream data analysis)
    - views: HBase, Cassandra, MongoDB
3. Serving layer

In general: choose the most suitable technology, but also be able to adapt on what's available.

7 HDFS and Hadoop: command line commands

The content of a HDFS file can be accessed by means of

- Command line commands
- A basic web interface provided by Apache Hadoop. The HDFS content can only be browsed and its files downloaded from HDFS to the local file system, while uploading functionalities are not available.
- Vendor-specific web interfaces providing a full set of functionalities (upload, download, rename, delete, …) (e.g., the HUE web application of Cloudera).

7.1 User folder

Each user of the Hadoop cluster has a personal folder in the HDFS file system. The default folder of a user is

```
/user/username
```

7.2 Command line

The hdfs command can be executed in a Linux shell to read/write/modify/delete the content of the distributed file system. The parameters/arguments of hdfs command are used to specify the operation to execute.

7.2.1 Content of a folder

To list the content of a folder of the HDFS file system, use

```
hdfs dfs -ls folder
```

> **Example**
>
> The command `hdfs dfs -ls /user/garza` shows the content (list of files and folders) of the `/user/garza` folder.
> The command `hdfs dfs -ls .` shows the content of the current folder (i.e., the content of `/user/current_username`).
> Notice that the mapping between the local linux user and the user of the cluster is based on
>
> - A Kerberos ticket if Kerberos is active
> - Otherwise the local linux user is considered

7.2.2 Content of a file

To show the content of a file in the HDFS file system, use

```
hdfs dfs -cat file_name
```

> **Example**
>
> The command `hdfs dfs -cat /user/garza/document.txt` shows the content of the `/user/garza/document.txt` file stored in HDFS.

7.2.3 Copy a file from local to HDFS

To copy a file from the local file system to the HDFS file system, use

```
hdfs dfs -put local_file HDFS_path
```

> **Example**
>
> The command `hdfs dfs -put /data/document.txt /user/garza/` copies the local file `/data/document.txt` in the folder `/user/garza` in HDFS.

7.2.4 Copy a file from HDFS to local

To copy a file from the HDFS file system to the local file system, use

```
hdfs dfs -get HDFS_path local_file
```

> **Example**
>
> The command `hdfs dfs -get /user/garza/document.txt /data/` copies the HDFS file `/user/garza/document.txt` in the local file system folder `/data/`.

7.2.5 Delete a file

To delete a file from the HDFS file system, use

```
hdfs dfs -rm HDFS_path
```

> **Example**
>
> The command `hdfs dfs -rm /user/garza/document.txt` delete from HDFS the file `/user/garza/document.txt`

7.2.6 Other commands

There are many other linux-like commands, for example

- rmdir
- du
- tail

See the HDFS commands guide for a complete list.

8 Hadoop

The Hadoop programs are executed (submitted to the cluster) by using the hadoop command. It is a command line program, characterized by a set of parameters, such as

- the name of the jar file containing all the classes of the MapReduce application we want to execute
- the name of the Driver class
- the parameters/arguments of the MapReduce application

8.1 Example

The following command executes/submits a MapReduce application

```
hadoop jar MyApplication.jar \
it.polito.bigdata.hadoop.DriverMyApplication \
1 inputdatafolder/ outputdatafolder/
```

- It executes/submits the application contained in `MyApplication.jar`
- The Driver Class is `it.polito.bigdata.hadoop.DriverMyApplication`
- The application has three arguments
  - Number of reducers (`1`)
  - Input data folder (`inputdatafolder/`)
  - Output data folder (`outputdatafolder/`)

9 Introduction to Hadoop and MapReduce

9.1 Data volumes

- The amount of data increases every day
- Some numbers (2012):
  - Data processed by Google every day: 100+ PB
  - Data processed by Facebook every day: 10+ PB
- To analyze them, systems that scale with respect to the data volume are needed

> **Example: Google**
>
> Consider this situation: you have to analyze 10 billion web pages, and the average size of a webpage is 20KB. So
>
> - The total size of the collection: 10 billion x 20KBs = 200TB
> - Assuming the usage of HDD hard disk (read bandwidth: 150MB/sec), the time needed to read all web pages (without analyzing them) is equal to 2 million seconds (i.e., more than 15 days).
> - Assuming the usage of SSD hard disk (read bandwidth: 550MB/sec), the time needed to read all web pages (without analyzing them) is equal to 2 million seconds (i.e., more than 4 days).
> - A single node architecture is not adequate

9.2 Failures

Failures are part of everyday life, especially in a data center. A single server stays up for 3 years (~1000 days). Statistically

- With 10 servers: 1 failure every 100 days (~3 months)
- With 100 servers: 1 failure every 10 days
- With 1000 servers: 1 failure/day

The main sources of failures

- Hardware/Software
- Electrical, Cooling, …
- Unavailability of a resource due to overload

> **Examples**
>
> LALN data [DSN 2006]
>
> - Data for 5000 machines, for 9 years
> - Hardware failures: 60%, Software: 20%, Network 5%
>
> DRAM error analysis [Sigmetrics 2009]
>
> - Data for 2.5 years
> - 8% of DIMMs affected by errors
>
> Disk drive failure analysis [FAST 2007]
>
> - Utilization and temperature major causes of failures

Failure types

- Permanent (e.g., broken motherboard)
- Transient (e.g., unavailability of a resource due to overload)

9.3 Network bandwidth

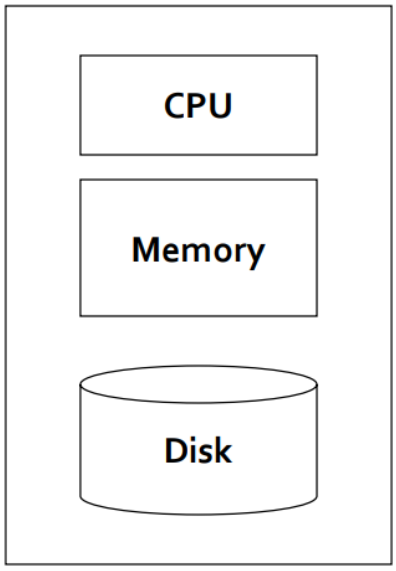
Network becomes the bottleneck if big amounts of data need to be exchanged between nodes/servers. Assuming a network bandwidth (in a data centre) equal to 10 Gbps, it means that moving 10 TB from one server to another would take more than 2 hours. So, data should be moved across nodes only when it is indispensable.

Instead of moving data to the data centre, the code (i.e., programs) should be moved between the nodes: this approach is called **Data Locality**, and in this way very few MBs of code are exchanged between the severs, instead of huge amount of data.

10 Architectures

10.1 Single node architecture

Figure 10.1: Single node architecture



Small data: data can be completely loaded in main memory.

Large data: data can not be completely loaded in main memory.

- Load in main memory one chunk of data at a time, process it and store some statistics
- Combine statistics to compute the final result

10.2 Cluster architecture

To overcome the previously explained issues, a new architecture based on clusters of servers (i.e., data centres) has been devised. In this way:

- Computation is distributed across servers

Figure 10.2: Single node architecture: Machine Learning and Statistics

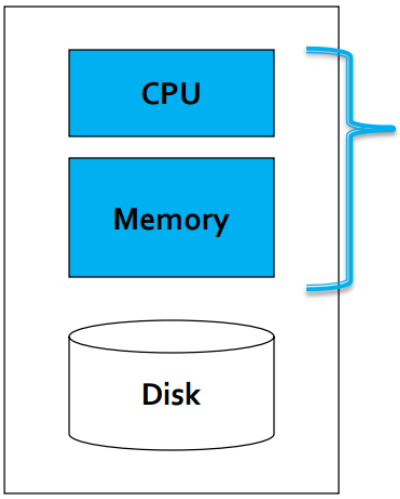
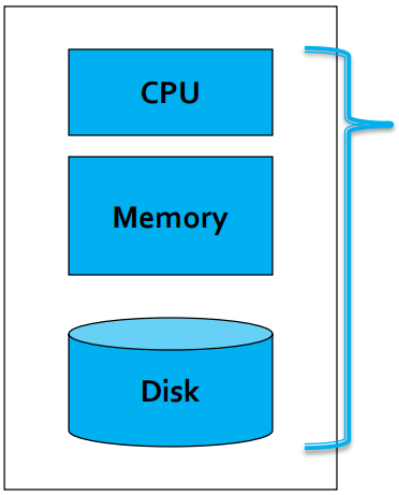


Figure 10.3: Single node architecture: "Classical" data mining"



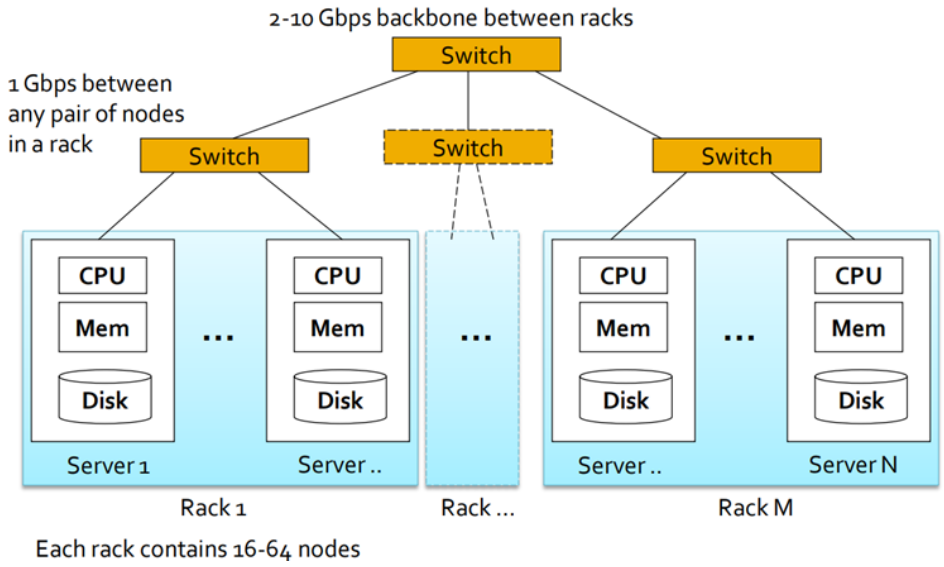
- Data are stored/distributed across servers

The standard architecture in the Big data context (2012) is based on

- Cluster of commodity Linux nodes/servers (32 GB of main memory per node)
- Gigabit Ethernet interconnection

10.2.1 Commodity cluster architecture

Figure 10.4: Commodity cluster architecture



The servers in each rack are very similar to each other, so that the servers would take the same time to process the data and none of them will become a bottleneck for the overall processing.

Notice that

- In each rack, the servers are directly connected with each other in pairs
- Racks are directly connected with each other in pairs

10.3 Scalability

Current systems must scale to address

- The increasing amount of data to analyze
- The increasing number of users to serve

- The increasing complexity of the problems

Two approaches are usually used to address scalability issues

- Vertical scalability (scale up)
- Horizontal scalability (scale out)

10.3.1 Scale up vs. Scale out

- Vertical scalability (*scale up*): **add more power/resources** (i.e., main memory, CPUs) to a **single node** (high-performing server). The cost of super-computers is not linear with respect to their resources: the marginal cost increases as the power/resources increase.
- Horizontal scalability (*scale out*): **add more nodes** (commodity servers) to a system. The cost scales approximately linearly with respect to the number of added nodes. But data center efficiency is a difficult problem to solve.

For data-intensive workloads, a large number of commodity servers is preferred over a small number of high-performing servers, since, at the same cost, it is possible to deploy a system that processes data more efficiently and is more fault-tolerant.

Horizontal scalability (scale out) is preferred for big data applications, but distributed computing is hard: new systems hiding the complexity of the distributed part of the problem to developers are needed.

10.4 Cluster computing challenges

1. Distributed programming is hard
    - Problem decomposition and parallelization
    - Task synchronization
2. Task scheduling of distributed applications is critical: assign tasks to nodes by trying to
    - Speed up the execution of the application
    - Exploit (almost) all the available resources
    - Reduce the impact of node failures
3. Distributed data storage

How to store data persistently on disk and keep it available if nodes can fail? **Redundancy** is the solution, but it increases the complexity of the system.

4. Network bottleneck

Reduce the amount of data send through the network by moving computation and code to data.

> **Distributed computing history**
>
> Distributed computing is not a new topic
>
> - HPC (High-performance computing) ~1960
> - Grid computing ~1990
> - Distributed databases ~1990
>
> Hence, many solutions to the mentioned challenges are already available, but we are now facing big data-driven problems: the former solutions are not adequate to address big data volumes.

10.5 Typical Big data problem

The typical way to address a Big Data problem (given a collection of historical data)

- Iterate over a large number of records/objects
- Extract something of interest from each record/object
- Aggregate intermediate results
- Generate final output

Notice that, if in the second step it is needed to have some kind of knowledge of what's in the other records, this Big data framework is not the best solution: the computations on isolated records is not possible anymore, and so this whole architecture is not suitable.

The challenges:

- Parallelization
- Distributed storage of large data sets (Terabytes, Petabytes)
- Node Failure management
- Network bottleneck
- Diverse input format (data diversity & heterogeneity)

11 Apache Hadoop

It is scalable fault-tolerant distributed system for Big Data

- Distributed Data Storage
- Distributed Data Processing

It borrowed concepts/ideas from the systems designed at Google (Google File System for Google's MapReduce). It is open source project under the Apache license, but there are also many commercial implementations (e.g., Cloudera, Hortonworks, MapR).

i Hadoop history

Table 11.1: Timeline

Date	Event
Dec 2004	Google published a paper about GFS
July 2005	Nutch uses MapReduce
Feb 2006	Hadoop becomes a Lucene subproject
Apr 2007	Yahoo! runs it on a 1000-node cluster
Jan 2008	Hadoop becomes an Apache Top Level Project
Jul 2008	Hadoop is tested on a 4000 node cluster
Feb 2009	The Yahoo! Search WebMap is a Hadoop application that runs on more than 10,000 core Linux cluster
Jun 2009	Yahoo! made available the source code of its production version of Hadoop
2010	Facebook claimed that they have the largest Hadoop cluster in the world with 21 PB of storage
Jul 27, 2011	Facebook announced the data has grown to 30 PB

Who uses/used Hadoop

- Amazon
- Facebook
- Google
- IBM
- Joost
- Last.fm
- New York Times
- PowerSet
- Veoh
- Yahoo!

11.1 Hadoop vs. HPC

Hadoop

- Designed for Data intensive workloads
- Usually, no CPU demanding/intensive tasks

HPC (High-performance computing)

- A supercomputer with a high-level computational capacity (performance of a supercomputer is measured in floating-point operations per second (FLOPS))
- Designed for CPU intensive tasks
- Usually it is used to process "small" data sets

11.2 Main components

Core components of Hadoop:

1. Distributed Big Data Processing Infrastructure based on the MapReduce programming paradigm
    - Provides a high-level abstraction view: programmers do not need to care about task scheduling and synchronization
    - Fault-tolerant: node and task failures are automatically managed by the Hadoop system
2. HDFS (Hadoop Distributed File System)
    - High availability distributed storage

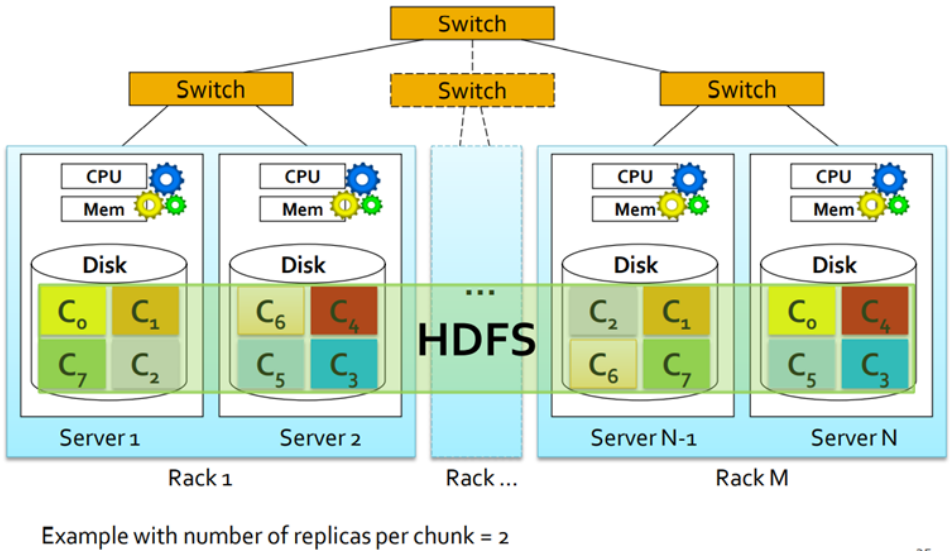
- Fault-tolerant

Hadoop virtualizes the file system, so that the interaction resembles a local file system, even if this case it spans on multiple disks on multiple servers.

So Hadoop is in charge of:

- splitting the input files
- store the data in different servers
- managing the reputation of the blocks

Figure 11.1: Hadoop main components



Notice that, in this example, the number of replicas (i.e., the number of copies) of each block (e.g., C_0, C_1, C_6, etc.) is equal to two. Multiple copies are needed to correctly manage server failures: two copies are never stored in the same server.

Notice that, with 2 copies of the same file, the user is always sure that 1 failure can be managed with no interruptions in the data processing and without the risk of losing data. In general, the number of failures that and HDFS can sustain with no repercussions is equal to (**number of copies**) $-$ 1.

When a failure occures, Hadoop immediately starts to create new copies of the data, to reach again the set number of replicas.

11.3 Distributed Big data processing infrastructure

Hadoop allows to separate the *what* from the *how* because Hadoop programs are based on the MapReduce programming paradigm:

- MapReduce abstracts away the "distributed" part of the problem (scheduling, synchronization, etc), so that programmers can focus on the *what*;
- the distributed part (scheduling, synchronization, etc) of the problem is handled by the framework: the Hadoop infrastructure focuses on the *how*.

But an in-depth knowledge of the Hadoop framework is important to develop efficient applications: the design of the application must exploit data locality and limit network usage/data sharing.

11.4 HDFS

HDFS is the standard Apache Hadoop distributed file system. It provides global file namespace, and stores data redundantly on multiple nodes to provide persistence and availability (fault-tolerant file system).

The typical usage pattern for Hadoop:

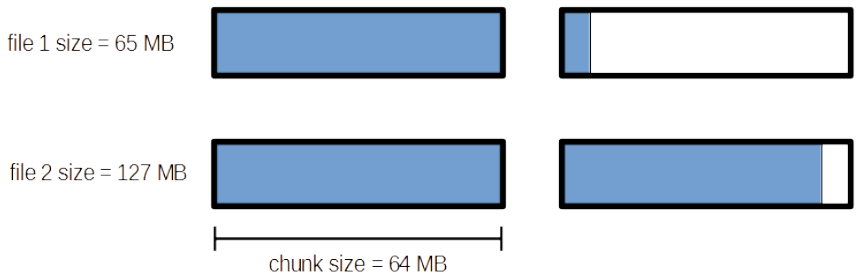
- huge files (GB to TB);
- data is rarely updated (create new files or append to existing ones);
- reads and appends are common, and random read/write operations are not performed.

Each file is split in **chunks** (also called **blocks**) that are spread across the servers.

- Each chunck is replicated on different servers (usually there are 3 replicas per chunk), ensuring persistence and availability. To further increase persistence and availability, replicas are stored in different racks, if it possible.
- Each chunk contains a part of the content of *one single file*. It is not possible to have the content of two files in the same chunk/block
- Typically each chunk is 64-128 MB, and the chunk size is defined when configuring Hadoop.

Example

Figure 11.2: 2 files in 4 chunks



Each square represents a chunk in the HDFS. Each chunk contains 64 MB of data, so file 1 (65 MB) sticks out by 1 MB from a single chunk, while file 2 (127 MB) does not completely fill two chunks. The empty chunk portions are not filled by any other file. So, even if the total space occupied from the files would be 192 MB (3 chunks), the actual space they occupy is 256 (4 chunks): Hadoop does not allow two files to occupy the same chunk, so that two different processes would not try to access a block at the same time.

The Master node, (a.k.a., *Name Nodes* in HDFS) is a special node/server that

- Stores HDFS metadata (e.g., the mapping between the name of a file and the location of its chunks)
- Might be replicated (to prevent stoppings due to the failure of the Master node)

Client applications can access the file through HDFS APIs: they talk to the master node to find data/chuck servers associated with the file of interest, and then connect to the selected chunk servers to access data.

Hadoop ecosystem

The HDFS and the YARN scheduler are the two main components of Hadoop, however there are modules, and each project/system addresses one specific class of problems.

- Hive: a distributed relational database, based on MapReduce, for querying data stored in HDFS by means of a query language based on SQL;
- HBase: a distributed column-oriented database that uses HDFS for storing data;
- Pig: a data flow language and execution environment, based on MapReduce, for exploring very large datasets;
- Sqoop: a tool for efficiently moving data from traditional relational databases and

external flat file sources to HDFS;
- ZooKeeper: a distributed coordination service, that provides primitives such as distributed locks.
- …

The integration of these components with Hadoop is not as good as the integration of the Spark components with Spark.

12 MapReduce: introduction

12.1 Word count

Input	Problem	Output
a large textual file of words	count the number of times each distinct word appears in the file	a list of pairs word, number, counting the number of occurrences of each specific word in the input file

12.1.1 Case 1: Entire file fits in main memory

A traditional single node approach is probably the most efficient solution in this case. The complexity and overheads of a distributed system affects the performance when files are "small" ("small" depends on the available resources).

12.1.2 Case 2: File too large to fit in main memory

How to split this problem in a set of (almost) independent sub-tasks, and execute them in parallel on a cluster of servers?

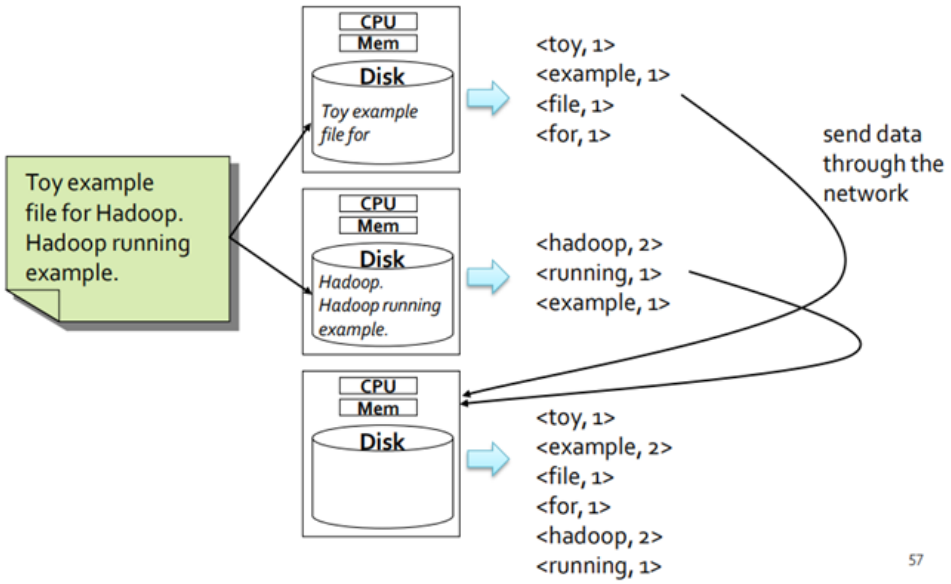
Assuming that

- The cluster has 3 servers
- The content of the input file is: "Toy example file for Hadoop. Hadoop running example"
- The input file is split into 2 chunks
- The number of replicas is 1

The problem can be easily parallelized:

1. Each server processes its chunk of data and counts the number of times each word appears in its own chunk
    - Each server can execute its sub-task independently from the other servers of the cluster: asynchronization is not needed in this phase

Figure 12.1: Word count solution



- The output generated from each chunk by each server represents a partial result

2. Each server sends its local (partial) list of pairs $<$ **word**, **number of occurrences in its chunk** $>$ to a server that is in charge of aggregating all local results and computing the global result. The server in charge of computing the global result needs to receive all the local (partial) results to compute and emit the final list: a synchronization operation is needed in this phase.

Assume a more realistic situation

- The file size is 100 GB and the number of distinct words occurring in it is at most 1000
- The cluster has 101 servers
- The file is spread across 100 servers (1 server is the Master node) and each of these servers contains one (different) chunk of the input file (i.e., the file is optimally spread across 100 servers, and so each server contains 1/100 of the file in its local hard drives)

12.1.2.1 Complexity

- Each server reads 1 GB of data from its local hard drive (it reads one chunk from HDFS): the time needed to process the data is equal to a few seconds;
- Each local list consists of at most 1,000 pairs (because the number of distinct words is 1,000): each list consists of a few MBs;

- The maximum amount of data sent on the network is 100 times the size of a local list (number of servers x local list size): the MBs that are moved through the network consists of some MBs.

So, the critical step is the first one: the result of this phase should be as small as possible, to reduce the data moving between nodes during the following phase.

Is also the aggregating step parallelizable? Yes, in the sense that the key-value pairs associated with the same key are sent to the same server in order to apply the aggregating function. So, different servers work in parallel, computing the aggregations on different keys.

12.1.2.2 Scalability

Scalability can be defined along two dimensions

- In terms of **data**: given twice the amount of data, the word count algorithm takes approximately no more than twice as long to run. Each server has to process twice the data, and so execution time to compute local list is doubled.
- In terms of **resources**: given twice the number of servers, the word count algorithm takes approximately no more than half as long to run. Each server processes half of the data, and execution time to compute local list is halved.

We are assuming that the time needed to send local results to the node in charge of computing the final result and the computation of the final result are considered negligible in this running example. However, notice that frequently this assumption is not true, indeed it depends on the complexity of the problem and on the ability of the developer to limit the amount of data sent on the network.

12.2 MapReduce approach key ideas

- Scale "out", not "up": increase the number of servers, avoiding to upgrade the resources (CPU, memory) of the current ones
- Move processing to data: the network has a limited bandwidth
- Process data sequentially, avoid random access: seek operations are expensive. Big data applications usually read and analyze all input records/objects: random access is useless

12.2.1 Data locality

Traditional distributed systems (e.g., HPC) move data to computing nodes (servers). This approach cannot be used to process TBs of data, since the network bandwidth is limited So, Hadoop moves code to data: code (few KB) is copied and executed on the servers where the chunks of data are stored. This approach is based on "data locality".

12.3 Hadoop and MapReduce usage scope

Hadoop/MapReduce is designed for

- Batch processing involving (mostly) full scans of the input data
- Data-intensive applications
  - Read and process the whole Web (e.g., PageRank computation)
  - Read and process the whole Social Graph (e.g., LinkPrediction, a.k.a. "friend suggestion")
  - Log analysis (e.g., Network traces, Smart-meter data)

In general, MapReduce can be used when the same function is applied on multiple records **one at a time**, and its result then has to be aggregated.

> ⚠ Warning
>
> Notice that Hadoop/MapReduce is not the panacea for all Big Data problems. In particular, does not feet well
>
> - Iterative problems
> - Recursive problems
> - Stream data processing
> - Real-time processing

13 The MapReduce programming paradigm

The MapReduce programming paradigm is based on the basic concepts of Functional programming. Actually, MapReduce "implements" a subset of functional programming, and, because of this, the programming model appears quite limited and strict: everything is based on two "functions" with predefined signatures, that are Map and Reduce.

13.1 What can MapReduce do

Solving complex problems is difficult, however there are several important problems that can be adapted to MapReduce

- Log analysis
- PageRank computation
- Social graph analysis
- Sensor data analysis
- Smart-city data analysis
- Network capture analysis

13.2 Building blocks: Map and Reduce

MapReduce is based on two main "building blocks", which are the Map and Reduce functions.

- Map function: it is applied over each element of an input data set and emits a set of (key, value) pairs
- Reduce function: it is applied over each set of (key, value) pairs (emitted by the Map function) with the same key and emits a set of (key, value) pairs. This is the final result.

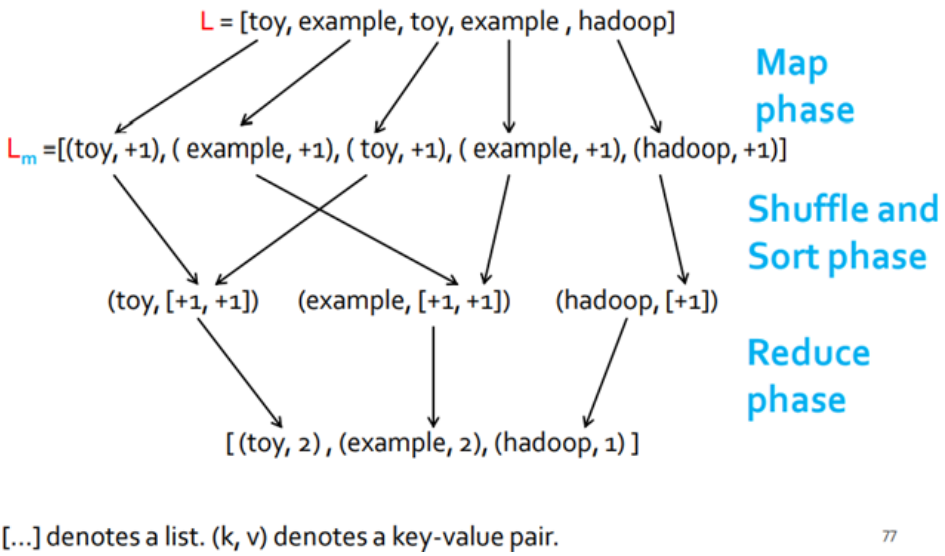
13.3 Solving the word count problem

Input	Problem	Output
a large textual file of words	count the number of times each distinct word appears in the file	a list of pairs word, number, counting the number of occurrences of each specific word in the input file

The input textual file is considered as a list L of words

$$L = [\text{toy}, \text{example}, \text{toy}, \text{example}, \text{hadoop}]$$

Figure 13.1: Word count running example



- Map phase: apply a function on each element of a list of key-value pairs (notice that the example above is not 100% correct: the elements of the list should also be key-value pairs);
- Shuffle and sort phase: group by key; in this phase the key-value pairs having the same key are collected together in the same node, but no computation is performed;
- Reduce phase: apply an aggregating function on each group; this step can be parallelized: one node may consider some keys, while another one considers others.

A key-value pair $(w, 1)$ is emitted for each word w in L.

In other words, the Map function m is

$$m(w) = (w, 1)$$

A new list of (key, value) pairs L_m is generated. Notice that, in this case the key-value pairs generated for each word is just one, but in other cases more than one key-value pair is generated from each element of the starting list.

Then, the key-value pairs in L_m are aggregated by key (i.e., by word w in the example).

13.3.1 Map

In the Map step, one group G_w is generated for each word w. Each group G_w is a key-list pair

$$(w, [\textbf{list of values}])$$

where [**list of values**] contains all the values of the pairs associated with the word w.

Considering the example, **[list of values]** is a list of $[1, 1, 1, ...]$, and, given a group G_w, the number of ones in $[1, 1, 1, ...]$ is equal to the occurrences of word w in the input file.

Notice that also the input of Map should be a list of key-value pairs. If a simple list of elements is passed to Map, Hadoop transforms the elements in key-value pairs, such that the value is equal to the element (e.g., the word) and the key is equal to the offset of the element in the input file.

13.3.2 Reduce

For each group G_w a key-value pair is emitted as follows

$$(w, \sum_{G_w} [\textbf{list of values}])$$

So, the result of the Reduce function is $r(G_w) = (w, \sum_{Gw} [\textbf{list of values}])$.

The resulting list of emitted pairs is the solution of the word count problem: in the list there is one pair (word w, number of occurrences) for each word in our running example.

13.4 MapReduce Phases

13.4.1 Map

The Map phase can be viewed as a transformation over each element of a data set. This transformation is a function m defined by developers, and it is invoked one time for each input element. Each invocation of m happens in isolation, allowing the parallelization of the application of m to each element of a data set in a straightforward manner.

The *formal definition* of Map is

$$(k_1, v_1) \rightarrow [(k_2, v_2)]$$

Notice that

- Since the input data set is a list of key-value pairs, the argument of the Map function is a key-value pair; so, the Map function N times, where N is the number of input key-value pairs;
- The Map function emits a list of key-value pairs for each input record, and the list can also be empty;
- No data is moved between nodes during this phase.

13.4.2 Reduce

The Reduce phase can be viewed as an aggregate operation. The aggregate function is a function r defined by developers, and it is invoked one time for each distinct key, aggregating all the values associated with it. Also the reduce phase can be performed in parallel and in isolation, since each group of key-value pairs with the same key can be processed in isolation.

The *formal definition* of Reduce is

$$(k_2, [v_2]) \rightarrow [(k_3, v_3)]$$

Notice that

- The Reduce function receives a list of values $[v_2]$ associated with a specific key k_2; so the Reduce function is invoked M times, where M is the number of different keys in the input list;
- The Reduce function emits a list of key-value pairs.

13.4.3 Shuffle and sort

The shuffle and sort phase is always the same: it works by grouping the output of the Map phase by key. It does not need to be defined by developers, and it is already provided by the Hadoop system.

13.5 Data structures

Key-value pair is the basic data structure in MapReduce. Keys and values can be integers, float, strings, ..., in general they can also be (almost) arbitrary data structures defined by the designer. Notice that both input and output of a MapReduce program are lists of key-value pairs.

All in all, the design of MapReduce involves imposing the key-value structure on the input and output data sets. For example, in a collection of Web pages, input keys may be URLs and values may be their HTML content.

In many applications, the key part of the input data set is ignored. In other words, the Map function usually does not consider the key of its key-value pair argument (e.g., word count problem). Some specific applications exploit also the keys of the input data (e.g., keys can be used to uniquely identify records/objects).

13.6 Pseudocode of word count solution using MapReduce

Map

```
def map(key, value):
    '''
    :key: offset of the word in the file
    :value: a word of the input document
    '''
    return (value, 1)
```

Reduce

```
def reduce(key, values):
    '''
    :key: a word
    :values: a list of integers
    '''
    occurrences = 0
    for c in values:
        occurrences = occurrences + c
    return (key, occurrences)
```

14 How to write MapReduce programs in Hadoop

Designers and developers focus on the definition of the Map and Reduce functions (i.e., m and r), and they don't need to manage the distributed execution of the map, shuffle and sort, and reduce phases. Indeed, the Hadoop framework coordinates the execution of the MapReduce program, managing:

- the parallel execution of the map and reduce phases
- the execution of the shuffle and sort phase
- the scheduling of the subtasks
- the synchronization

The programming language to use to give instructions to Hadoop is Java. A Hadoop MapReduce program consists of three main parts:

- Driver
- Mapper
- Reducer

Each part is "implemented" by means of a specific class.

Terminology

Term	Definition
Driver class	The class containing the method/code that coordinates the configuration of the job and the "workflow" of the application
Mapper class	A class "implementing" the map function
Reducer class	A class "implementing" the reduce function
Driver	Instance of the Driver class (i.e., an object)
Mapper	Instance of the Mapper class (i.e., an object)
Reducer	Instance of the Reducer class (i.e., an object)
(Hadoop) Job	Execution/run of a MapReduce code over a data set
Task	Execution/run of a Mapper (Map task) or a Reducer (Reduce task) on a slice of data. Notice that there may be many tasks for each job

Input split	Fixed-size piece of the input data. Usually each split has approximately the same size of a HDFS block/chunk

14.1 Driver (instance)

The Driver is characterized by the `main()` method, which accepts arguments from the command line (i.e., it is the entry point of the application). Also, it has a run() method

- It configures the job
- It submits the job to the Hadoop Cluster
- It "coordinates" the work flow of the application
- It runs on the client machine (i.e., it does not run on the cluster)

14.2 Mapper (instance)

The Mapper is an instance of the Mapper class.

- It "implements" the map phase;
- It is characterized by the `map()` method, which processes the `(key, value)` pairs of the input file and emits `(key, value)` pairs and is invoked one time for each input `(key, value)` pair;
- It runs on the cluster.

Tip

The Driver will try to create one Mapper instance for each input block, pushing to the maximum parallelization possible.

14.3 Reducer (instance)

The Reducer is an instance of the Reduce class.

- It "implements" the reduce phase;
- It is characterized by the `reduce()` method, which processes `(key, [list of values])` pairs and emits `(key, value)` pairs and is invoked one time for each distinct key;
- It runs on the cluster.

15 Hadoop implementation of the MapReduce phases

The main characteristics Hadoop implementation of the MapReduce are the following

- The input `key-value` pairs are read from the HDFS file system.
- The map method of the Mapper is invoked over each input `key-value` pair, and emits a set of intermediate `key-value` pairs that are stored in the local file system of the computing server (they are not stored in HDFS).
- The intermediate results are aggregated by means of a shuffle and sort procedure, and a set of `(key, [list of values])` pairs is generated. Notice that one `(key, [list of values])` for each distinct key.
- The reduce method of the Reducer is applied over each `(key, [list of values])` pair, and emits a set of `key-value` pairs that are stored in HDFS (the final result of the MapReduce application).
- Intermediate `key-value` pairs are transient, which means that they are not stored on the distributed files system, while they are stored locally to the node producing or processing them.
- In order to parallelize the work/the job, Hadoop executes a set of tasks in parallel
  - It instantiates one Mapper (Task) for each input split
  - It instantiates a user-specified number of Reducers: each reducer is associated with a set of keys, and it receives and processes all the key-value pairs associated with its set of keys
- Mappers and Reducers are executed on the nodes/servers of the clusters

15.1 Driver class

The Driver class extends the `org.apache.hadoop.conf.Configured` class and implements the `org.apache.hadoop.util.Tool` interface [1].

[1] An **interface** is like a template of a class, defining which methods must be implemented to be compliant with the interface

Figure 15.1: MapReduce data flow with a single reducer

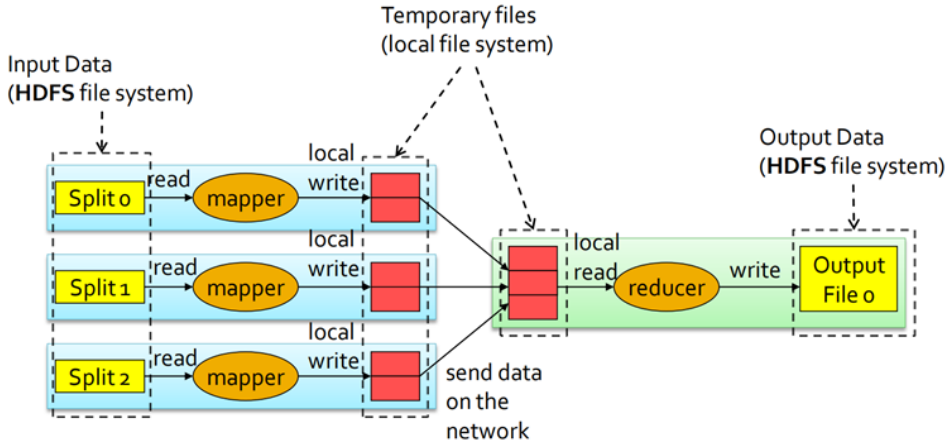
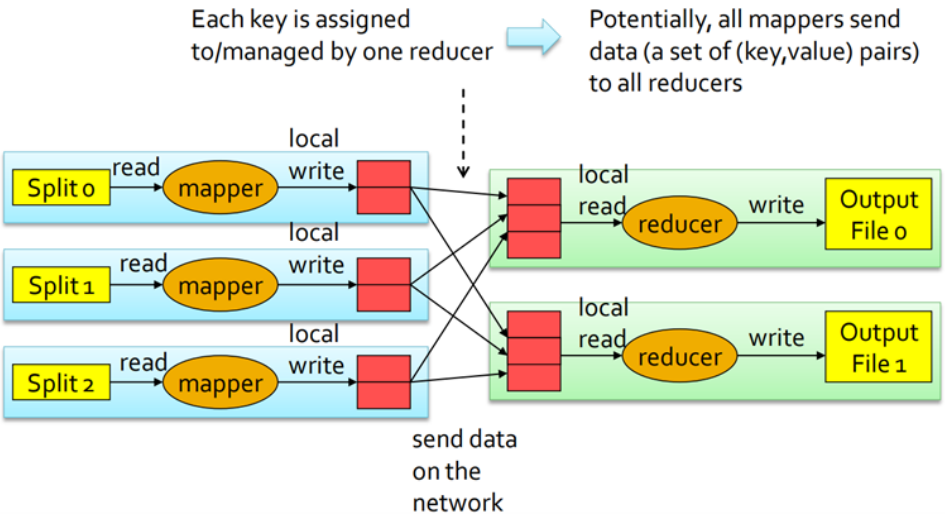


Figure 15.2: MapReduce data flow with multiple reducers



It is possible to write a Driver class that does not extend Configured and does not implement Tool, however some low level details related to some command line parameters must be managed in that case.

The designer/developer implements the `main()` and `run()` methods.

The `run()` method configures the job, defining

- The name of the Job
- The job Input format
- The job Output format
- The Mapper class
  - Name of the class
  - Type of its input `(key, value)` pairs
  - Type of its output `(key, value)` pairs
- The Reducer class
  - Name of the class
  - Type of its input `(key, value)` pairs
  - Type of its output `(key, value)` pairs
- The Number of Reducers[2]

15.2 Mapper class

The Mapper class extends the

```
1 org.apache.hadoop.mapreduce.Mapper
```

class which is a generic type/generic class with four type parameters:

- input key type
- input value type
- output key type
- output value type

The designer/developer implements the `map()` method, that is automatically called by the framework for each `(key, value)` pair of the input file.

The `map()` method

- Processes its input `(key, value)` pairs by using standard Java code

[2]Setting the number of Reducers is a balancing problem: having more Reducers decreases the time to aggregate the data, however it also increases the overhead needed to instantiate the Reducers

- Emits `(key, value)` pairs by using the `context.write(key, value)` method

15.3 Reducer class

The Reducer class extends the

```
org.apache.hadoop.mapreduce.Reducer
```

class, which is a generic type/generic class with four type parameters:

- input key type
- input value type
- output key type
- output value type

The designer/developer implements the `reduce()` method, that is automatically called by the framework for each `(key, [list of values])` pair obtained by aggregating the output of the mapper(s).

The `reduce()` method

- Processes its input `(key, [list of values])` pairs by using standard Java code
- Emits `(key, value)` pairs by using the `context.write(key, value)` method

15.4 Data Types

Hadoop has its own basic data types optimized for network serialization

- `org.apache.hadoop.io.Text`: like Java String
- `org.apache.hadoop.io.IntWritable`: like Java Integer
- `org.apache.hadoop.io.LongWritable`: like Java Long
- `org.apache.hadoop.io.FloatWritable`: like Java Float
- ...

The basic Hadoop data types implement the `org.apache.hadoop.io.Writable` and `org.apache.hadoop.io.WritableComparable` interfaces

- All classes (data types) used to represent **keys** are instances of `WritableComparable`: keys must be "comparable" for supporting the sort and shuffle phase
- All classes (data types) used to represent **values** are instances of `Writable`: usually, they are also instances of `WritableComparable` even if it is not indispensable

Developers can define new data types by implementing the `org.apache.hadoop.io.Writable` and/or `org.apache.hadoop.io.WritableComparable` interfaces, allowing to manage complex data types.

15.5 Input: `InputFormat`

The input of the MapReduce program is an HDFS file (or an HDFS folder), but the input of the Mapper is a set of `(key, value)` pairs.

The classes extending the `org.apache.hadoop.mapreduce.InputFormat` abstract class are used to read the input data and "logically transform" the input HDFS file in a set of `(key, value)` pairs.

`InputFormat` describes the input-format specification for a MapReduce application and processes the input file(s). The `InputFormat` class is used to

- Read input data and validate the compliance of the input file with the expected input-format
- Split the input file(s) into logical Input Splits, each of which is then assigned to an individual Mapper
- Provide the `RecordReader` implementation to be used to divide the logical input split in a set of `(key,value)` pairs (also called records) for the mapper

`InputFormat` identifies partitions of the data that form an input split

- Each input split is a (reference to a) part of the input file processed by a single mapper
- Each split is divided into records, and the mapper processes one record (i.e., a `(key,value)` pair) at a time

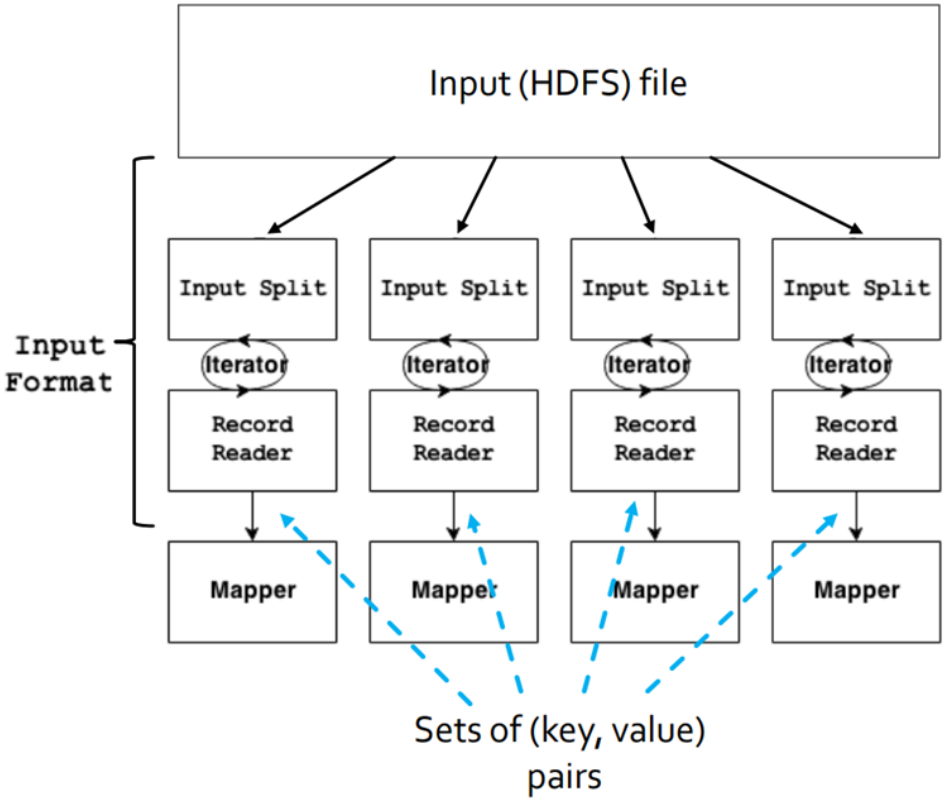
A set of predefined classes extending the `InputFormat` abstract class are available for standard input file formats

- `TextInputFormat`: `InputFormat` for plain text files
- `KeyValueTextInputFormat`: another `InputFormat` for plain text files
- `SequenceFileInputFormat`: an `InputFormat` for sequential/binary files
- …

15.5.1 `TextInputFormat`

`TextInputFormat` is an `InputFormat` for plain text files. Files are broken into lines, where either linefeed or carriage-return are used to signal end of line. One pair `(key, value)` is emitted for each line of the file:

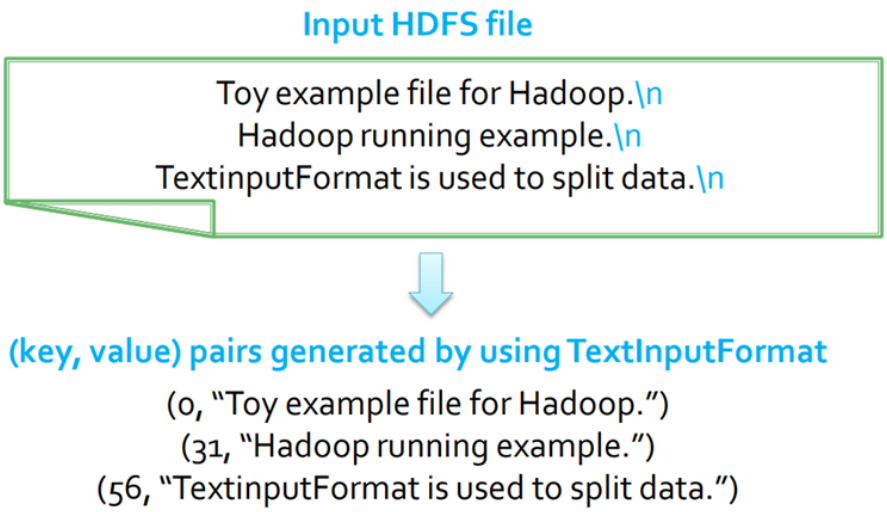
Figure 15.3: Getting data to the Mapper



- Key is the position (offset) of the line in the file
- Value is the content of the line

Example

Figure 15.4: Getting data to the Mapper



15.5.2 `KeyValueTextInputFormat`

`KeyValueTextInputFormat` is an `InputFormat` for plain text files, where each line must have the format

```
key<separator>value
```

and the default separator is tab (`\t`).

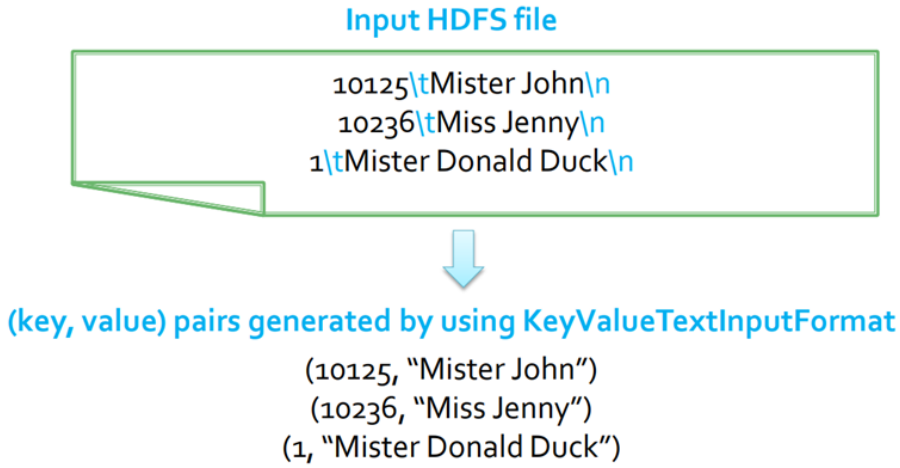
Files are broken into lines, and either linefeed or carriage-return are used to signal end of line, and each line is split into key and value parts by considering the separator symbol/character.

One pair `(key, value)` is emitted for each line of the file

- Key is the text preceding the separator
- Value is the text following the separator

Example

Figure 15.5: Getting data to the Mapper



15.6 Output: `OutputFormat`

The classes extending the `org.apache.hadoop.mapreduce.OutputFormat` abstract class are used to write the output of the MapReduce program in HDFS.

A set of predefined classes extending the `OutputFormat` abstract class are available for standard output file formats

- `TextOutputFormat`: an `OutputFormatfor` plain text files
- `SequenceFileOutputFormat`: an `OutputFormatfor` sequential/binary files
- ...

15.6.1 `TextOutputFormat`

`TextOutputFormat` is an `OutputFormat` for plain text files: for each output `(key, value)` pair, `TextOutputFormat` writes one line in the output file. In particular, the format of each output line is

```
1  "key\tvalue\n"
```

16 Structure of a MapReduce program in Hadoop

Always start from these templates. The parts of the code that should be changed to customize the Hadoop application are highlighted using notes.

16.1 Driver

```
/* Set package */ // # <1>
package it.polito.bigdata.hadoop.mypackage;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

/* Driver class */
public class MapReduceAppDriver extends Configured implements Tool { // # <2>
    @Override
    public int run(String[] args) throws Exception {
        /* variables */
        int exitCode;
        //...

        // Parse parameters
        numberOfReducers = Integer.parseInt(args[0]); // Number of instances of the Reduce
        inputPath = new Path(args[1]); // Can be the path to a folder or to a file. If thi
```

```
27          outputDir = new Path(args[2]); // This is always the path to a folder
28
29          // Define and configure a new job
30          Configuration conf = this.getConf(); // Create a configuration object to design in
31          Job job = Job.getInstance(conf); // Creation of the job, that is the application i
32
33          // Assign a name to the job
34          job.setJobName("My First MapReduce program"); // # <3>
35
36          // Set path of the input file/folder (if it is a folder, the job reads all the fil
37          FileInputFormat.addInputPath(job, inputPath);
38
39          // Set path of the output folder for this job
40          FileOutputFormat.setOutputPath(job, outputDir);
41
42          // Set input format
43          // TextInputFormat = textual files; the input types are (keys: LongWritable, value
44          // KeyValueTextInputFormat = textual files; the input types are (keys: text, value
45          job.setInputFormatClass(TextInputFormat.class); // This class also includes the in
46
47          // Set job output format
48          job.setOutputFormatClass(TextOutputFormat.class); // # <5>
49
50          // Specify the class of the Driver for this job
51          job.setJarByClass(MapReduceAppDriver.class); // # <6>
52
53          // Set mapper class
54          job.setMapperClass(MyMapperClass.class); // # <7>
55
56          // Set map output key and value classes; these are also the key - value types of t
57          job.setMapOutputKeyClass(output key type.class); // where type changes depending o
58          job.setMapOutputValueClass(output value type.class); // # <9>
59
60          // Set reduce class
61          job.setReducerClass(MyReducerClass.class); // # <10>
62
63          // Set reduce output key and value classes
64          job.setOutputKeyClass(output key type.class); // # <11>
65          job.setOutputValueClass(output value type.class); // # <12>
66
67          // Set number of reducers
```

```
68          job.setNumReduceTasks(numberOfReducers);
69
70          // Execute the job and wait for completion
71          if (job.waitForCompletion(true)==true) // with this method the application is run
72              exitCode=0;
73          else
74              exitCode=1;
75          return exitCode;
76      } // End of the run method
77
78      /* main method of the driver class */
79      public static void main(String args[]) throws Exception { // This part of the code is
80          /* Exploit the ToolRunner class to "configure" and run the Hadoop application */
81          int res = ToolRunner.run(
82              new Configuration(),
83              new MapReduceAppDriver(), // # <13>
84              args
85          );
86          System.exit(res);
87      } // End of the main method
88  } // End of public class MapReduceAppDriver
```

1. mypackage
2. MapReduceAppDriver
3. "My First MapReduce program"
4. TextInputFormat
5. TextInputFormat
6. MapReduceAppDriver
7. MyMapperClass
8. output value type
9. output value type
10. MyReducerClass
11. output value type
12. output value type
13. MapReduceAppDriver

16.2 Mapper

```java
/* Set package */
package it.polito.bigdata.hadoop.mypackage; // # <1>

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.io.*;

/* Mapper Class */
class myMapperClass extends Mapper< // Mapper is a template // # <2>
    MapperInputKeyType, // Input key type (must be consistent with the InputFormat class s
    MapperInputValueType, // Input value type (must be consistent with the InputFormat cla
    MapperOutputKeyType, // Output key type // # <5>
    MapperOutputValueType // Output value type // # <6>
>{
/* Implementation of the map method */
    protected void map(
        MapperInputKeyType key, // Input key // # <7>
        MapperInputValueType value, // Input value // # <8>
        Context context // This is an object containing the write method, that has to be i
    ) throws IOException, InterruptedException {

        /*
        Process the input (key, value) pair and emit a set of (key,value) pairs.
        context.write(...) is used to emit (key, value) pairs context.write(new outputkey,
        */

        context.write(new outputkey, new outputvalue); // # <9>
        // Notice context.write(...) has to be invoked a number of times equal to the numb

        // In the mapper instance also setup and cleanup methods can be implemented, but a

    } // End of the map method
} // End of class myMapperClass
```

1. mypackage
2. myMapperClass
3. MapperInputKeyType
4. MapperInputValueType

5. `MapperOutputKeyType`
6. `MapperOutputValueType`
7. `MapperInputKeyType`
8. `MapperInputValueType`
9. `outputkey` and `outputvalue`

16.3 Reducer

```
/* Set package */
package it.polito.bigdata.hadoop.mypackage; // # <1>

/* Import libraries */
import java.io.IOException;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.io.*;

/* Reducer Class */
class myReducerClass extends Reducer< // Reducer is a template // # <2>
    ReducerInputKeyType, // Input key type (must be consistent with the OutputKeyType of t
    ReducerInputValueType, // Input value type (must be consistent with the OutputValueTyp
    ReducerOutputKeyType, // Output key type (must be consistent with the OutputFormat cla
    ReducerOutputValueType // Output value type (must be consistent with the OutputFormat
>{
    /* Implementation of the reduce method */
    protected void reduce(
        ReducerInputKeyType key, // Input key // # <7>
        Iterable<ReducerInputValueType> values, // Input values (list of values). Notice t
        Context context
    ) throws IOException, InterruptedException {

        /*
        Process the input (key, [list of values]) pair and emit a set of (key,value) pairs
        context.write(...) is used to emit (key, value) pairs context.write(new outputkey,
        */

        context.write(new outputkey, new outputvalue); // # <9>
        // Notice context.write(...) has to be invoked a number of times equal to the numb
        // "new" has to be always specified

    } // End of the reduce method
```

```
33 } // End of class myReducerClass
```

1. `mypackage`
2. `myReducerClass`
3. `ReducerInputKeyType`
4. `ReducerInputValueType`
5. `ReducerOutputKeyType`
6. `ReducerOutputValueType`
7. `ReducerInputKeyType`
8. `ReducerInputValueType`
9. `outputkey` and `outputvalue`

16.4 Example of a MapReduce program in Hadoop: Word Count

The Word count problem consists of

- Input: (unstructured) textual file, where each line of the input file can contains a set of words
- Output: number of occurrences of each word appearing in the input file
- Parameters/arguments of the application:
  - `args[0]`: number of instances of the reducer
  - `args[1]`: path of the input file
  - `args[2]`: path of the output folder

Word Count input and output examples

Input file

```
Toy example file for Hadoop. Hadoop running example.
```

Output file

```
(toy,1)
(example,2)
(file,1)
(for,1)
(hadoop,2)
(running,1)
```

16.4.1 Driver

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

/* Driver class */
public class WordCount extends Configured implements Tool {
    @Override
    public intrun(String[] args) throws Exception {
        Path inputPath;
        Path outputDir;
        int numberOfReducers;
        int exitCode;

        // Parse input parameters
        numberOfReducers = Integer.parseInt(args[0]);
        inputPath = new Path(args[1]);
        outputDir = new Path(args[2]);

        // Define and configure a new job
        Configuration conf = this.getConf();
        Job job = Job.getInstance(conf);

        // Assign a name to the job
        job.setJobName("WordCounter");

        // Set path of the input file/folder (if it is a folder, the job reads all the fil
```

```
40          FileInputFormat.addInputPath(job, inputPath);
41
42          // Set path of the output folder for this job
43          FileOutputFormat.setOutputPath(job, outputDir);
44
45          // Set input format
46          // TextInputFormat = textual files
47          job.setInputFormatClass(TextInputFormat.class);
48
49          // Set job output format
50          job.setOutputFormatClass(TextOutputFormat.class);
51
52          // Specify the class of the Driver for this job
53          job.setJarByClass(WordCount.class);
54
55          // Set mapper class
56          job.setMapperClass(WordCountMapper.class);
57
58          // Set map output key and value classes
59          job.setMapOutputKeyClass(Text.class);
60          job.setMapOutputValueClass(IntWritable.class);
61
62          // Set reduce class
63          job.setReducerClass(WordCountReducer.class);
64
65          // Set reduce output key and value classes
66          job.setOutputKeyClass(Text.class);
67          job.setOutputValueClass(IntWritable.class);
68
69          // Set number of reducers
70          job.setNumReduceTasks(numberOfReducers);
71
72          // Execute the job and wait for completion
73          if (job.waitForCompletion(true)==true)
74              exitCode=0;
75          else
76              exitCode=1;
77          return exitCode;
78      } // End of the run method
79
80      /* main method of the driver class */
```

```
    public static void main(String args[]) throws Exception {

        /* Exploit the ToolRunner class to "configure" and run the
        Hadoop application */

        intres = ToolRunner.run(
            new Configuration(),
            new WordCount(),
            args
        );
        System.exit(res);
    } // End of the main method
} // End of public class WordCount
```

16.4.2 Mapper

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

/* MapperClass */
class WordCountMapper extends Mapper<
    LongWritable, // Input key type
    Text, // Input value type
    Text, // Output key type
    IntWritable // Output value type
>{
    /* Implementation of the map method */
    protected void map(
        LongWritable key, // Input key type
        Text value, // Input value type
        Context context
    ) throws IOException, InterruptedException {
        // Split each sentence in words. Use whitespace(s) as delimiter
```

```
        // The split method returns an array of strings
        String[] words = value.toString().split("\\s+");

        // Iterate over the set of words
        for(String word : words) {
            // Transform word case
            String cleanedWord = word.toLowerCase();

            // emit one pair (word, 1) for each input word
            context.write(new Text(cleanedWord), new IntWritable(1));
        }
    } // End map method
} // End of class WordCountMapper
```

16.4.3 Reducer

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

/* Reducer Class */
class WordCountReducer extends Reducer<
Text, // Input key type
IntWritable, // Input value type
Text, // Output key type
IntWritable // Output value type
>{
    /* Implementation of the reduce method */
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context
    ) throws IOException, InterruptedException{
        int occurrances= 0;

```

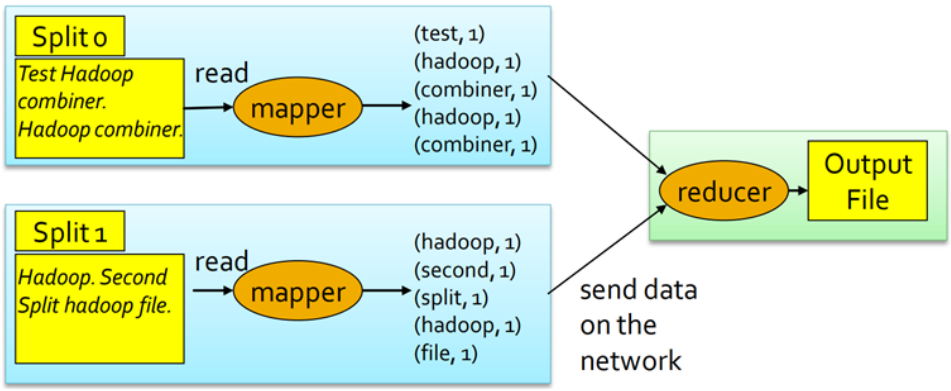
```
25          // Iterate over the set of values and sum them
26          for (IntWritable value : values) {
27              occurrances = occurrances+ value.get();
28          }
29
30          // Emit the total number of occurrences of the current word
31          context.write(key, new IntWritable(occurrances));
32      } // End reduce method
33  } // End of class WordCountReducer
```

17 Combiner

In standard MapReduce applications, the `(key,value)` pairs emitted by the Mappers are sent to the Reducers through the network. However, some pre-aggregations could be performed to limit the amount of network data by using Combiners (also called "minireducers").

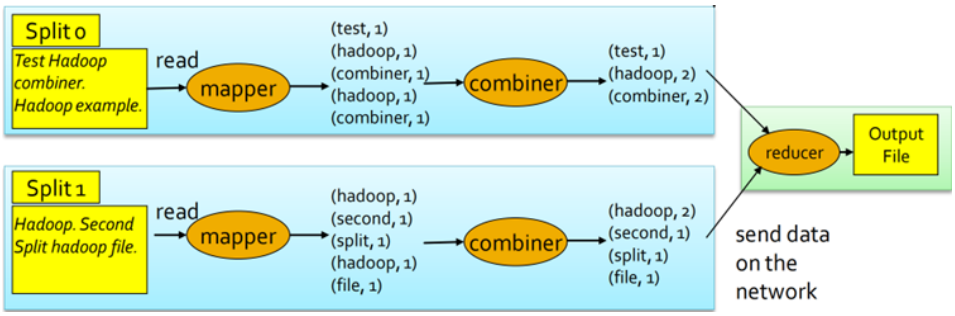
Consider the standard word count problem, and suppose that the input file is split in two input splits, hence, two Mappers are instantiated (one for each split).

Figure 17.1: Word count without Combiner



A combiner can be locally called on the output `(key, value)` pairs of each mapper (it works on data stored in the main-memory or on the local hard disks) to pre-aggregate data, reducing the data moving through the network.

Figure 17.2: Word count with Combiner



So, in MapReduce applications that include Combiners after the Mappers, the `(key,value)` pairs emitted by the Mappers are analyzed in main-memory (or on the local disk) and aggregated by the Combiners. Each Combiner pre-aggregates the values associated with the pairs emitted by the Mappers of a cluster node, limiting the amount of network data generated by each cluster node.

> ⚠ Combiner scope of application
>
> - Combiners work only if the reduce function is **commutative** and **associative**.
> - The execution of combiners is not guaranteed: Hadoop decides at runtime if executing a combiner, and so the user cannot be sure of the combiner execution just by checking the code. Because of this, the developer/designer should write MapReduce jobs whose successful executions **do not depend** on whether the Combiner is executed.

17.1 Combiner (instance)

The Combiner is an instance of the `org.apache.hadoop.mapreduce.Reducer` class. Notice that there is not a specific combiner-template class.

- It "implements" a pre-reduce phase that aggregates the pairs emitted in each node by Mappers
- It is characterized by the `reduce()` method
- It processes `(key, [list of values])` pairs and emits `(key, value)` pairs
- It runs on the cluster

17.2 Combiner class

The Combiner class extends the `org.apache.hadoop.mapreduce.Reducer` class, that is a generic type/generic class with four type parameters:

- input key type
- input value type
- output key type
- output value type

Combiners and Reducers extend the **same class**, and the designer/developer implements the `reduce()` method also for the Combiner instances. The Combiner is automatically called by Hadoop for each `(key, [list of values])` pair obtained by aggregating the local output of a Mapper.

The Combiner class is specified by using the `job.setCombinerClass()` method in the run method of the Driver (i.e., in the job configuration part of the code).

17.3 Example: adding the Combiner to the Word Count problem

Consider the word count problem (see Section 16.4 for details), to add the combiner to solution seen before:

- Specify the combiner class in the Driver
- Define the Combiner class. The reduce method of the combiner aggregates local pairs emitted by the mappers of a single cluster node, and emits partial results (local number of occurrences for each word) from each cluster node that is used to run our application.

17.3.1 Specify combiner class in the Driver

Add the call to the combiner class in the Driver, before the return around line 68

```
1          // Set combiner class
2          job.setCombinerClass(WordCountCombiner.class);
```

17.3.2 Define the Combiner class

```
1  /* Set package */
2  package it.polito.bigdata.hadoop.wordcount;
3
4  /* Import libraries */
5  import java.io.IOException;
6  import org.apache.hadoop.io.IntWritable;
7  import org.apache.hadoop.io.Text;
8  import org.apache.hadoop.mapreduce.Reducer;
9
10 /* Combiner Class */
11 class WordCountCombiner extends Reducer<
12     Text, // Input key type
13     IntWritable, // Input value type
14     Text, // Output key type
15     IntWritable // Output value type
16 >{
17 /* Implementation of the reduce method */
```

```
18      protected void reduce(
19          Text key, // Input key type
20          Iterable<IntWritable> values, // Input value type
21          Context context
22      ) throws IOException, InterruptedException{
23          int occurrances= 0;
24          // Iterate over the set of values and sum them
25          for (IntWritable value : values) {
26              occurrances = occurrances+ value.get();
27          }
28          // Emit the total number of occurrences of the current word
29          context.write(key, new IntWritable(occurrances));
30      } // End reduce method
31  } // End of class WordCountCombiner
```

17.4 Final thoughts

The reducer and the combiner classes perform the same computation (the reduce method of the two classes is the same). Indeed, the developer/designer does not really need two different classes: he can simply specify that WordCountReducer is also the combiner class, for example by adding in the driver `job.setCombinerClass(WordCountReducer.class)`. In 99% of the Hadoop applications the same class can be used to implement both combiner and reducer.

18 Personalized Data Types

Personalized Data Types are useful when the **value** of a key-value pair is a **complex data type**. Personalized Data Types are defined by implementing the `org.apache.hadoop.io.Writable` interface. The following methods must be implemented

- `public void readFields(DataInput in)`
- `public void write(DataOutput out)`

To properly format the output of the job usually also the following method is "redefined"

- `public String toString()`

Suppose to be interested in complex values composed of two parts, such as a counter (int) and a sum (float). In this case, an ad-hoc Data Type can be used to implement this complex data type in Hadoop.

18.1 Example

```
1  /* Set package */
2
3  package it.polito.bigdata.hadoop.combinerexample;
4  import java.io.DataInput;
5  import java.io.DataOutput;
6  import java.io.IOException;
7  public class SumAndCountWritable implements
8  org.apache.hadoop.io.Writable {
9      /* Private variables */
10     private float sum = 0;
11     private int count = 0;
12
13     /* Methods to get and set private variables of the class */
14     public float getSum() {
15         return sum;
16     }
17
```

```
18      public void setSum(float sumValue) {
19          sum=sumValue;
20      }
21
22      public int getCount() {
23          return count;
24      }
25
26      public void setCount(int countValue) {
27          count=countValue;
28      }
29
30      /* Methods to serialize and deserialize the contents of the
31      instances of this class */
32      @Override /* Serialize the fields of this object to out */
33      public void write(DataOutput out) throws IOException {
34          out.writeFloat(sum);
35          out.writeInt(count);
36      }
37
38      @Override /* Deserialize the fields of this object from in */
39      public void readFields(DataInput in) throws IOException {
40          sum=in.readFloat();
41          count=in.readInt();
42      }
43
44      /* Specify how to convert the contents of the instances of this
45      class to a String
46      * Useful to specify how to store/write the content of this class
47      * in a textual file */
48      public String toString()
49      {
50          String formattedString=
51          new String("sum="+sum+",count="+count);
52          return formattedString;
53      }
54  }
```

18.2 Complex keys

Personalized Data Types can be used also to manage complex **keys**. In that case the Personalized Data Type must implement the `org.apache.hadoop.io.WritableComparable` interface, since keys must be

- compared/sorted: it is possible by implementing the `compareTo()` method
- split in groups: it is possible by implementing the `hashCode()` method

19 Sharing parameters among Driver, Mappers, and Reducers

The configuration object is used to share the (basic) configuration of the Hadoop environment across the driver, the mappers and the reducers of the application/job. It stores a list of `(property-name, property-value)` pairs.

Also, personalized `(property-name, property-value)` pairs can be specified in the driver, and they can be used to share some parameters of the application with mappers and reducers. The personalized `(property-name, property-value)` pairs are useful to define shared small (constant) properties that are available only during the execution of the program. The driver sets these parameters, and Mappers and Reducers can access them, however they cannot modify them.

19.1 How to use these parameters

In the driver

1. Retrieve the configuration object

```
Configuration conf = this.getConf();
```

2. Set personalized properties

```
conf.set("property-name", "value");
```

In the Mapper and/or Reducer

```
context.getConfiguration().get("property-name")
```

This method returns a String containing the value of the specified property.

20 Counters

Hadoop provides a set of basic, built-in, counters to store some statistics about jobs, mappers, reducers, for example

- number of input and output records (i.e., pairs)
- number of transmitted bytes

Also other ad-hoc, user-defined, counters can be defined to compute global "statistics" associated with the goal of the application.

20.1 User-defined counters

User-defined counters are defined by means of Java enum, and each application can define an arbitrary number of enums. The name of the enum is the group name,and each enum has a number of "fields", which are the counter names.

Counters are incremented in the Mappers and Reducers by using the `increment()` method

```
context.getCounter(countername).increment(value);
```

The global/final value of each counter, which is available at the end of the job, is then stored/printed by the Driver (at the end of the execution of the job). Driver can retrieve the final values of the counters using the `getCounters()` and `findCounter()` methods.

User-defined counters can be also defined on the fly by using the method `incrCounter("group name", "counter name", value)`. Dynamic counters are useful when the set of counters is unknown at design time.

20.2 Example: use the counters

In the driver, add

```
public static enum COUNTERS {
    ERROR_COUNT,
```

```
    MISSING_FIELDS_RECORD_COUNT
}
```

This enum defines two counters

- `COUNTERS.ERROR_COUNT`
- `COUNTERS.MISSING_FIELDS_RECORD_COUNT`

To increment the `COUNTERS.ERROR_COUNT` counter in the mapper or the reducer, use

```
context.getCounter(COUNTERS.ERROR_COUNT).increment(1);
```

To retrieve the final value of the `COUNTERS.ERROR_COUNT` counter in the driver, use

```
Counter errorCounter = job.getCounters().findCounter(COUNTERS.ERROR_COUNT);
```

21 Map-only job

In some applications, all the work can be performed by the mapper(s) (e.g., record filtering applications): Hadoop allows executing Map-only jobs, avoiding the reduce phase, and also the shuffle and sort phase.

The output of the map job is directly stored in HDFS, since the set of pairs emitted by the map phase is already the final output.

21.1 Implementation of a Map-only job

To implement a Map-only job

- Implement the map method
- Set the number of reducers to 0 during the configuration of the job (in the driver), writing

```
1 job.setNumReduceTasks(0);
```

22 In-Mapper combiner

Mapper classes are also characterized by a setup and a cleanup method, which are empty if they are not overridden.

22.1 Setup method

The setup method is called once for each mapper prior to the many calls of the map method. It can be used to set the values of in-mapper variables, which are used to maintain in-mapper statistics and preserve the state (locally for each mapper) within and across calls to the map method.

22.2 Cleanup method

The map method, invoked many times, updates the value of the in-mapper variables. Each mapper (each instance of the mapper class) has its own copy of the in-mapper variables.

The cleanup method is called once for each mapper after the many calls to the map method, and it can be used to emit `(key,value)` pairs based on the values of the in-mapper variables/statistics.

Also the reducer classes are characterized by a setup and a cleanup method.

- The setup method is called once for each reducer prior to the many calls of the reduce method.
- The cleanup method is called once for each reducer after the many calls of the reduce method.

In-MapperCombiners are a possible improvement over "standard" Combiners

- Initialize a set of in-mapper variables during the instance of the Mapper, in the setup method of the mapper;
- Update the in-mapper variables/statistics in the map method. Usually, no (key,value) pairs are emitted in the map method of an in-mapper combiner.

After all the input records (input `(key, value)` pairs) of a mapper have been analyzed by the map method, emit the output `(key, value)` pairs of the mapper: `(key, value)` pairs are emitted in the cleanup method of the mapper based on the values of the in-mapper variables

The in-mapper variables are used to perform the work of the combiner in the mapper, allowing to improve the overall performance of the application. However, pay attention to the amount of used main memory: each mapper may use a limited amount of main-memory, hence in-mapper variables should be "small" (at least smaller than the maximum amount of memory assigned to each mapper).

22.3 In-Mapper combiner: Word count pseudocode

```
1 class MAPPER
2     method setup
3         A = new AssociativeArray
4     method map(offset key, line l)
5         for all word w  line l do
6             A{w} = A{w} + 1
7     method cleanup
8         for all word w in A do
9             EMIT(term w , count A{w})
```

23 Maven project

23.1 Structure

- *src* folder: contains the source code. May contain subfolders, but the important point is that it must contain the java files
  - DriverBigData.java
  - MapperBigData.java
  - ReducerBigData.java
- *target* folder:
  - *.jar* file: useful to run the application on the cluster. It's the java archive that collects the three classes of the Hadoop application
- *pom.xml* file: used to configure the Hadoop application

23.2 How to run the project

Using Eclipse

- select the Driver `.java` file
- Right click
- Click "Run As"
- If the arguments have already been set:
  - Click "Java Application"
- Otherwise
  - Click "Run Configurations", to set the arguments
  - Go to "Arguments" section, and write the arguments. The arguments are
    - the number of reducers: `2`
    - the (relative) path of the input folder `example_data`
    - the (relative) path of the output folder `example_data_output`

```
2 example_data example_data_output
```

The output files are

- an empty file "_SUCCESS", if the application run successfully
- one file for each reducer instance: the intersection between the sets of words in each file is empty, which means that all the same words were processed by the same Reducer. For this reason the output is always a folder and not a single file.

23.3 How to create a `.jar` file from the project

Using Eclipse, to create a `.jar` file from the project to run the project on the cluster

- Right click on the project name (e.g., "MapReduceProject")
- Click "Runs As"
- Click "Maven build..."
- In "Goals" write "package"
- Click "Run"

23.4 How to run the `.jar` in the BigData@Polito cluster

- Go to https://jupyter.polito.it/ (i.e., the server gateway) and connect using the credentials
- Copy the `.jar` file on server
- Upload the input data in the HDFS
- Use the terminal to run the `.jar`, using the `hadoop` command

```
hadoop jar Exercise1-1.0.0.jar \
it.polito.bigdata.hadoop.exercise1.DriverBigData \
2 example_data example_data_output
```

In this configuration there are 3 file systems

- The local file system on the personal PC
- The local file system on the gateway server
- The distributed file system on the Hadoop cluster (the interface to manage it is https://bigdatalab.polito.it/hue)

24 MapReduce patterns - 1

Summarization Patterns are used to implement applications that produce top-level/summarized view of the data, such as

- Numerical summarizations (Statistics)
- Inverted index
- Counting with counters

24.1 Numerical summarizations

The goal is to group records/objects by a key field(s) and calculate a numerical aggregate (e.g., average, max, min, standard deviation) per group, to provide a top-level view of large input data sets so that a few high-level statistics can be analyzed by domain experts to identify trends, anomalies, etc.

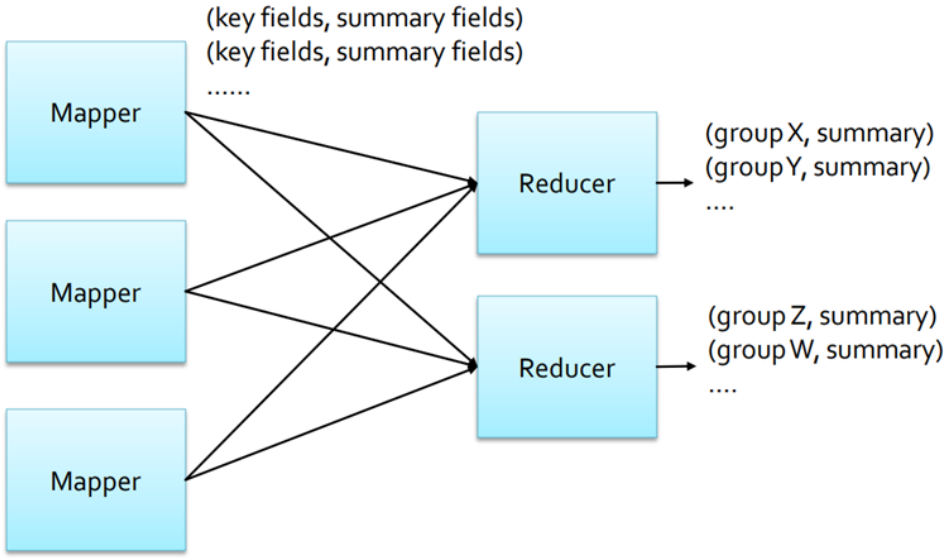
24.1.1 Structure

- Mappers output `(key, value)` pairs where
  - key is associated with the fields used to define groups;
  - value is associated with the fields used to compute the aggregate statistics.
- Reducers receive a set of numerical values for each "group-by" key and compute the final statistics for each "group". Combiners can be used to speed up performances, if the computed statistic has specific properties (e.g., it is commutative and associative).

Use cases are

- Word count
- Record count (per group)
- Min/Max/Count (per group)
- Average/Median/Standard deviation (per group)

Figure 24.1: Numerical summarization structure



24.2 Inverted index summarization

The goal is to build an index from the input data to support faster searches or data enrichment: it maps terms to a list of identifiers to improve search efficiency.

24.2.1 Structure

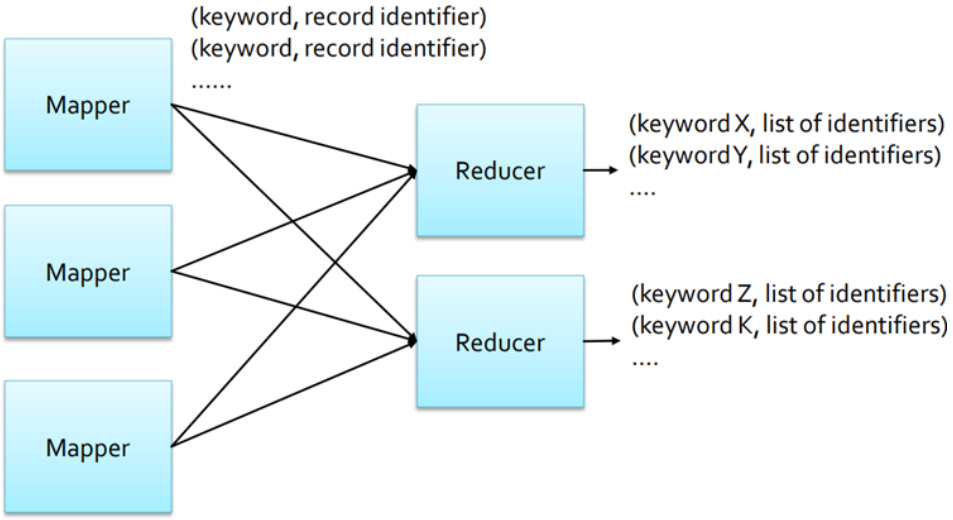
- Mappers output `(key, value)` pairs where
  - key is the set of fields to index (a keyword)
  - value is a unique identifier of the objects to associate with each "keyword"
- Reducers receive a set of identifiers for each keyword and simply concatenate them
- Combiners are usually not useful when using this pattern, since there are no values to aggregate

A use case is a web search engine (word – List of URLs, i.e. Inverted Index).

24.3 Counting with counters

The goal is to compute count summarizations of data sets to provide a top-level view of large data sets, so that few high-level statistics can be analyzed by domain experts to identify trends, anomalies, ...

Figure 24.2: Numerical summarization structure



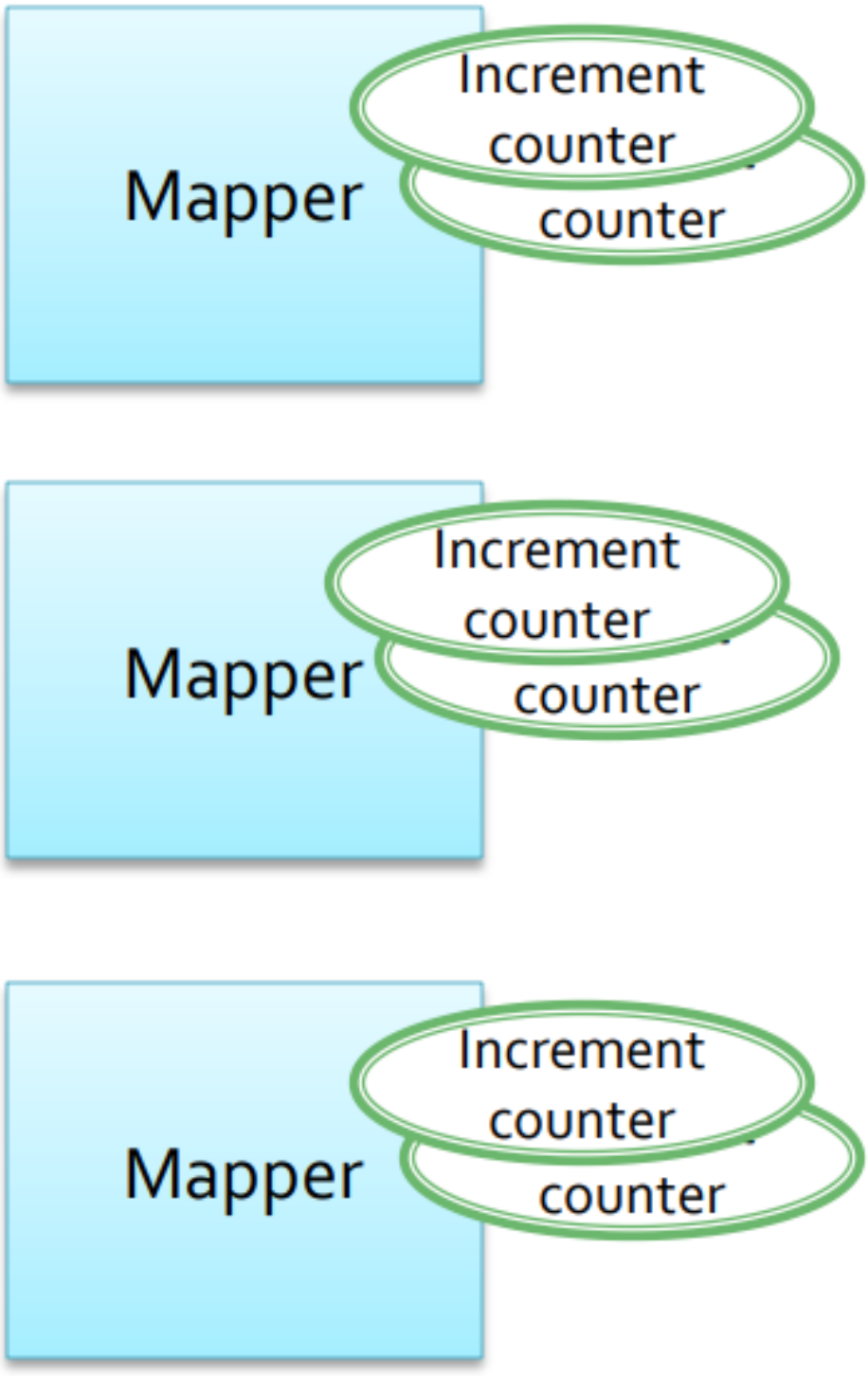
24.3.1 Structure

- Mappers process each input record and increment a set of counters
- This is a map-only job: no reducers and no combiners have to be implemented
- The results are stored/printed by the Driver of the application

Use cases

- Count number of records
- Count a small number of unique instances
- Summarizations

Figure 24.3: Numerical summarization structure



25 Filtering patterns

Are used to select the subset of input records of interest

- Filtering
- Top K
- Distinct

25.1 Filtering

The goal is to filter out input records that are not of interest/keep only the ones that are of interest, to focus the analysis of the records of interest. Indeed, depending on the goals of your application, frequently only a small subset of the input data is of interest for further analyses.

25.1.1 Structure

The input of the mapper is a set of records

- Key = primary key
- Value = record

Mappers output one `(key, value)` pair for each record that satisfies the enforced filtering rule

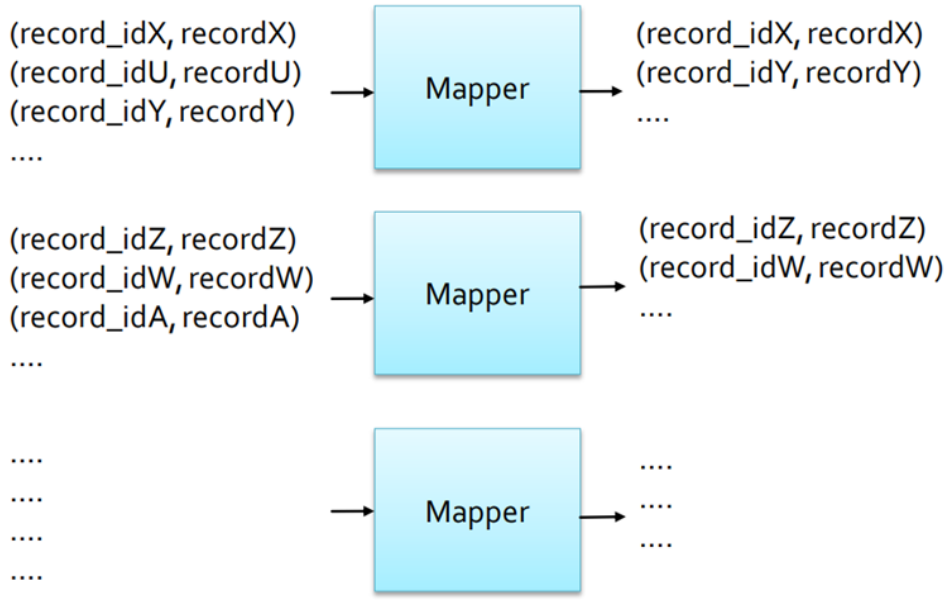
- Key is associated with the primary key of the record
- Value is associated with the selected record

Reducers are useless in this pattern, since a map-only job is executed (number of reduce set to 0).

Use cases

- Record filtering
- Tracking events
- Distributed grep
- Data cleaning

Figure 25.1: Numerical summarization structure



25.2 Top K

The goal is to select a small set of top K records according to a ranking function to focus on the most important records of the input data set: frequently the interesting records are those ranking first according to a ranking function (i.e., most profitable items, outliers).

25.2.1 Structure

25.2.1.1 Mappers

Each mapper initializes an in-mapper (local) top k list. k is usually small (e.g., 10), and the current (local) top k-records of each mapper(i.e., instance of the mapper class) can be stored in main memory

- The initialization is performed in the setup method of the mapper
- The map function updates the current in-mapper top k list

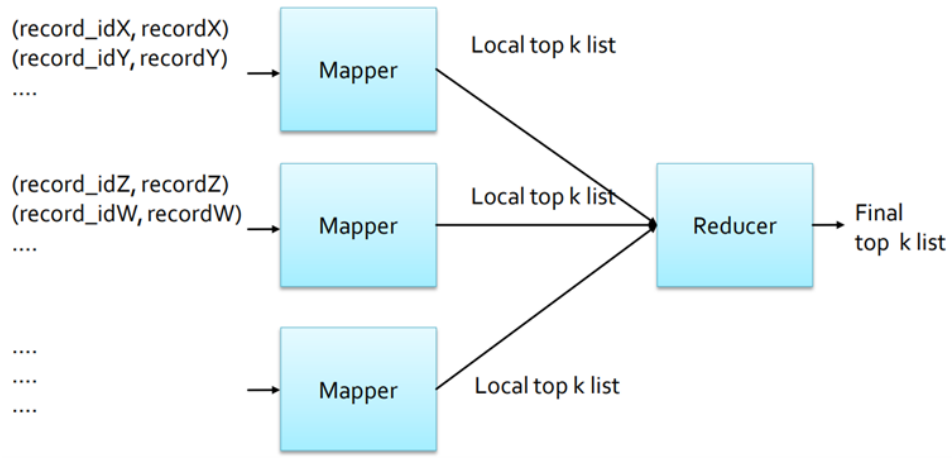
The cleanup method emits the k `(key, value)` pairs associated with the in-mapper local top k records

- Key is the "null key"
- Value is a in-mapper top k record

25.2.1.2 Reducer

A single reducer must be instantiated (i.e., one single instance of the reducer class). One single global view over the intermediate results emitted by the mappers to compute the final top k records. It computes the final top k list by merging the local lists emitted by the mappers. All input `(key, value)` pairs have the same key, hence the reduce method is called only once

Figure 25.2: Numerical summarization structure



Use cases

- Outlier analysis (based on a ranking function)
- Select interesting data (based on a ranking function)

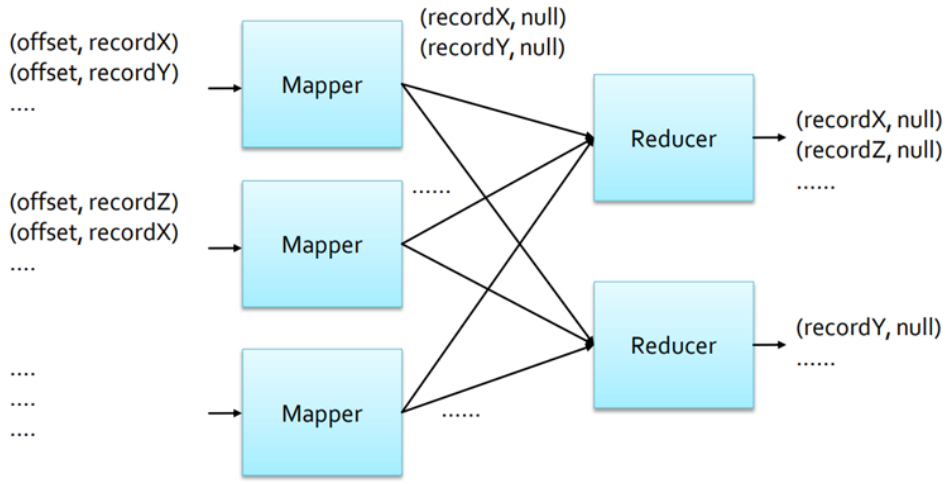
25.3 Distinct

The goal is to find a unique set of values/records, since in some applications duplicate records are useless (actually duplicated records are frequently useless).

- Mappers emit one `(key, value)` pair for each input record
  - Key = input record
  - Value = null value
- Reducers emit one `(key, value)` pair for each input `(key, list of values)` pair
  - Key = input key, (i.e., input record)
  - Value = null value

Use cases

Figure 25.3: Numerical summarization structure



- Duplicate data removal
- Distinct value selection

26 MapReduce and Hadoop Advanced Topics

In some applications data are read from two or more datasets, also having different formats.

Hadoop allows reading data from multiple inputs (multiple datasets) with different formats by specifying one mapper for each input dataset. However, the key-value pairs emitted by the mappers must be consistent in terms of data types.

> **Example of a use case**
>
> The input data is collected from different sensors: all sensors measure the same "measure", but sensors developed by different vendors use a different data format to store the gathered data/measurements.

In the driver use the `addInputPath` method of the `MultipleInputs` class multiple times to

- Add one input path at a time
- Specify the input format class for each input path
- Specify the Mapper class associated with each input path

> **Multiple inputs example**
>
> ```
> MultipleInputs.addInputPath(
>     job,
>     new Path(args[1]),
>     TextInputFormat.class,
>     Mapper1.class
> );
>
> MultipleInputs.addInputPath(
>     job,
>     new Path(args[2]),
>     TextInputFormat.class,
>     Mapper2.class
> );
> ```
>
> - Specify two input paths (`args[1]` and `args[2]`)

- The data of both paths are read by using the `TextInputFormat` class
- `Mapper1` is the class used to manage the input key-value pairs associated with the first path
- `Mapper2` is the class used to manage the input key-value pairs associated with the second path

27 Multiple outputs

In some applications it could be useful to store the output key-value pairs of a MapReduce application in different files. Each file contains a specific subset of the emitted key-value pairs, based on some rules (usually this approach is useful for splitting and filtering operations), and each file name has a prefix that is used to specify the "content" of the file.

All the files are stored in one single output directory: there aren't multiple output directories, but only multiple output files with different prefixes.

Hadoop allows specifying the prefix of the output files: the standard prefix is "part-" (see the content of the output directory of some of the previous applications).

The `MultipleOutputs` class is used to specify the prefixes of the output files

- One different prefix for each "type" of output file
- There will be one output file of each type for each reducer (for each mapperfor map-only jobs)

27.1 Driver

Use the method `MultipleOutputs.addNamedOutput` multiple times in the Driver to specify the prefixes of the output files. This method has 4 parameter

- The job object
- The "name/prefix" of MultipleOutputs
- The OutputFormat class
- The key output data type class
- The value output data type class

Call this method one time for each "output file type"

Multiple outputs example

```
MultipleOutputs.addNamedOutput(
    job,
    "hightemp",
    TextOutputFormat.class,
    Text.class,
    NullWritable.class
);

MultipleOutputs.addNamedOutput(
    job,
    "normaltemp",
    TextOutputFormat.class,
    Text.class,
    NullWritable.class
);
```

This example defines two types of output files

- The first type of output files while have the prefix `"hightemp"`
- The second type of output files while have the prefix `"normaltemp"`

27.2 Map-only

Define a private `MultipleOutputs` variable in the mapper if the job is a map-only job (in the reducer otherwise)

```
private MultipleOutputs<Text, NullWritable> mos = null;
```

Create an instance of the `MultipleOutputs` class in the setup method of the mapper (or in the reducer)

```
mos = new MultipleOutputs<Text, NullWritable>(context);
```

Use the write method of the `MultipleOutputs` object in the map method (or in the reduce method) to write the key-value pairs in the file of interest

Example

```
mos.write("hightemp", key, value);
```

This example writes the current key-value pair in a file with the prefix `"hightemp-"`

```
mos.write("normaltemp", key, value);
```

This example writes the current key-value pair in a file with the prefix `"normaltemp-"`

Close the `MultipleOutputs` object in the cleanup method of the mapper (or of the reducer)

Example

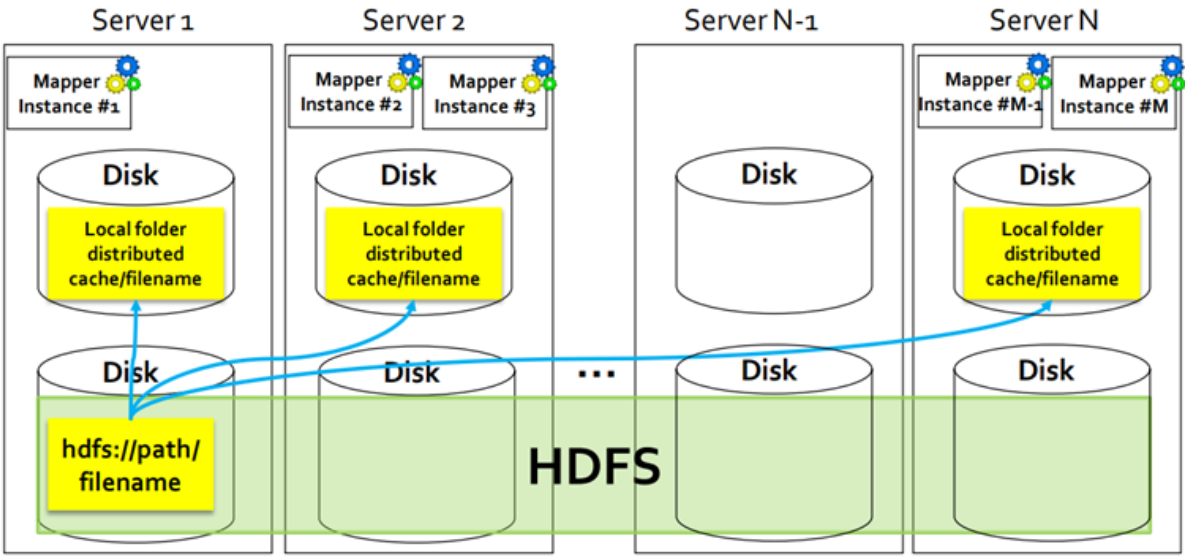
```
mos.close();
```

28 Distributed cache

Some applications need to share and cache (small) read-only files to perform efficiently their task. These files should be accessible by all nodes of the cluster in an efficient way, hence a copy of the shared/cached (HDFS) files should be available locally in all nodes used to run the application.

`DistributedCache` is a facility provided by the Hadoop-based MapReduce framework to cache files (e.g., text, archives, jars needed by applications).

Figure 28.1: Distributed cache structure



In image Figure 28.1, in HDFS disks there are the HDFS file(s) to be shared by means of the distributed cache, while on the disks there are local copies of the file(s) shared by means of the distributed cache. A local copy of the file(s) shared by means of the distributed cache is created only in the servers running the application that uses the shared file(s).

In the Driver of the application, the set of shared/cached files are specifiedby using the `job.addCacheFile(path)` method. During the initialization of the job, Hadoop creates a "local copy" of the shared/cached files in all nodes that are used to execute some tasks (mappers or reducers) of the job (i.e., of the running application). The shared/cache file is read

by the mapper (or the reducer), usually in its setup method, since the shared/cached file is available locally in the used nodes/servers, its content can be read efficiently.

The efficiency of the distributed cache depends on the number of multiple mappers (or reducers) running on the same node/server: for each node a local copy of the file is copied during the initialization of the job, and the local copy of the file is used by all mappers (reducers) running on the same node/server.

Without the distributed cache, each mapper (reducer) should read, in the setup method, the shared HDFS file, hence, more time is needed because reading data from HDFS is more inefficient than reading data from the local file system of the node running the mappers (reducers).

28.1 Example

28.1.1 Driver

```
public int run(String[] args) throws Exception {
    //...

    // Add the shared/cached HDFS file in the
    // distributed cache
    job.addCacheFile(new Path("hdfs path/filename").toUri());

    //...
}
```

28.1.2 Mapper/Reducer

```
protected void setup(Context context) throws IOException, InterruptedException{

    String line;
    // Retrieve the (original) paths of the distributed files
    URI[] urisCachedFiles = context.getCacheFiles();

    // Read the content of the cached file and process it.
    // In this example the content of the first shared file is opened.
    BufferedReaderfile = new BufferedReader(
        new FileReader(
            new File(
```

```
                    new Path(urisCachedFiles[0].getPath()).getName()
                )
            )
        );

        // Iterate over the lines of the file
        while ((line = file.readLine()) != null) {
            // process the current line
            //...
        }
        file.close();
    }
```

Notice that `.getName()` retrieves the name of the file. The shared file is stored in the root of a local temporary folder (one for each server that is used to run the application) associated with the distributed cache. The path of the original folder is different from the one used to store the local copy of the shared file.

29 MapReduce patterns - 2

Data organization patterns are used to reorganize/split in subsets the input data

- Binning
- Shuffling

The output of an application based on an organization pattern is usually the input of another application(s)

29.1 Binning

The goal is to organize/move the input records into categories, to partition a big data set into distinct, smaller data sets ("bins") containing similar records. Each partition is usually the input of a following analysis.

This is done because the input data set contains heterogonous data, but each data analysis usually is focused only on a specific subsets of the data.

29.1.1 Structure

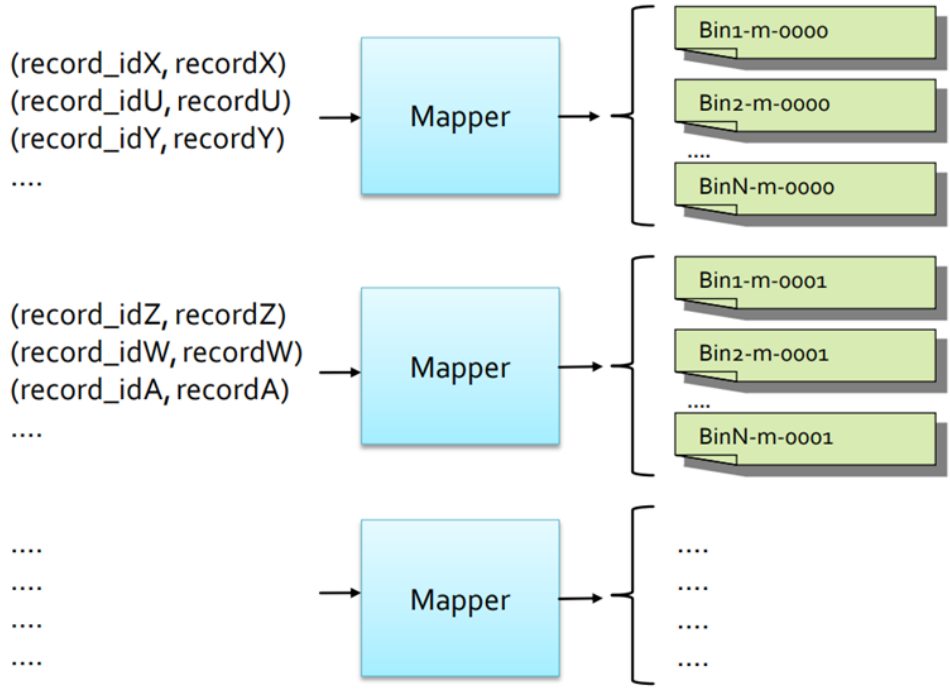
Binning is based on a Map-only job

- Driver sets the list of "bins/output files" by means of `MultipleOutputs`
- Mappers select, for each input `(key, value)` pair, the output bin/file associated with it and emit a `(key,value)` in that file
  - key of the emitted pair is key of the input pair
  - value of the emitted pair is value of the input pair
- No Combiner or Reducer is used in this pattern

29.2 Shuffling

The goal is to randomize the order of the data (records), for anonymization reasons or for selecting a subset of random data (records).

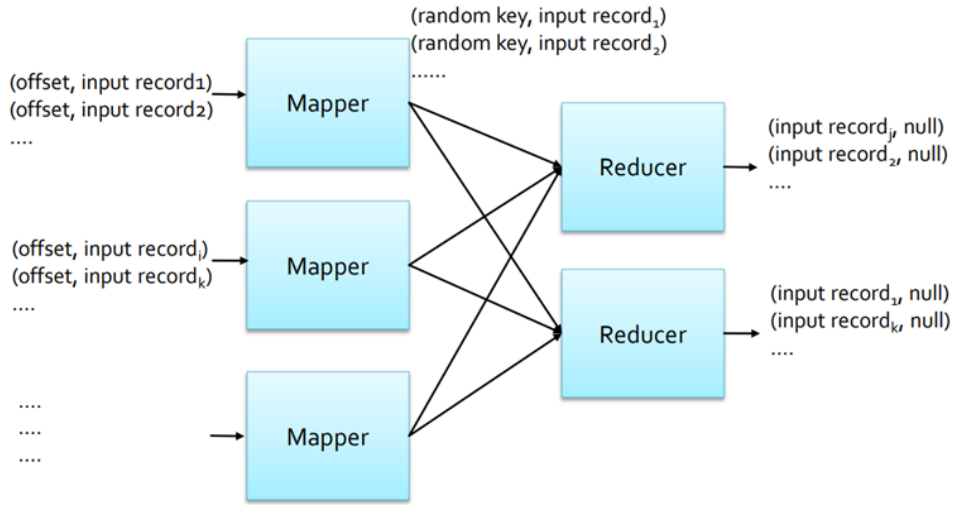
Figure 29.1: Binning structure



29.2.1 Structure

- Mappers emit one `(key, value)` for each input record
  - key is a random key (i.e., a random number)
  - value is the input record
- Reducers emit one `(key, value)` pair for each value in `[list-of-values]` of the input `(key, [list-of-values])` pair

Figure 29.2: Shuffling structure



30 Metapatterns

Metapatterns are used to organize the workflow of a complex application executing many jobs

- Job Chaining

30.1 Job Chaining

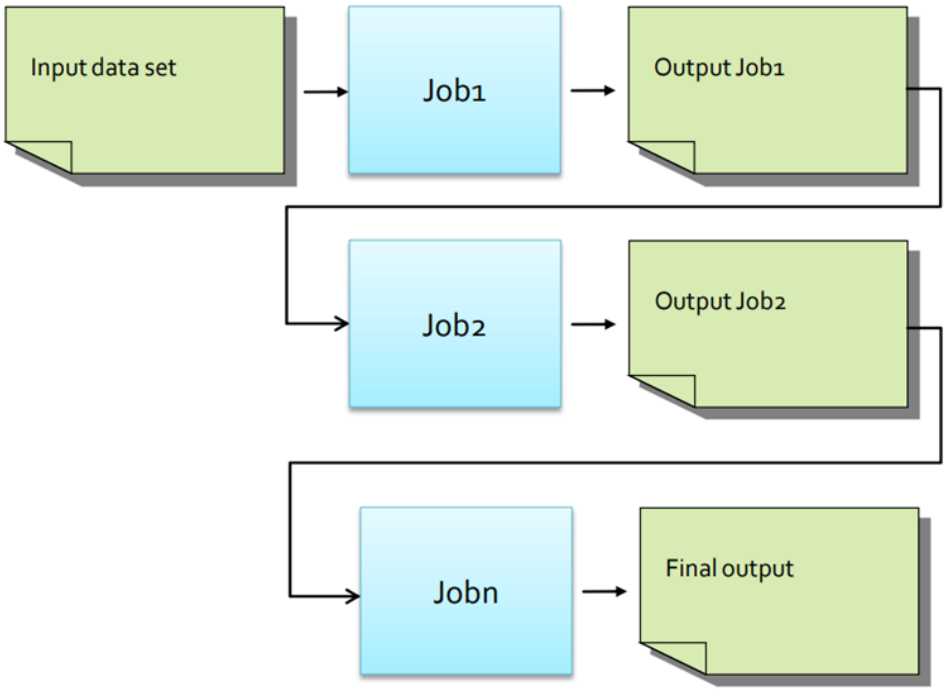
The goal is to execute a sequence of jobs (synchronizing them). Job chaining allows to manage the workflow of complex applications based on many phases (iterations). Each phase is associated with a different MapReduce Job (i.e., one sub-application), and the output of a phase is the input of the next one. This is done because real application are usually based on many phases.

30.1.1 Structure

- The (single) Driver contains the workflow of the application and executes the jobs in the proper order
- Mappers, reducers, and combiners: each phase of the complex application is implement by a MapReduce Job (i.e., it is associated with a mapper, a reducer, and a combiner, if it is useful)

More complex workflows, which execute jobs in parallel, can also be implemented, however, the synchronization of the jobs become more complex.

Figure 30.1: Job chaining structure



31 Join patterns

Are use to implement the join operators of the relational algebra (i.e., the join operators of traditional relational databases)

- Reduce side join
- Map side join

The explanation will focus on the natural join however, the pattern is analogous for the other types of joins (theta-, semi-, outer-join).

31.1 Reduce side natural join

The goal is to join the content of two relations (i.e., relational tables) when both tables are large.

31.1.1 Structure

There are two mapper classes, that is one mapper class for each table. Mappers emit one `(key, value)` pair for each input record

- Key is the value of the common attribute(s)
- Value is the concatenation of the name of the table of the current record and the content of the current record

Example

Suppose join the following tables have to be joined

- ***Users*** with schema *userid*, *name*, *surname*
- ***Likes*** with schema *userid*, *movieGenre*

The values `userid=u1`, `name=Paolo`, `surname=Garza` of the ***Users*** table will generate the pair

```
1  (userid=u1, "Users:name=Paolo, surname=Garza")
```

The values `userid=u1`, `movieGenre=horror` of the ***Likes*** table will generate the pair

`(userid=u1, "Likes:movieGenre=horror")`

The reducers iterate over the values associated with each key (value of the common attributes) and compute the "local natural join" for the current key. So, they generate a copy for each pair of values such that one record is a record of the first table and the other is the record of the other table.

Example

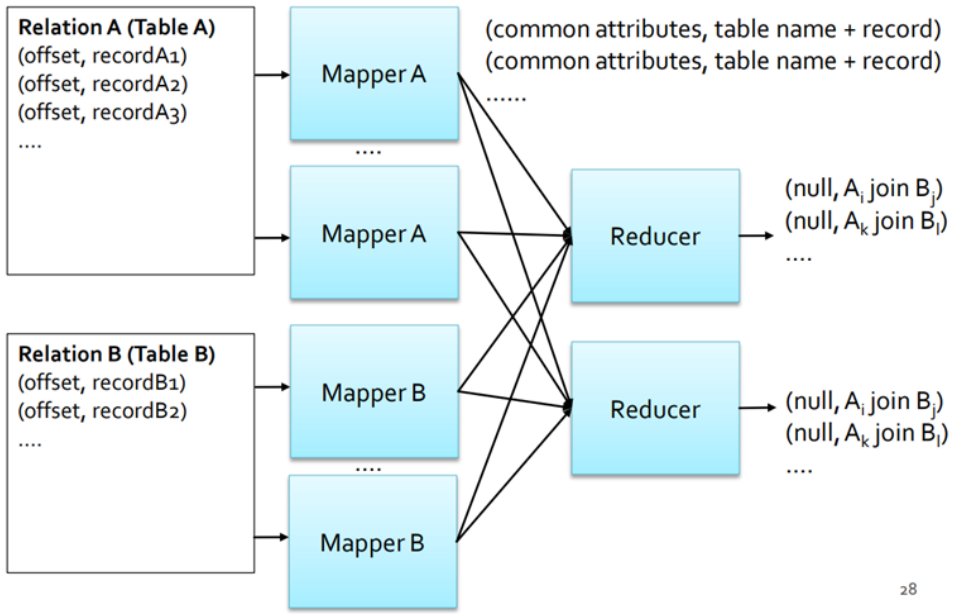
The `(key, [list of values])` pair

```
(userid=u1, ["User:name=Paolo, surname=Garza", "Likes:movieGenre=horror", "Likes:movieGe
```

will generate the following output `(key,value)` pairs

```
(userid=u1, "name=Paolo, surname=Garza, genre=horror")

(userid=u1, "name=Paolo, surname=Garza, genre=adventure")
```

Figure 31.1: Reduce side natural join structure



31.2 Map side natural join

The goal is to join the content of two relations (i.e., relational tables) when one table is large, while the other is small enough to be completely loaded in main memory (frequently one of the two tables is small).

31.2.1 Structure

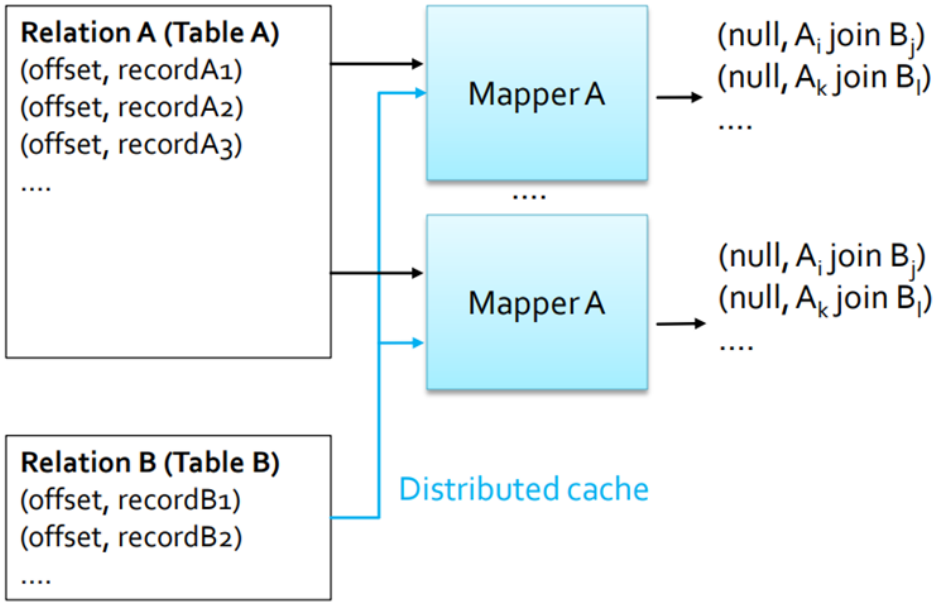
This is a Map-only job

- Mapper class processes the content of the large table: it receives one input `(key,value)` pair for each record of the large table, and joins it with the "small" table.

The distributed cache approach is used to "provide" a copy of the small table to all mappers: each mapper performs the "local natural join" between the current record (of the large table) it is processing and the records of the small table (that is in the distributed cache).

Notice that the content of the small table (file) is loaded in the main memory of each mapper during the execution of its setup method.

Figure 31.2: Map side natural join structure



31.3 Other join patterns

The SQL language is characterized by many types of joins

- Theta-join
- Semi-join
- Outer-join

The same patterns used for implementing the natural join can be used also for the other SQL joins.

The "local join" in the reducer of the reduce side natural join (in the mapper of the map side natural join) is replaced with the type of join of interest (theta-, semi-, or outer-join).

32 Relational Algebra Operations and MapReduce

The relational algebra and the SQL language have many useful operators

- Selection
- Projection
- Union, intersection, and difference
- Join (see Join design patterns)
- Aggregations and Group by (see the Summarization design patterns)

The MapReduce paradigm can be used to implement relational operators, however the MapReduce implementation is efficient only when a full scan of the input table(s) is needed (i.e., when queries are not selective and process all data). Selective queries, which return few tuples/records of the input tables, are usually not efficient when implemented by using a MapReduce approach.

Most preprocessing activities involve relational operators (e.g., ETL processes in the data warehousing application context).

Relations/Tables (also the big ones) can be stored in the HDFS distributed file system, broken in blocks and spread across the servers of the Hadoop cluster.

Notice that in relational algebra, relations/tables do not contain duplicate records by definition, and this constraint must be satisfied by both the input and the output relations/tables.

$$\sigma_C(R)$$

Selection applies predicate (condition) C to each record of table R, and produces a relation containing only the records that satisfy predicate C.

The selection operator can be implemented by using the filtering pattern.

> **Example**
>
> Given the table *Courses*

CCode	CName	Semester	ProfID
M2170	Computer science	1	D102
M4880	Digital systems	2	D104
F1401	Electronics	1	D104
F0410	Databases	2	D102

Find the courses held in the second semester

$$\sigma_{\mathbf{Semester}=2}(\mathbf{Courses})$$

The resulting table is

CCode	CName	Semester	ProfID
M4880	Digital systems	2	D104
F0410	Databases	2	D102

Selection is a map-only job, where each mapper analyzes one record at a time of its split and, if the record satisfies C then it emits a `(key,value)` pair with `key=record` and `value=null`, otherwise, it discards the record.

33 Projection

$$\pi_S(R)$$

Projection, for each record of table R, keeps only the attributes in S. It produces a relation with a schema equal to S (i.e., a relation containing only the attributes in S), an it removes duplicates, if any.

Example

Given the table *Professors*

ProfId	PSurname	Department
D102	Smith	Computer Engineering
D105	Jones	Computer Engineering
D104	Smith	Electronics

Find the surnames of all professors.

$$\pi_{\mathbf{PSurname}}(\mathbf{Professors})$$

The resulting table is

PSurname
Smith
Jones

Notice that duplicated values are removed.

In a projection

- Each mapper analyzes one record at a time of its split, and, for each record r in R, it selects the values of the attributes in S and constructs a new record r', and emits a `(key,value)` pair with `key=r'` and `value=null`.
- Each reducer emits one `(key, value)` pair for each input `(key, [list of values])` pair with `key=r'` and `value=null`.

34 Union

$$R \cup S$$

Given that R and S have the same schema, an union produces a relation with the same schema of R and S. There is a record t in the output of the union operator for each record t appearing in R or S. Duplicated records are removed.

Example

Given the tables *DegreeCourseProf*

ProfId	PSurname	Department
D102	Smith	Computer Engineering
D105	Jones	Computer Engineering
D104	White	Electronics

and *MasterCourseProf*

ProfId	PSurname	Department
D102	Smith	Computer Engineering
D101	Red	Electronics

Find information relative to the professors of degree courses or master's degrees.

$$\textbf{DegreeCourseProf} \cup \textbf{MasterCourseProf}$$

The resulting table is

ProfId	PSurname	Department
D102	Smith	Computer Engineering
D105	Jones	Computer Engineering
D104	White	Electronics
D101	Red	Electronics

In a union

- Mappers, for each input record t in R, emit one `(key, value)` pair with `key=t` and `value=null`, and for each input record t in S, emit one `(key, value)` pair with `key=t` and `value=null`.
- Reducers emit one `(key, value)` pair for each input `(key, [list of values])` pair with `key=t` and `value=null` (i.e., one single copy of each input record is emitted).

35 Intersection

$$R \cap S$$

Given that R and S have the same schema, an intersection produces a relation with the same schema of R and S. There is a record t in the output of the intersection operator if and only if t appears in both relations (R and S).

Example

Given the tables *DegreeCourseProf*

ProfId	PSurname	Department
D102	Smith	Computer Engineering
D105	Jones	Computer Engineering
D104	White	Electronics

and *MasterCourseProf*

ProfId	PSurname	Department
D102	Smith	Computer Engineering
D101	Red	Electronics

Find information relative to professors teaching both degree courses and master's courses.

$$\textbf{DegreeCourseProf} \cap \textbf{MasterCourseProf}$$

The resulting table is

ProfId	PSurname	Department
D102	Smith	Computer Engineering

In an intersection

- Mappers, for each input record t in R, emit one `(key, value)` pair with `key=t` and `value="R"`, and For each input record t in S, emit one `(key, value)` pair with `key=t` and `value="S"`.
- Reducers emit one `(key, value)` pair with `key=t` and `value=null` for each input `(key, [list of values])` pair with `[list of values]` containing two values. Notice that it happens if and only if both R and S contain t.

36 Difference

$$R - S$$

Given that R and S have the same schema, a difference produces a relation with the same schema of R and S. There is a record t in the output of the difference operator if and only if t appears in R but not in S.

Example

Given the tables *DegreeCourseProf*

ProfId	PSurname	Department
D102	Smith	Computer Engineering
D105	Jones	Computer Engineering
D104	White	Electronics

and *MasterCourseProf*

ProfId	PSurname	Department
D102	Smith	Computer Engineering
D101	Red	Electronics

Find the professors teaching degree courses but not master's courses.

$$\textbf{DegreeCourseProf} - \textbf{MasterCourseProf}$$

The resulting table is

ProfId	PSurname	Department
D105	Jones	Computer Engineering
D104	White	Electronics

In a difference

- Mappers, for each input record t in R, emit one `(key, value)` pair with `key=t` and `value=name` of the relation (i.e., R). For each input record t in S, emit one `(key, value)` pair with `key=t` and `value=name` of the relation (i.e., S). Notice that two mapper classes are needed: one for each relation.
- Reducers emit one `(key, value)` pair with `key=t` and `value=null` for each input `(key, [list of values])` pair with `[list of values]` containing only the value R. Notice that it happens if and only if t appears in R but not in S.

37 Join

The join operators can be implemented by using the Join pattern, using the reduce side or the map side pattern depending on the size of the input relations/tables.

38 Aggregations and Group by

Aggregations and Group by are implemented by using the Summarization pattern.

39 How to submit/execute a Spark application

Spark programs are executed (submitted) by using the `spark-submit` command. It is a command line program, characterized by a set of parameters (e.g., the name of the jar file containing all the classes of the Spark application we want to execute, the name of the Driver class, the parameters of the Spark application).

`spark-submit` has also two parameters that are used to specify where the application is executed.

39.1 Options of `spark-submit: --master`

```
1  --master
```

It specifies which environment/scheduler is used to execute the application

`spark://host:port`	The spark scheduler is used
`mesos://host:port`	The mesos scheduler is used
`yarn`	The YARN scheduler (i.e., the one of Hadoop)
`local`	The application is executed exclusively on the local PC

39.2 Options of `spark-submit: --deploy-mode`

```
1  --deploy-mode
```

It specifies where the Driver is launched/executed

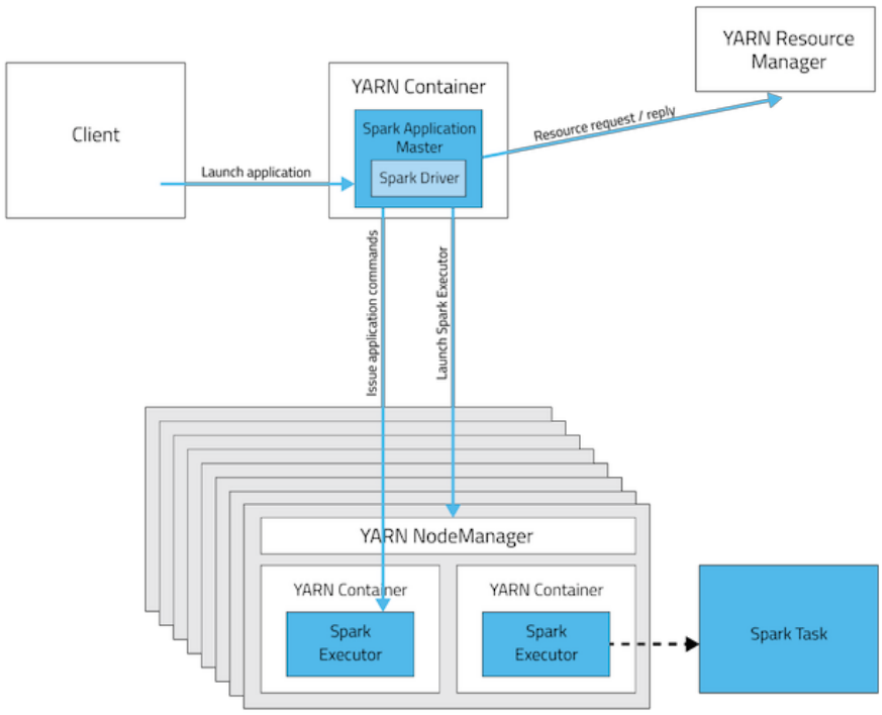
`client`	The driver is launched locally (in the "local" PC executing spark-submit)
`cluster`	The driver is launched on one node of the cluster

Deployment mode: cluster and client

In **cluster** mode

- The Spark driver runs in the ApplicationMaster on a cluster node.
- The cluster nodes are used also to store RDDs and execute transformations and actions on the RDDs
- A single process in a YARN container is responsible for both driving the application and requesting resources from YARN.
- The resources (memory and CPU) of the client that launches the application are not used.

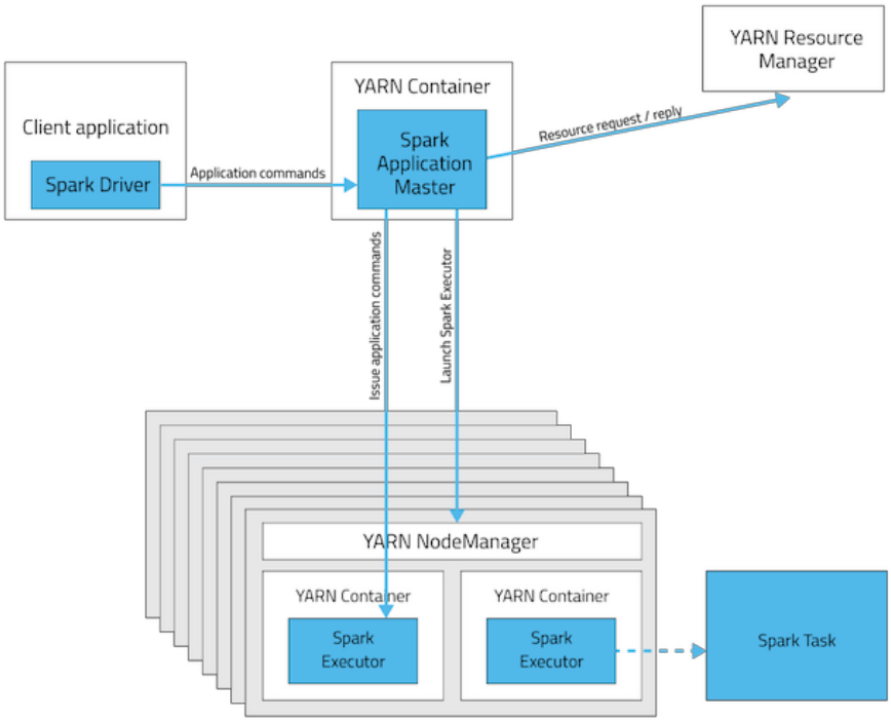
Figure 39.1: Cluster deployment mode



In **client** mode

- The Spark driver runs on the host where the job is submitted (i.e., the resources of the client are used to execute the Driver)
- The cluster nodes are used to store RDDs and execute transformations and actions on the RDDs
- The ApplicationMaster is responsible only for requesting executor containers from YARN.

Figure 39.2: Client deployment mode



39.3 Setting the executors

`spark-submit` allows specifying the characteristics of the executors

option	meaning	default value
`--num-executors`	The number of executors	2 executors
`--executor-cores`	The number of cores per executor	1 core
`--executor-memory`	Main memory per executor	1 GB

Notice that the maximum values of these parameters are limited by the configuration of the cluster.

39.4 Setting the drivers

`spark-submit` allows specifying the characteristics of the driver

option	meaning	default value
–driver-cores	The number of cores for the driver	1 core
–driver-memory	Main memory for the driver	1 GB

Also the maximum values of these parameters are limited by the configuration of the cluster when the `--deploy-mode` is set to `cluster`.

39.5 Execution examples

The following command submits a Spark application on a Hadoop cluster

```
spark-submit \
--deploy-mode cluster \
--master yarn MyApplication.py arguments
```

It executes/submits the application contained in `MyApplication.py`, and the application is executed on a Hadoop cluster based on the YARN scheduler. Notice that the Driver is executed in a node of cluster.

The following command submits a Spark application on a local PC

```
spark-submit \
--deploy-mode client \
--master local MyApplication.py arguments
```

It executes/submits the application contained in `MyApplication.py`. Notice that the application is completely executed on the local PC:

- Both Driver and Executors
- Hadoop is not needed in this case
- Only the Spark software is needed

40 Introduction to Spark

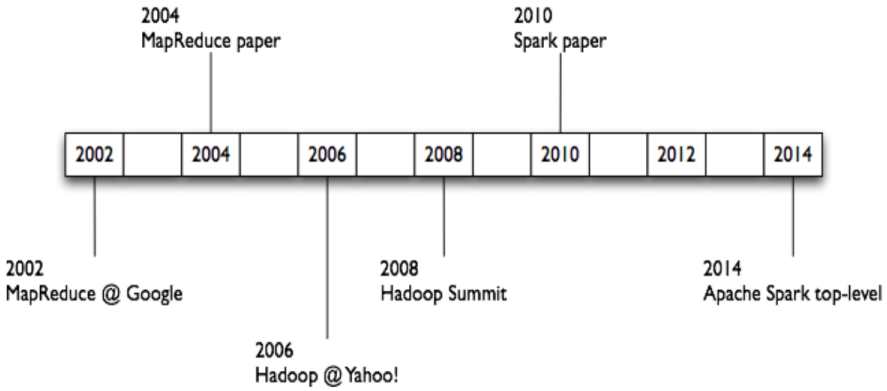
Apache SparkTM is a fast and general-purpose engine for large-scale data processing. Spark aims at achieving the following goals in the Big data context:

- Generality: diverse workloads, operators, job sizes
- Low latency: sub-second
- Fault tolerance: faults are the norm, not the exception
- Simplicity: often comes from generality

Tip

Originally developed at the University of California - Berkeley's AMPLab

Figure 40.1: Spark history

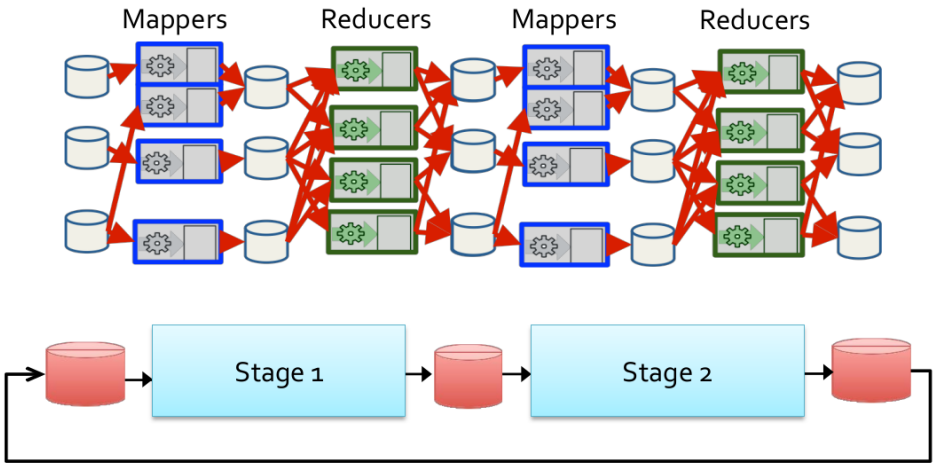


41 Motivations

41.1 MapReduce and Spark iterative jobs and data I/O

Iterative jobs, with MapReduce, involve a lot of disk I/O for each iteration and stage, and disk I/O is very slow (even if it is local I/O)

Figure 41.1: Iterative jobs



- Motivation: using MapReduce for complex iterative jobs or multiple jobs on the same data involves lots of disk I/O
- Opportunity: the cost of main memory decreased, hence, large main memories are available in each server
- Solution: keep more data in main memory, and that's the basic idea of Spark

So an iterative job in MapReduce makes wide use of disk reading/writing

Instead, an iterative job in Spark uses the main memory

Data (or at least part of it) are shared between the iterations by using the main memory , which is 10 to 100 times faster than disk.

Moreover, to run multiple queries on the same data, in MapReduce the data must be read multiple times (once for each query)

Figure 41.2: Iterative jobs in MapReduce

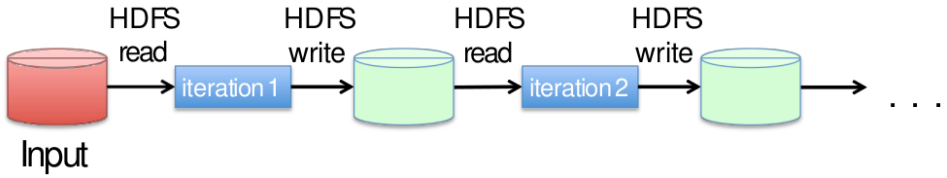


Figure 41.3: Iterative jobs in Spark

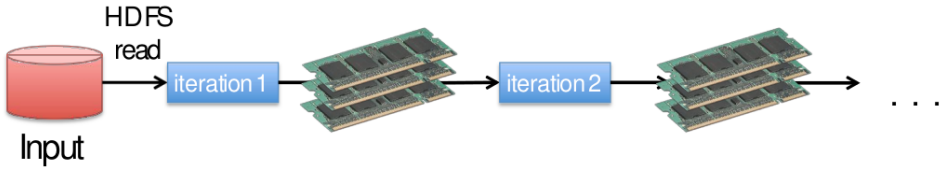
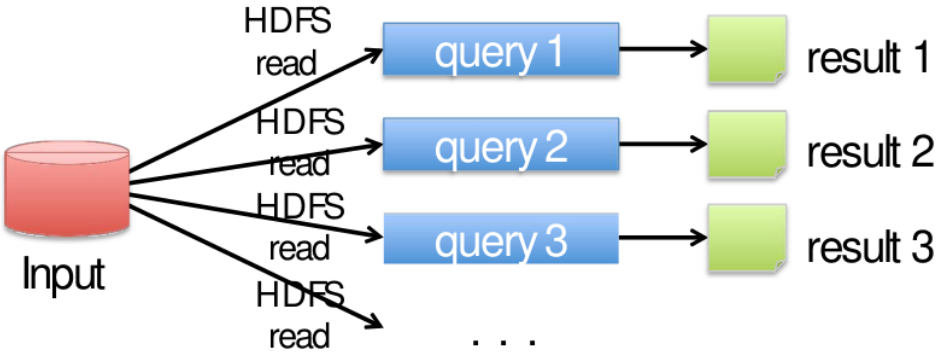
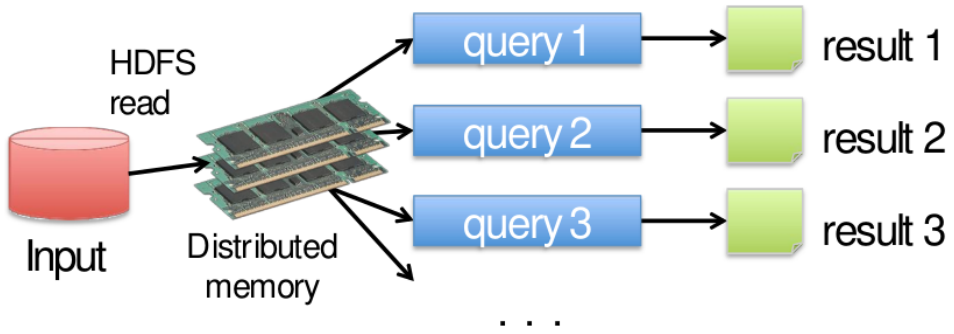


Figure 41.4: Analysing the same data in MapReduce



Instead, in Spark the data have to be loaded only once in the main memory

Figure 41.5: Analysing the same data in Spark



In other words, data are read only once from HDFS and stored in main memory, splitting of the data across the main memory of each server.

41.2 Resilient distributed data sets (RDDs)

In Spark, data are represented as Resilient Distributed Datasets (RDDs), which are Partitioned/Distributed collections of objects spread across the nodes of a cluster, and are stored in main memory (when it is possible) or on local disk.

Spark programs are written in terms of operations on resilient distributed data sets.

RDDs are built and manipulated through a set of parallel transformations (e.g., map, filter, join) and actions (e.g., count, collect, save), and RDDs are automatically rebuilt on machine failure.

The Spark computing framework provides a programming abstraction (based on RDDs) and transparent mechanisms to execute code in parallel on RDDs

- It hides complexities of fault-tolerance and slow machines
- It manages scheduling and synchronization of the jobs

41.3 MapReduce vs Spark

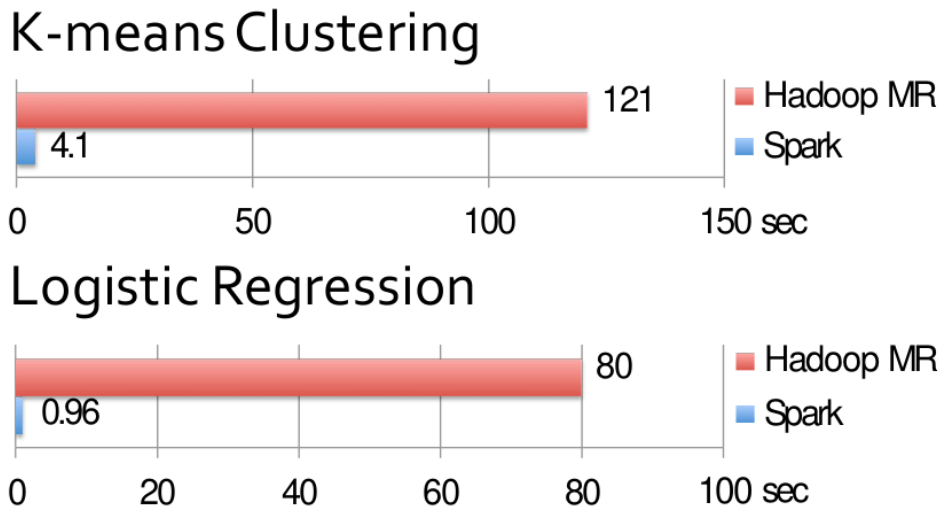
	Hadoop MapReduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, …
Execution model	Batch	Batch, interactive, streaming

	Hadoop MapReduce	Spark
Programming environments	Java	Scala, Java, Python, R

With respect to MapReduce, Spark has a lower overhead for starting jobs and has less expensive shuffles.

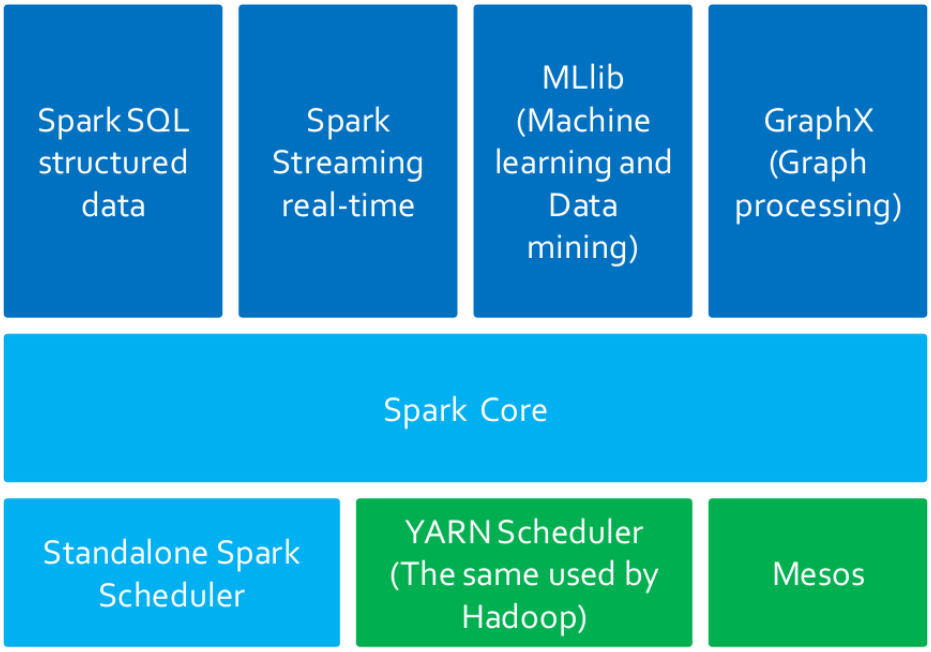
In-memory RDDs can make a big difference in performance

Figure 41.6: Perfomance comparison



42 Main components

Figure 42.1: Spark main components



Spark is based on a basic component (the Spark Core component) that is exploited by all the high-level data analytics components: this solution provides a more uniform and efficient solution with respect to Hadoop where many non-integrated tools are available. In this way, when the efficiency of the core component is increased also the efficiency of the other high-level components increases.

42.0.1 Spark Core

Spark Core contains the basic functionalities of Spark exploited by all components

- Task scheduling
- Memory management
- Fault recovery

- …

It provides the APIs that are used to create RDDs and applies transformations and actions on them.

42.0.2 Spark SQL

Spark SQL for structured data is used to interact with structured datasets by means of the SQL language or specific querying APIs (based on Datasets).

It exploits a query optimizer engine, and supports also Hive Query Language (HQL). It interacts with many data sources (e.g., Hive Tables, Parquet, Json).

42.0.3 Spark Streaming

Spark Streaming for real-time data is used to process live streams of data in real-time. The APIs of the Streaming real-time components operated on RDDs and are similar to the ones used to process standard RDDs associated with "static" data sources.

42.0.4 MLlib

MLlib is a machine learning/data mining library that can be used to apply the parallel versions of some machine learning/data mining algorithms

- Data preprocessing and dimensional reduction
- Classification algorithms
- Clustering algorithms
- Itemset mining
- …

42.0.5 GraphX and GraphFrames

GraphX is a graph processing library that provides algorithms for manipulating graphs (e.g., subgraph searching, PageRank). Notice that the Python version is not available.

GraphFrames is a graph library based on DataFrames and Python.

42.0.6 Spark schedulers

Spark can exploit many schedulers to execute its applications

- Hadoop YARN: it is the standard scheduler of Hadoop
- Mesos cluster: another popular scheduler
- Standalone Spark Scheduler: a simple cluster scheduler included in Spark

43 Basic concepts

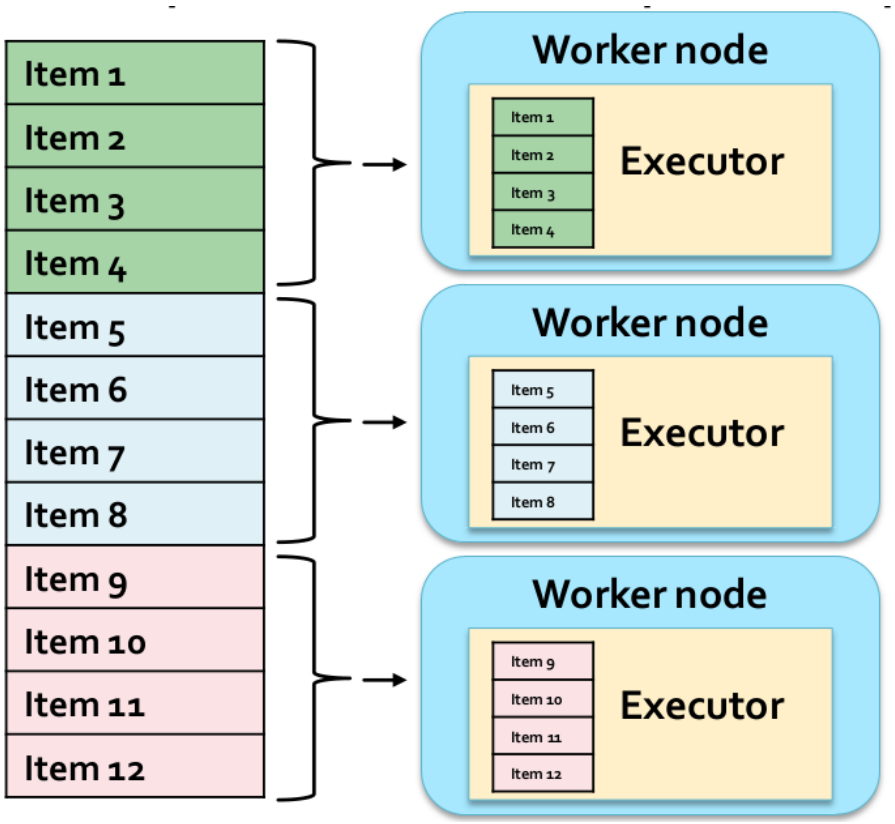
43.1 Resilient Distributed Data sets (RDDs)

RDDs are the primary abstraction in Spark: they are distributed collections of objects spread across the nodes of a clusters, which means that they are split in partitions, and each node of the cluster that is running an application contains at least one partition of the RDD(s) that is (are) defined in the application.

RDDs are stored in the main memory of the executors running in the nodes of the cluster (when it is possible) or in the local disk of the nodes if there is not enough main memory. This allows to execute in parallel the code invoked on eah node: each executor of a worker node runs the specified code on its partition of the RDD.

Example of an RDD split in 3 partitions

Figure 43.1: Example of RDD splits



More partitions mean more parallelism.

RDDs are immutable once constructed (i.e., the content of an RDD cannot be modified). Spark tracks lineage information to efficiently recompute lost data in case of failures of some executors: for each RDD, Spark knows how it has been constructed and can rebuilt it if a failure occurs. This information is represented by means of a DAG (Direct Acyclic Graph) connecting input data and RDDs.

RDDs can be created

- by parallelizing existing collections of the hosting programming language (e.g., collections and lists of Scala, Java, Pyhton, or R): in this case the number of partition is specified by the user
- from (large) files stored in HDFS: in this case there is one partition per HDFS block
- from files stored in many traditional file systems or databases

- by transforming an existing RDDs: in this case the number of partitions depends on the type of transformation

Spark programs are written in terms of operations on resilient distributed data sets

- Transformations: map, filter, join, …
- Actions: count, collect, save, …

To summarize, in the Spark framework

- Spark manages scheduling and synchronization of the jobs
- Spark manages the split of RDDs in partitions and allocates RDDs partitions in the nodes of the cluster
- Spark hides complexities of fault-tolerance and slow machines (RDDs are automatically rebuilt in case of machine failures)

44 Spark Programs

44.1 Supported languages

Spark supports many programming languages

- Scala: this is the language used to develop the Spark framework and all its components (Spark Core, Spark SQL, Spark Streaming, MLlib, GraphX)
- Java
- Python
- R

44.2 Structure of Spark programs

Spark official terminology

Term	Definition
Application	User program built on Spark, consisting of a driver program and executors on the cluster.
Driver program	The process running the `main()` function of the application and creating the `SparkContext`.
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN).
Deploy mode	It distinguishes where the driver process runs: in "cluster" mode (in this case the framework launches the driver inside of the cluster) or in "client" mode (in this case the submitter launches the driver outside of the cluster).
Worker node	Any node of the cluster that can run application code in the cluster.
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them; each application has its own executors.
Task	A unit of work that will be sent to one executor.

Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect).
Stage	Each job gets divided into smaller sets of tasks called stages, such that the output of one stage is the input of the next stage(s), except for the stages that compute (part of) the final result (i.e., the stages without output edges in the graph representing the workflow of the application). Indeed, the outputs of those stages is stored in HDFS or a database.

The shuffle operation is always executed between two stages

- Data must be grouped/repartitioned based on a grouping criteria that is different with respect to the one used in the previous stage
- Similar to the shuffle operation between the map and the reduce phases in MapReduce
- Shuffle is a heavy operation

See the official documentation for more.

The Driver program contains the main method. It defines the workflow of the application, and accesses Spark through the `SparkContext` object, which represents a connection to the cluster.

The Driver program defines Resilient Distributed Datasets (RDDs) that are allocated in the nodes of the cluster, and invokes parallel operations on RDDs.

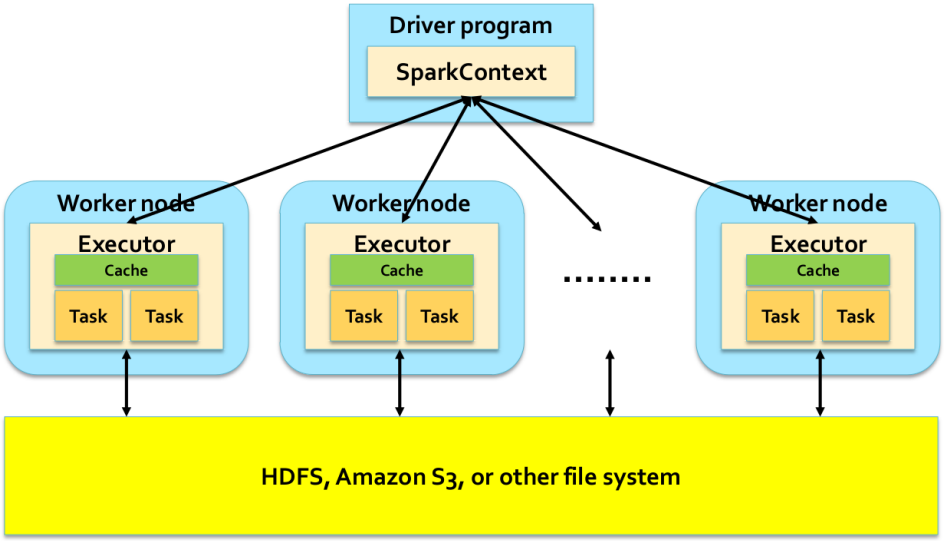
The Driver program defines

- Local variables: these are standard variables of the Python programs
- RDDs: these are distributed variables stored in the nodes of the cluster
- The `SparkContext` object, which allows to
  - create RDDs
  - submit executors (processes) that execute in parallel specific operations on RDDs
  - perform Transformations and Actions

The worker nodes of the cluster are used to run your application by means of executors. Each executor runs on its partition of the RDD(s) the operations that are specified in the driver.

RDDs are distributed across executors (each RDD is split in partitions that are spread across the available executors).

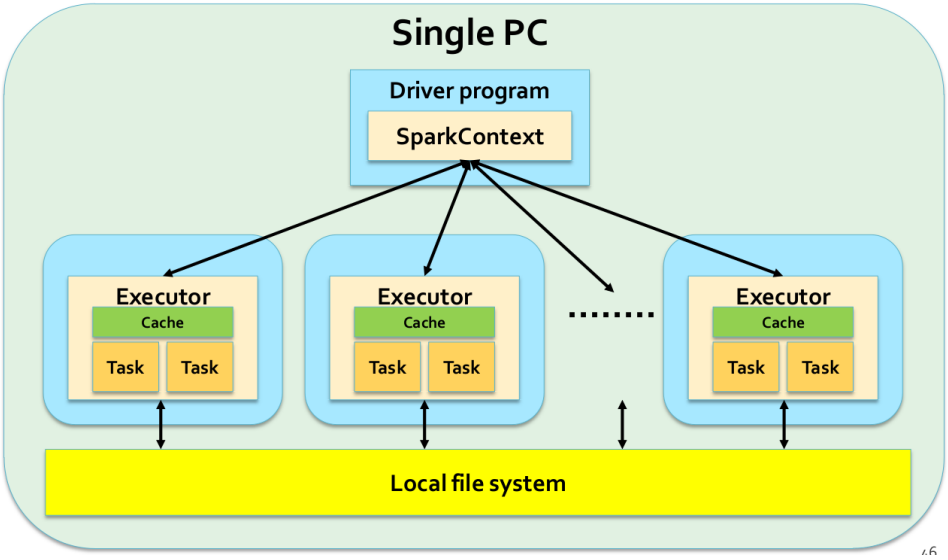
Figure 44.1: Distributed execution of Spark



44.3 Local execution of Spark

Spark programs can also be executed locally: local threads are used to parallelize the execution of the application on RDDs on a single PC. Local threads can be seen are "pseudo-worker" nodes, and a local scheduler is launched to run Spark programs locally. It is useful to develop and test the applications before deploying them on the cluster.

Figure 44.2: Distributed execution of Spark



45 Spark program examples

45.1 Count line program

The steps of this program are

- count the number of lines of the input file, whose name is set to “myfile.txt”
- print the results on the standard output

```
from pyspark import SparkConf, SparkContext

if __name__ == "__main__":

    # Create a configuration object and
    # set the name of the application
    conf = SparkConf().setAppName("Spark Line Count") # <1>

    # Create a Spark Context object
    sc = SparkContext(conf=conf) # <1>

    # Store the path of the input file in inputfile
    inputFile= "myfile.txt" # <1>

    # Build an RDD of Strings from the input textual file
    # Each element of the RDD is a line of the input file
    linesRDD = sc.textFile(inputFile) # <2>

    # Count the number of lines in the input file
    # Store the returned value in the local variable numLines
    numLines = linesRDD.count() # <1>

    # Print the output in the standard output
    print("NumLines:", numLines)

    # Close the Spark Context object
    sc.stop()
```

1. Local Python variable: it is allocated in the main memory of the same process instancing the Driver.
2. It is allocated/stored in the main memory or in the local disk of the executors of the worker nodes.

- Local variables can be used to store only "small" objects/data (i.e., the maximum size is equal to the main memory of the process associated with the Driver)
- RDDs are used to store "big/large" collections of objects/data in the nodes of the cluster
  - In the main memory of the worker nodes, when it is possible
  - In the local disks of the worker nodes, when it is necessary

45.2 Word Count program

In the Word Count implemented by means of Spark

- The name of the input file is specified by using a command line parameter (i.e., `argv[1]`)
- The output of the application (i.e., the pairs (word, number of occurrences) are stored in an output folder (i.e., `argv[2]`))

Notice that there is no need to worry about the details.

```
from pyspark import SparkConf, SparkContext
import sys

if __name__ == "__main__":
    """
    Word count example
    """
    inputFile= sys.argv[1]
    outputPath = sys.argv[2]

    # Create a configuration object and
    # set the name of the application
    conf = SparkConf().setAppName("Spark Word Count")

    # Create a Spark Context object
    sc = SparkContext(conf=conf)

    # Build an RDD of Strings from the input textual file
    # Each element of the RDD is a line of the input file
    lines = sc.textFile(inputFile)
```

```
21
22      # Split/transform the content of lines in a
23      # list of words and store them in the words RDD
24      words = lines.flatMap(lambda line: line.split(sep=' '))
25
26      # Map/transform each word in the words RDD
27      # to a pair/tuple (word,1) and store the result
28      # in the words_one RDD
29      words_one = words.map(lambda word: (word, 1))
30
31      # Count the num. of occurrences of each word.
32      # Reduce by key the pairs of the words_one RDD and store
33      # the result (the list of pairs (word, num. of occurrences)
34      # in the counts RDD
35      counts = words_one.reduceByKey(lambda c1, c2: c1 + c2)
36
37      # Store the result in the output folder
38      counts.saveAsTextFile(outputPath)
39
40      # Close/Stop the Spark Context object
41      sc.stop()
```

46 RDD based programming

The "connection" of the driver to the cluster is based on the Spark Context object

- In Python the name of the class is `SparkContext`
- The Spark Context is built by means of the constructor of the `SparkContext` class
- The only parameter is a configuration object

i Example

```
# Create a configuration object and
# set the name of the application
conf = SparkConf().setAppName("Application name")

# Create a Spark Context object
sc = SparkContext(conf=conf)
```

The Spark Context object can be obtained also by using the `SparkContext.getOrCreate(conf)` method, whose only parameter is a configuration object. Notice that, if the `SparkContext` object already exists for this application, the current `SparkContext` object is returned, otherwise, a new `SparkContext` object is returned: there is always one single `SparkContext` object for each application.

i Example

```
# Create a configuration object and
# set the name of the application
conf = SparkConf().setAppName("Application name")

# Retrieve the current SparkContext object or
# create a new one
sc = SparkContext.getOrCreate(conf=conf)
```

47 RDD basics

A Spark RDD is an immutable distributed collection of objects. Each RDD is split in partitions, allowing to parallelize the code based on RDDs (i.e., code is executed on each partition in isolation).

RDDs can contain any type of Scala, Java, and Python objects, including user-defined classes.

48 RDD: create and save

RDDs can be created

- By loading an external dataset (e.g., the content of a folder, a single file, a database table)
- By parallelizing a local collection of objects created in the Driver (e.g., a Java collection)

48.1 Create RDDs from files

To built an RDD from **an input textual file**, use the `textFile(name)` method of the `SparkContext` class.

- The returned RDD is an RDD of Strings associated with the content of the name textual file;
- Each line of the input file is associated with an object (a string) of the instantiated RDD;
- By default, if the input file is an HDFS file the number of partitions of the created RDD is equal to the number of HDFS blocks used to store the file, in order to support data locality.

Example

```
# Build an RDD of strings from the input textual file
# myfile.txt
# Each element of the RDD is a line of the input file
inputFile = "myfile.txt"
lines = sc.textFile(inputFile)
```

Notice that no computation occurs when `sc.textFile()` is invoked: Spark only records how to create the RDD, and the data is lazily read from the input file only when the data is needed (i.e., when an action is applied on lines, or on one of its "descendant" RDDs).

To build an RDD from a **folder containing textual files**, use the `textFile(name)` method of the `SparkContext` class.

- If name is the path of a folder all files inside that folder are considered;

- The returned RDD contains one string for each line of the files contained on the name folder.

Example

```
# Build an RDD of strings from all the files stored in
# myfolder
# Each element of the RDD is a line of the input files
inputFolder = "myfolder/"
lines = sc.textFile(inputFolder)
```

Notice that all files inside myfolder are considered, also those without suffix or with a suffix different from ".txt".

To set the (minimum) **number of partitions**, use the `textFile(name, minPartitions)` method of the `SparkContext` class.

- This option can be used to increase the parallelization of the submitted application;
- For the HDFS files, the number of partitions `minPartitions` must be greater than the number of blocks/chunks.

Example

```
# Build an RDD of strings from the input textual file
# myfile.txt
# The number of partitions is manually set to 4
# Each element of the RDD is a line of the input file
inputFile = "myfile.txt"
lines = sc.textFile(inputFile, 4)
```

48.2 Create RDDs from a local Python collection

An RDD can be built from a local Python collection/list of local python objects using the `parallelize(c)` method of the `SparkContext` class

- The created RDD is an RDD of objects of the same type of objects of the input python collection `c`
- In the created RDD, there is one object for each element of the input collection
- Spark tries to set the number of partitions automatically based on your cluster's characteristics

Example

```python
# Create a local python list
inputList = ['First element', 'Second element', 'Third element']

# Build an RDD of Strings from the local list.
# The number of partitions is set automatically by Spark
# There is one element of the RDD for each element
# of the local list
distRDDList = sc.parallelize(inputList)
```

Notice that no computation occurs when `sc.parallelize(c)` is invoked: Spark only records how to create the RDD, and the data is lazily read from the input file only when the data is needed (i.e., when an action is applied on `distRDDlist`, or on one of its "descendant" RDDs).

When the `parallelize(c)` is invoked, Spark tries to set the number of partitions automatically based on the cluster's characteristics, but the developer can set the number of partition by using the method `parallelize(c, numSlices)` of the `SparkContext` class.

Example

```python
# Create a local python list
inputList = ['First element', 'Second element', 'Third element']

# Build an RDD of Strings from the local list.
# The number of partitions is set to 3
# There is one element of the RDD for each element
# of the local list
distRDDList = sc.parallelize(inputList, 3)
```

48.3 Save RDDs

An RDD can be easily stored in textual (HDFS) files using the `saveAsTextFile(path)` method of the `RDD` class

- `path` is the path of a folder
- The method is invoked on the RDD to store in the output folder

- Each object of the RDD on which the `saveAsTextFile` method is invoked is stored in one line of the output files stored in the output folder, and there is one output file for each partition of the input RDD.

Example

```
# Store the content of linesRDD in the output folder
# Each element of the RDD is stored in one line
# of the textual files of the output folder
outputPath="risFolder/"
linesRDD.saveAsTextFile(outputPath)
```

Notice that `saveAsTextFile()` is an action, hence Spark computes the content associated with `linesRDD` when `saveAsTextFile()` is invoked. Spark computes the content of an RDD only when that content is needed.
Moreover, notice that the output folder contains one textual file for each partition of `linesRDD`, such that each output file contains the elements of one partition.

48.4 Retrieve the content of RDDs and store it local Python variables

The content of an RDD can be retrieved from the nodes of the cluster and stored in a local python variable of the Driver using the `collect()` method of the `RDD` class.

The `collect()` method of the `RDD` class is invoked on the RDD to retrieve. It returns a local python list of objects containing the same objects of the considered RDD.

Warning

Pay attention to the size of the RDD: large RDDs cannot be stored in a local variable of the Driver.

Example

```
# Retrieve the content of the linesRDD and store it
# in a local python list
# The local python list contains a copy of each
# element of linesRDD
contentOfLines=linesRDD.collect()
```

`contentOfLines`	Local python variable: it is allocated in the main memory of the Driver process/task
`linesRDD`	RDD of strings: it is distributed across the nodes of the cluster

49 Transformations and Actions

RDD support two types of operations

- Transformations
- Actions

49.1 Transformations

Transformations are operations on RDDs that **return a new RDD**. This type of operation apply a transformation on the elements of the input RDD(s) and the result of the transformation is stored in/associated with a new RDD.

Remember that RDDs are immutable, hence the content of an already existing RDD cannot be changed, and it only possible to applied a transformation on the content of an RDD and then store/assign the result in/to a new RDD.

Transformations are computed **lazily**, which means that transformations are computed (executed) only when an action is applied on the RDDs generated by the transformation operations. When a transformation is invoked, Spark keeps only track of the dependency between the input RDD and the new RDD returned by the transformation, and the content of the new RDD is not computed.

The graph of dependencies between RDDs represents the information about which RDDs are used to create a new RDD. This is called **lineage graph**, and it is represented as a **DAG (Directed Acyclic Graph)**: it is needed to compute the content of an RDD the first time an action is invoked on it, or to compute again the content of an RDD (or some of its partitions) when failures occur.

The lineage graph is also useful for **optimization** purposes: when the content of an RDD is needed, Spark can consider the chain of transformations that are applied to compute the content of the needed RDD and potentially decide how to execute the chain of transformations. In this way, Spark can potentially change the order of some transformations or merge some of them based on its optimization engine.

49.2 Actions

Actions are operations that

- return **results to the Driver program** (i.e., return local python variables). Pay attention to the size of the returned results because they must be stored in the main memory of the Driver program.
- write the result in the storage (output file/folder). The size of the result can be large in this case since it is directly stored in the (distributed) file system.

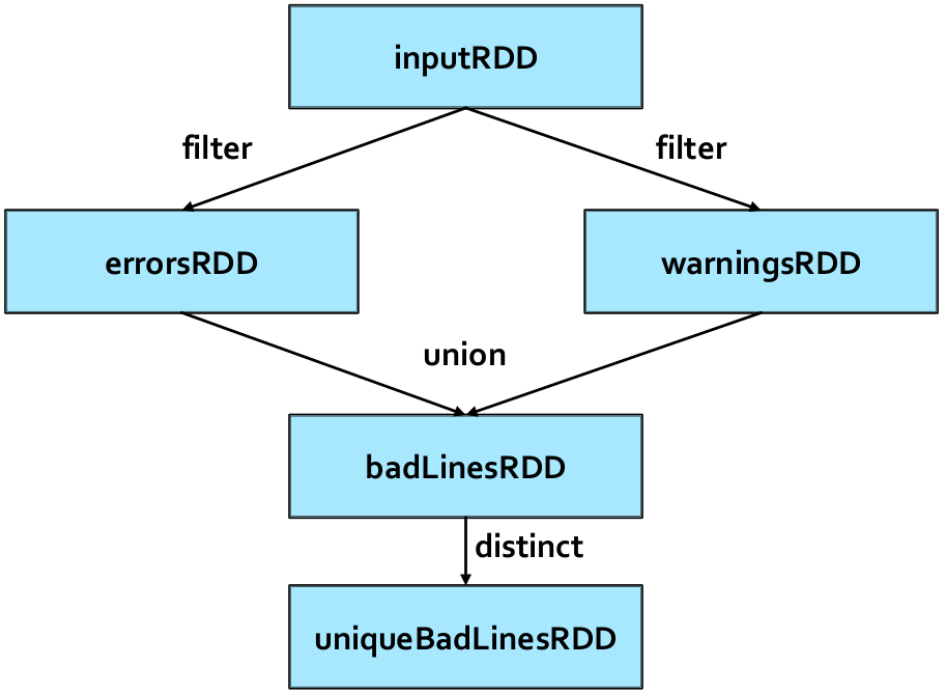
49.2.1 Example of lineage graph (DAG)

Consider the following code

```
from pyspark import SparkConf, SparkContext
import sys

if __name__ == "__main__":
conf = SparkConf().setAppName("Spark Application")
sc = SparkContext(conf=conf)

# Read the content of a log file
inputRDD = sc.textFile("log.txt")

# Select the rows containing the word "error"
errorsRDD = inputRDD.filter(lambda line: line.find('error')>=0)

# Select the rows containing the word "warning"
warningRDD = inputRDD.filter(lambda line: line.find('warning')>=0)

# Union of errorsRDD and warningRDD
# The result is associated with a new RDD: badLinesRDD
badLinesRDD = errorsRDD.union(warningRDD)

# Remove duplicates lines (i.e., those lines containing
# both "error" and "warning")
uniqueBadLinesRDD = badLinesRDD.distinct()

# Count the number of bad lines by applying
# the count() action
numBadLines = uniqueBadLinesRDD.count()
```

```
28
29 # Print the result on the standard output of the driver
30 print("Lines with problems:", numBadLines)
```

Figure 49.1: Visual representation of the DAG



Notice that:

- The application reads the input log file only when the `count()` action is invoked: this is the first action of the program;
- `filter()`, `union()`, and `distinct()` are transformations, so they are computed lazily;
- Also `textFile()` is computed lazily, however it is not a transformation because it is not applied on an RDD.

Spark, similarly to an SQL optimizer, can potentially optimize the execution of some transformations; for instance, in this case the two filters + union + distinct can be potentially optimized and transformed in one single filter applying the constraint (i.e. The element contains the string "error" or "warning"). This optimization improves the efficiency of the application, but Spark can performs this kind of optimizations only on particular types of RDDs: Datasets and DataFrames.

50 Passing functions to Transformations and Actions

Many transformations (and some actions) are based on user provided functions that specify which transformation function must be applied on the elements of the input RDD. For example, the `filter()` transformation selects the elements of an RDD satisfying a user specified constraint, which is a Boolean function applied on each element of the input RDD.

Each language has its own solution to pass functions to Spark's transformations and actions. In Python, it is possible to use

- Lambda functions/expressions: simple functions that can be written as one single expression
- Local user defined functions (local defs): used for multi-statement functions or statements that do not return a value

50.1 Example based on the filter transformation

1. Create an RDD from a log file;
2. Create a new RDD containing only the lines of the log file containing the word "error". The `filter()` transformation applies the filter constraint on each element of the input RDD; the filter constraint is specified by means of a Boolean function that returns true for the elements satisfying the constraint and false for the others.

50.1.1 Solution based on lambda expressions (`lambda`)

```
# Read the content of a log file
inputRDD = sc.textFile("log.txt")

# Select the rows containing the word "error"
errorsRDD = inputRDD.filter(lambda l: l.find('error')>=0)
```

`lambda l: l.find('error')>=0`	This part of the code, which is based on a lambda expression, defines on the fly the function to apply. This part of the code is applied on each object of `inputRDD`: if it returns true then the current object is "stored" in the new `errorsRDD` RDD, otherwise the input object is discarded.

50.1.2 Solution based on function (`def`)

```
# Define the content of the Boolean function that is applied
# to select the elements of interest
def myFunction(l):
    if l.find('error')>=0: return True
    else: return False

# Read the content of a log file
inputRDD = sc.textFile("log.txt")

# Select the rows containing the word "error"
errorsRDD = inputRDD.filter(myFunction)
```

`def myFunction(l):`	When it is invoked, this function analyses the value of the parameter line and returns True if the string line contains the substring "error". Otherwise, it returns False.
`.filter(myFunction)`	For each object `o` in `inputRDD`, the `myFunction` function is automatically invoked. If `myFunction` returns True, then `o` is stored in the new RDD `errorsRDD`. Otherwise, `o` is discarded.

50.1.3 Solution based on function (`def`)

```
# Define the content of the Boolean function that is applied
# to select the elements of interest
def myFunction(l):
    return l.find('error')>=0

# Read the content of a log file
inputRDD = sc.textFile("log.txt")

# Select the rows containing the word "error"
```

```
10 errorsRDD = inputRDD.filter(myFunction)
```

`return l.find('error')>=0`	This part of the code is the same used in the lambda-based solution.
`.filter(myFunction)`	For each object `o` in `inputRDD`, the `myFunction` function is automatically invoked. If `myFunction` returns True, then `o` is stored in the new RDD `errorsRDD`. Otherwise, `o` is discarded.

50.1.4 Solution comparison

The two solutions are more or less equivalent in terms of efficiency

Lambda function-based code (`lambda`)	Local user defined functions (local `def`)
More concise	Less concise
More readable	Less readable
Multi-statement functions or statements that do not return a value are not supported	Multi-statement functions or statements that do not return a value are supported
Code cannot be reused	Code can be reused (some functions are used in several applications)

51 Basic Transformations

- Some basic transformations analyze the content of one single RDD and return a new RDD (e.g., `filter()`, `map()`, `flatMap()`, `distinct()`, `sample()`)
- Some other transformations analyze the content of two (input) RDDs and return a new RDD (e.g., `union()`, `intersection()`, `substract()`, `cartesian()`)

51.1 Single input RDD transformations

51.1.1 Filter transformation

The filter transformation is applied on one single RDD and returns a new RDD containing only the elements of the input RDD that satisfy a user specified condition.

The filter transformation is based on the `filter(f)` method of the `RDD` class: a function `f` returning a Boolean value is passed to the `filter` method, where `f` contains the code associated with the condition that we want to apply on each element `e` of the input RDD. If the condition is satisfied then the call method returns true and the input element `e` is selected, otherwise, it returns false and the `e` element is discarded.

Example 1

1. Create an RDD from a log file
2. Create a new RDD containing only the lines of the log file containing the word "error".

```
# Read the content of a log file
inputRDD = sc.textFile("log.txt")

# Select the rows containing the word "error"
errorsRDD = inputRDD.filter(lambda e: e.find('error')>=0)
```

Notice that, in this case, the input RDD contains strings, hence, the implemented lambda function is applied on one string at a time and returns a Boolean value.

Example 2

1. Create an RDD of integers containing the values `[1, 2, 3, 3]`;
2. Create a new RDD containing only the values greater than `2`.

Using `lambda`

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList);

# Select the values greater than 2
greaterRDD = inputRDD.filter(lambda num : num>2)
```

Notice that the input RDD contains integers, hence, the implemented lambda function is applied on one integer at a time and returns a Boolean value.

Using `def`

```
# Define the function to be applied in the filter transformation
def greaterThan2(num):
    return num>2

# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList);

# Select the values greater than 2
greaterRDD = inputRDD.filter(greaterThan2)
```

In this case, the function to apply is defined using `def` and then is passed to the filter transformation.

51.1.2 Map transformation

The map transformation is used to create a new RDD by applying a function `f` on each element of the input RDD: the new RDD contains exactly one element `y` for each element `x` of the input RDD, in particular the value of `y` is obtained by applying a user defined function `f` on `x` (e.g., `y= f(x)`). The data type of `y` can be different from the data type of `x`.

The map transformation is based on the RDD `map(f)` method of the RDD class: a function `f` implementing the transformation is passed to the `map` method, where `f` contains the code that

is applied over each element of the input RDD to create the elements of the returned RDD. For each input element of the input RDD exactly one single new element is returned by `f`.

Example 1

1. Create an RDD from a textual file containing the surnames of a list of users (each line of the file contains one surname);
2. Create a new RDD containing the length of each surname.

```
# Read the content of the input textual file
inputRDD = sc.textFile("usernames.txt")

# Compute the lengths of the input surnames
lenghtsRDD = inputRDD.map(lambda line: len(line))
```

Notice that the input RDD is an RDD of strings, hence, also the input of the lambda function is a String. Instead, the new RDD is an RDD of Integers, since the lambda function returns a new Integer for each input element.

Example 2

1. Create an RDD of integers containing the values `[1, 2, 3, 3]`;
2. Create a new RDD containing the square of each input element.

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList)

# Compute the square of each input element
squaresRDD = inputRDD.map(lambda element: element*element)
```

51.1.3 FlatMap transformation

The flatMap transformation is used to create a new RDD by applying a function `f` on each element of the input RDD. The new RDD contains a list of elements obtained by applying `f` on each element `x` of the input RDD; in other words, the function `f` applied on an element `x` of the input RDD returns a list of values `[y]` (e.g., `[y]= f(x)`). `[y]` can be the empty list.

The final result is the concatenation of the list of values obtained by applying `f` over all the elements of the input RDD (i.e., the final RDD contains the concatenation of the lists obtained by applying `f` over all the elements of the input RDD).

Notice that

- duplicates are not removed
- the data type of `y` can be different from the data type of `x`

The flatMap transformation is based on the `flatMap(f)` method of the `RDD` class A function `f` implementing the transformation is passed to the `flatMap` method, where `f` contains the code that is applied on each element of the input RDD and returns a list of elements which will be included in the new returned RDD: for each element of the input RDD a list of new elements is returned by `f`. The returned list can be empty.

Example

1. Create an RDD from a textual file containing a generic text (each line of the input file can contain many words).
2. Create a new RDD containing the list of words, with repetitions, occurring in the input textual document. In other words, each element of the returned RDD is one of the words occurring in the input textual file, and the words occurring multiple times in the input file appear multiple times, as distinct elements, also in the returned RDD.

```
# Read the content of the input textual file
inputRDD = sc.textFile("document.txt")

# Compute/identify the list of words occurring in document.txt
listOfWordsRDD = inputRDD.flatMap(lambda l: l.split(' '))
```

In this case the lambda function returns a "list" of values for each input element. However, notice that the new RDD (i.e., `listOfWordsRDD`) contains the "concatenation" of the lists obtained by applying the lambda function over all the elements of `inputRDD`: the new RDD is an RDD of strings and not an RDD of lists of strings.

51.1.4 Distinct information

The distinct transformation is applied on one single RDD and returns a new RDD containing the list of distinct elements (values) of the input RDD.

The distinct transformation is based on the `distinct()` method of the `RDD` class, and no functions are needed in this case.

A shuffle operation is executed for computing the result of the distinct transformation, so that data from different input partitions gets be compared to remove duplicates. The shuffle operation is used to repartition the input data: all the repetitions of the same input element

are associated with the same output partition (in which one single copy of the element is stored), and a hash function assigns each input element to one of the new partitions.

Example 1

1. Create an RDD from a textual file containing the names of a list of users (each line of the input file contains one name);
2. Create a new RDD containing the list of distinct names occurring in the input file. The type of the new RDD is the same of the input RDD.

```
# Read the content of a textual input file
inputRDD = sc.textFile("names.txt")

# Select the distinct names occurring in inputRDD
distinctNamesRDD = inputRDD.distinct()
```

Example 2

1. Create an RDD of integers containing the values `[1, 2, 3, 3]`;
2. Create a new RDD containing only the distinct values appearing in the input RDD.

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList)

# Compute the set of distinct words occurring in inputRDD
distinctIntRDD = inputRDD.distinct()
```

51.1.5 SortBy transformation

The sortBy transformation is applied on one RDD and returns a new RDD containing the same content of the input RDD sorted in ascending order.

The sortBy transformation is based on the `sortBy(keyfunc)` method of the `RDD` class: each element of the input RDD is initially mapped to a new value by applying the specified function `keyfunc`, and then the input elements are sorted by considering the values returned by the invocation of `keyfunc` on the input values.

The `sortBy(keyfunc, ascending)` method of the `RDD` class allows specifying if the values in the returned RDD are sorted in ascending or descending order by using the Boolean parameter `ascending`

- `ascending` set to True means ascending order
- `ascending` set to False means descending order

Example 1

1. Create an RDD from a textual file containing the names of a list of users (each line of the input file contains one name);
2. Create a new RDD containing the list of users sorted by name (based on the alphabetic order).

```
# Read the content of a textual input file
inputRDD = sc.textFile("names.txt")

# Sort the content of the input RDD by name.
# Store the sorted result in a new RDD
sortedNamesRDD = inputRDD.sortBy(lambda name: name)
```

Notice that each input element of the lambda expression is a string. The goal is sorting the input names (strings) in alphabetic order, which is the standard sort order for strings. For this reason the lambda function returns the input strings without modifying them.

Example 2

1. Create an RDD from a textual file containing the names of a list of users (each line of the input file contains one name);
2. Create a new RDD containing the list of users sorted by the length of their name (i.e., the sort order is based on `len(name)`).

```
# Read the content of a textual input file
inputRDD = sc.textFile("names.txt")

# Sort the content of the input RDD by name.
# Store the sorted result in a new RDD
sortedNamesLenRDD = inputRDD.sortBy(lambda name: len(name))
```

In this case, each input element is a string but we are interested in sorting the input names (strings) by length (integer), which is not the standard sort order for strings. For this reason the lambda function returns the length of each input string, and the sort operation is performed on the returned integer values (the lengths of the input names).

51.1.6 Sample transformation

The sample transformation is applied on one single RDD and returns a new RDD containing a random sample of the elements (values) of the input RDD.

The sample transformation is based on the `sample(withReplacement, fraction)` method of `RDD` class:

- `withReplacement` specifies if the random sample is with replacement (True) or not (False);
- `fraction` specifies the expected size of the sample as a fraction of the input RDD's size (values in the range $[0, 1]$).

Example 1

1. Create an RDD from a textual file containing a set of sentences (each line of the file contains one sentence);
2. Create a new RDD containing a random sample of sentences, using the "without replacement" strategy and setting fraction to 0.2 (i.e., 20).

```
# Read the content of a textual input file
inputRDD = sc.textFile("sentences.txt")

# Create a random sample of sentences
randomSentencesRDD = inputRDD.sample(False,0.2)
```

Example 2

1. Create an RDD of integers containing the values `[1, 2, 3, 3]`;
2. Create a new RDD containing a random sample of the input values, using the "replacement" strategy and setting fraction to 0.2 (i.e., 20).

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList)

# Create a sample of the inputRDD
randomSentencesRDD = inputRDD.sample(True,0.2)
```

51.2 Set transformations

Spark provides also a set of transformations that operate on two input RDDs and return a new RDD. Some of them implement standard set transformations:

- Union
- Intersection
- Subtract
- Cartesian

All these transformations have

- Two input RDDs: one is the RDD on which the method is invoked, while the other RDD is passed as parameter to the method
- One output RDD

All the involved RDDs have the same data type when union, intersection, or subtract are used, instead mixed data types can be used with the cartesian transformation.

51.2.1 Union transformation

The union transformation is based on the `union(other)` method of the `RDD` class: `other` is the second RDD to use, and the method returns a new RDD containing the union (with duplicates) of the elements of the two input RDDs.

> ⚠ Warning
>
> **Duplicates elements are not removed**. This choice is related to optimization reasons: removing duplicates means having a global view of the whole content of the two input RDDs, but, since each RDD is split in partitions that are stored in different nodes of the cluster, the contents of all partitions should be shared to remove duplicates, and that's a computationally costly operation.
> The shuffle operation is not needed in this case.

If removing duplicates is needed after performing the union transformation, apply the `distinct()` transformation on the output of the `union()` transformation, but pay attention that `distinct()` is a computational costly operation (it is associated with a shuffle operation). Use `distinct()` if and only if duplicate removal is indispensable for the application.

51.2.2 Intersection transformation

The intersection transformation is based on the `intersection(other)` method of the `RDD` class: `other` is the second RDD to use, and the method returns a new RDD containing the elements (without duplicates) occurring in both input RDDs.

Duplicates are removed: a shuffle operation is executed for computing the result of intersection, since elements from different input partitions must be compared to find common elements.

51.2.3 Subtract transformation

The subtract transformation is based on the `subtract(other)` method of the `RDD` class: `other` is the second RDD to use, and the result contains the elements appearing only in the RDD on which the subtract method is invoked. Notice that in this transformation the two input RDDs play different roles.

Duplicates are not removed, but a shuffle operation is executed for computing the result of subtract, since elements from different input partitions must be compared.

51.2.4 Cartesian transformation

The cartesian transformation is based on the `cartesian(other)` method of the `RDD` class: `other` is the second RDD to use, the data types of the objects of the two input RDDs can be different, and the returned RDD is an RDD of pairs (tuples) containing all the combinations composed of one element of the first input RDD and one element of the second input RDD (see later what an RDD of pairs is).

In this transformation a large amount of data is sent on the network: elements from different input partitions must be combined to compute the returned pairs, but the elements of the two input RDDs are stored in different partitions, which could be even in different servers.

51.2.5 Examples of set transformations

Example 1

1. Create two RDDs of integers
   - `inputRDD1` contains the values `[1, 2, 2, 3, 3]`
   - `inputRDD2` contains the values `[3, 4, 5]`
2. Create four new RDDs
   - `outputUnionRDD` contains the union of `inputRDD1` and `inputRDD2`

- `outputIntersectionRDD` contains the intersection of `inputRDD1` and `inputRDD2`
- `outputSubtractRDD` contains the result of `inputRDD1` `inputRDD2`
- `outputCartesianRDD` contains the cartesian product of `inputRDD1` and `inputRDD2`

```
# Create two RDD of integers
inputList1 = [1, 2, 2, 3, 3]
inputRDD1 = sc.parallelize(inputList1)

inputList2 = [3, 4, 5]
inputRDD2 = sc.parallelize(inputList2)

# Create four new RDDs by using union, intersection, subtract, and cartesian
outputUnionRDD = inputRDD1.union(inputRDD2)

outputIntersectionRDD = inputRDD1.intersection(inputRDD2)

outputSubtractRDD = inputRDD1.subtract(inputRDD2)

outputCartesianRDD = inputRDD1.cartesian(inputRDD2)
```

`outputCartesianRDD`	Each element of the returned RDD is a pair (tuple) of integer elements.

Example 2

1. Create two RDDs
   - `inputRDD1` contains the Integer values `[1, 2, 3]`
   - `inputRDD2` contains the String values `["A", "B"]`
2. Create a new RDD containing the cartesian product of `inputRDD1` and `inputRDD2`

```python
# Create an RDD of Integers and an RDD of Strings
inputList1 = [1, 2, 3]
inputRDD1 = sc.parallelize(inputList1)

inputList2 = ["A", "B"]
inputRDD2 = sc.parallelize(inputList2)

# Compute the cartesian product
outputCartesianRDD = inputRDD1.cartesian(inputRDD2)
```

`outputCartesianRDD`	Each element of the returned RDD is a pair (tuple) of integer elements.

51.3 Summary

51.3.1 Single input RDD transformations

All the examples reported in the following tables are applied on an RDD of integers containing the following elements (i.e., values): `[1,2,3,3]`.

Transformation	Purpose	Example function	Example result
`filter(f)`	Return an RDD consisting only of the elements of the input RDD that pass the condition passed to `filter()`. The input RDD and the new RDD have the same data type.	`filter(lambda x: x != 1)`	`[2,3,3]`
`map(f)`	Apply a function to each element in the RDD and return an RDD of the result. The applied function return one element for each element of the input RDD. The input RDD and the new RDD can have a different data type.	`map(lambda x: x+1)` For each input element `x`, the element with value `x+1` is included in the new RDD	`[2,3,4,4]`

Transformation	Purpose	Example function	Example result
`flatMap(f)`	Apply a function to each element in the RDD and return an RDD of the result. The applied function return a set of elements (from 0 to many) for each element of the input RDD. The input RDD and the new RDD can have a different data type.	`flatMap(lambda x: list(range(x,4))` For each input element `x`, the set of elements with values from `x` to 3 are returned	`[1,2,3,2,3,3,3]`
`distinct()`	Remove duplicates.	`distinct()`	`[1, 2, 3]`
`sortBy(keyfunc)`	Return a new RDD containing the same values of the input RDD sorted in ascending order.	`sortBy(lambda v: v)` Sort the input integer values in ascending order by using the standard integer sort order	`[1, 2, 3, 3]`
`sample(withReplacement, fraction)`	Sample the content of the input RDD, with or without replacement and return the selected sample. The input RDD and the new RDD have the same data type.	`sample(True, 0.2)`	Non deterministic

Transformation	Purpose	Example function	Example result
`filter(f)`	Return an RDD consisting only of the elements of the "input"" RDD that pass the condition passed to filter(). The "input" RDD and the new RDD have the same data type.	`filter(lambda x: x != 1)`	`[2,3,3]`
`map(f)`	Apply a function to each element in the RDD and return an RDD of the result. The applied function return one element for each element of the input RDD. The input RDD and the new RDD can have a different data type.	`map(lambda x: x+1)` For each input element `x`, the element with value `x+1` is included in the new RDD	`[2,3,4,4]`

Transformation	Purpose	Example function	Example result
`flatMap(f)`	Apply a function to each element in the RDD and return an RDD of the result. The applied function return a set of elements (from 0 to many) for each element of the input RDD. The input RDD and the new RDD can have a different data type.	`flatMap(lambda x: list(range(x,4))` For each input element `x`, the set of elements with values from `x` to 3 are returned	`[1,2,3,2,3,3,3]`
distinct()	Remove duplicates.	distinct()	[1, 2, 3]
sortBy(keyfunc)	Return a new RDD containing the same values of the input RDD sorted in ascending order	sortBy(lambda v: v) Sort the input integer values in ascending order by using the standard integer sort order	[1, 2, 3, 3]
sample(withReplacement, fraction)	Sample the content of the input RDD, with or without replacement and return the selected sample. The input RDD and the new RDD have the same data type.	sample(True, 0.2)	Non deterministic