

Table of contents

1 About this Book	2
2 Introduction to Big data	3
2.1 Who generates Big Data	3
Example of Big Data at work	3
2.2 The five Vs of Big Data	3
2.3 The bottleneck and the solution	4
Bottleneck	4
Solution	4
3 Big data architectures	5
3.1 Definition of Big data architecture	5
3.2 Lambda architecture	5
Definitions	5
Requirements	6
Queries	6
Basic structure	7
Detailed view	8
4 HDFS and Hadoop: command line commands	10
4.1 HDFS	10
User folder	10
Command line	10
4.2 Hadoop	12
Example	12
5 Introduction to Hadoop and MapReduce	13
5.1 Motivations of Hadoop and Big data frameworks	13
Data volumes	13
Failures	13
Network bandwidth	14
5.2 Architectures	15
Single node architecture	15
Cluster architecture	16
Scalability	17
Cluster computing challenges	18
Typical Big data problem	19
5.3 Apache Hadoop	19
Hadoop vs. HPC	20
Main components	20
Distributed Big data processing infrastructure	21

HDFS	22
5.4 MapReduce: introduction	23
Word count	23
MapReduce approach key ideas	25
Hadoop and MapReduce usage scope	26
5.5 The MapReduce programming paradigm	26
What can MapReduce do	26
Building blocks: Map and Reduce	27
Solving the word count problem	27
MapReduce Phases	29
Data structures	30
Pseudocode of word count solution using MapReduce	30
6 How to write MapReduce programs in Hadoop	31
6.1 The components: summary	31
Driver (instance)	32
Mapper (instance)	32
Reducer (instance)	32
6.2 Hadoop implementation of the MapReduce phases	32
Driver class	34
Mapper class	34
Reducer class	35
Data Types	35
Input: <code>InputFormat</code>	36
Output: <code>OutputFormat</code>	38
6.3 Structure of a MapReduce program in Hadoop	39
Driver	39
Mapper	41
Reducer	42
Example of a MapReduce program in Hadoop: Word Count	43
6.4 Combiner	48
Combiner (instance)	49
Combiner class	49
Example: adding the Combiner to the Word Count problem	50
Final thoughts	51
6.5 Personalized Data Types	51
Example	52
Complex keys	53
6.6 Sharing parameters among Driver, Mappers, and Reducers	54
How to use these parameters	54
6.7 Counters	54
User-defined counters	55
6.8 Map-only job	55
Implementation of a Map-only job	56
6.9 In-Mapper combiner	56
Setup method	56
Cleanup method	56
In-Mapper combiner: Word count pseudocode	57

6.10 Maven project	57
Structure	57
How to run the project	57
How to create a .jar file from the project	58
How to run the .jar in the BigData@Polito cluster	58
7 MapReduce patterns - 1	59
7.1 Summarization Patterns	59
Numerical summarizations	59
Inverted index summarization	60
Counting with counters	61
7.2 Filtering patterns	63
Filtering	63
Top K	64
Distinct	65
8 MapReduce and Hadoop Advanced Topics	67
8.1 Multiple inputs	67
8.2 Multiple outputs	68
Driver	68
Map-only	69
Example: map-only	69
8.3 Distributed cache	69
Example: distributed cache	71
9 MapReduce patterns - 2	72
9.1 Data organization patterns	72
Binning	72
Shuffling	73
9.2 Metapatterns	74
Job Chaining	74
9.3 Join patterns	75
Reduce side natural join	75
Example: join pattern	76
Example	76
Map side natural join	77
Other join patterns	78
10 Relational Algebra Operations and MapReduce	79
10.1 Selection	79
10.2 Projection	80
10.3 Union	81
10.4 Intersection	82
10.5 Difference	83
10.6 Join	84
10.7 Aggregations and Group by	84

11 How to submit/execute a Spark application	85
11.1 Spark submit	85
Options of spark-submit: --master	85
Options of spark-submit: --deploy-mode	85
Setting the executors	87
Setting the drivers	87
Execution examples	88
12 Introduction to Spark	89
12.1 Motivations	89
MapReduce and Spark iterative jobs and data I/O	89
Resilient distributed data sets (RDDs)	91
MapReduce vs Spark	92
12.2 Main components	93
12.3 Basic concepts	94
Resilient Distributed Data sets (RDDs)	94
12.4 Spark Programs	96
Supported languages	96
Structure of Spark programs	96
Local execution of Spark	98
12.5 Spark program examples	99
Count line program	99
Word Count program	100
13 RDD based programming	102
13.1 Spark Context	102
13.2 RDD basics	102
13.3 RDD: create and save	103
Create RDDs from files	103
Create RDDs from a local Python collection	104
Save RDDs	105
Retrieve the content of RDDs and store it local Python variables	106
13.4 Transformations and Actions	106
Transformations	106
Actions	107
13.5 Passing functions to Transformations and Actions	109
Example based on the filter transformation	109
13.6 Basic Transformations	111
Single input RDD transformations	111
Set transformations	117
Summary	120
13.7 Basic Actions	126
Collect action	126
Count action	127
CountByValue action	128
Take action	128
First action	129
Top action	129

TakeOrdered action	130
TakeSample action	131
Reduce	132
Fold action	133
Aggregate action	134
Summary	137
14 RDDs and key-value pairs	140
14.1 Creating RDDs of key-value pairs	140
14.2 RDDs of key-value pairs by using the Map transformation	140
14.3 RDDs of key-value pairs by using the flatMap transformation	141
14.4 RDDs of key-value pairs by using parallelize	142
14.5 Transformations on RDDs of key-value pairs	143
ReduceByKey transformation	143
FoldByKey transformation	144
CombineByKey transformation	145
GroupByKey transformation	148
MapValues transformation	149
FlatMapValues transformation	150
Keys transformation	151
Values transformation	152
SortByKey transformation	153
Summary	154
14.6 RDD-based programming	157
Transformations on two RDDs of key-value pairs	157
SubtractByKey transformation	157
Join transformation	158
CoGroup transformation	159
Summary	160
14.7 Actions on RDDs of key-value pairs	163
CountByKey action	163
CollectAsMap action	164
Lookup action	165
Summary	165
15 RDD of numbers	168
15.1 Summary	168
16 Cache, Accumulators, Broadcast Variables	171
16.1 Persistence and Cache	171
Remove data from cache	172
16.2 Accumulators	173
How to use accumulators	173
Personalized accumulators	175
16.3 Broadcast variables	175
16.4 RDDs and Partitions	178
partitionBy(numPartitions)	179
Default partitioning behavior of the main transformations	182

16.5 Broadcast join	184
17 Introduction to PageRank	186
17.1 PageRank formulations	187
Simple recursive formulation	187
Random jumps formulation	187
18 Spark SQL and DataFrames	190
18.1 Spark SQL	190
Spark SQL vs Spark RDD APIs	190
DataFrames	190
18.2 DataFrames	191
Creating DataFrames from csv files	191
Creating DataFrames from JSON files	192
Creating DataFrames from other data sources	194
Creating DataFrames from RDDs or Python lists	194
From DataFrame to RDD	195
18.3 Operations on DataFrames	196
Show method	196
PrintSchema method	197
Count method	197
Distinct method	197
Select method	198
SelectExpr method	199
Filter method	201
Where method	202
Join	202
Aggregate functions	205
groupBy and aggregate functions	206
Sort method	208
18.4 DataFrames and the SQL language	208
18.5 Save DataFrames	211
18.6 UDFs: User Defines Functions	213
18.7 Other notes	214
Data warehouse methods: cube and rollup	214
Set methods	215
Broadcast join	216
Execution plan	216
19 Spark MLlib	217
19.1 Data types	217
Local vectors	217
Local matrices	218
19.2 Main concepts	219
Transformer	220
Estimator	220
Pipeline	220
Parameters	221

19.3 Data preprocessing	221
Extracting, transformings, and selecting features	221
19.4 Feature transformations	221
VectorAssembler	222
Data Normalization	223
Categorical columns	226
20 Classification algorithms	234
20.1 Structured data classification	235
Example of logistic regression and structured data	235
Pipelines	239
Decision trees and structured data	242
20.2 Categorical class labels	245
StringIndexer and IndexToString	245
20.3 Textual data management and classification	249
20.4 Performance evaluation	254
20.5 Hyperparameter tuning	257
20.6 Sparse labeled data	259
21 Clustering algorithms	261
21.1 Main steps	262
21.2 K-means clustering algorithm	262
22 Regression algorithms	265
22.1 Linear regression	265
LR with structured data	265
LR with textual data	268
Parameter setting	268
23 Itemset and Association rule mining	269
23.1 The FP-Growth algorithm and Association rule mining	269
Steps for itemset and association rule mining in Spark	270
Input	270
24 Graph analytics in Spark	274
24.1 Introduction	274
24.2 Spark GraphX and GraphFrames	277
24.3 Building and querying graphs with GraphFrames	279
Building a Graph	279
Directed vs undirected edges	283
Cache graphs	283
Querying the graph	283
Motif finding	291
24.4 Basic statistics	300
degrees	300
inDegrees	300
outDegrees	300

25 Graph Analytics in Spark	305
25.1 Algorithms over graphs	305
Checkpoint directory	305
Breadth first search	305
Shortest path	314
Connected components	321
Strongly connected components	328
Label propagation	333
PageRank	337
Custom graph algorithms	343
26 Streaming data analytics frameworks	345
26.1 Introduction	345
What is streaming processing?	345
Stream processing frameworks for big streaming data analytics	346
Input data processing and result guarantees	347
26.2 Spark Streaming	347
What is Spark Streaming	347
Spark discretized stream processing	348
Word count using DStreams	349
Fault-tolerance	350
26.3 Spark streaming programs	350
Basic structure of a Spark streaming program	350
Spark streaming context	350
Input streams	351
Transformations	352
Start and run the computations	353
26.4 Windowed computation	355
Parameters	355
Basic window transformations	356
Checkpoint	358
26.5 Stateful computation	361
<code>updateStateByKey</code> transformation	361
26.6 Transform transformation	363
27 Spark structured streaming	365
27.1 What is Spark structured streaming?	365
Input data model	365
Queries	366
27.2 Key concepts	367
Input sources	368
Transformations	368
Outputs	369
Query run/execution	370
Triggers	370
Spark structured streaming examples	371
27.3 Event time and window operations	374
Event time and window operations: example 1	374

Late data	377
Event time and window operations: example 2	379
Watermarking	382
27.4 Join operations	383
28 Streaming data analytics frameworks	384
28.1 Stream processing frameworks for (big) streaming data analytics	384
Comparison among state of the art streaming frameworks	384
28.2 Introduction to Apache Storm	385
Data processing	385
Features of Storm	385
28.3 Storm core concepts	385
Main concepts	386
Data model	386
Spout	386
Bolt	387
Topology	387

1 About this Book

This Quarto Book contains notes collected during the lectures of “Distributed architectures for big data processing and analytics” (**DABDPA**) course, hold at the Master Degree in Data Science and Engineering (2022-2023) of Politecnico di Torino.

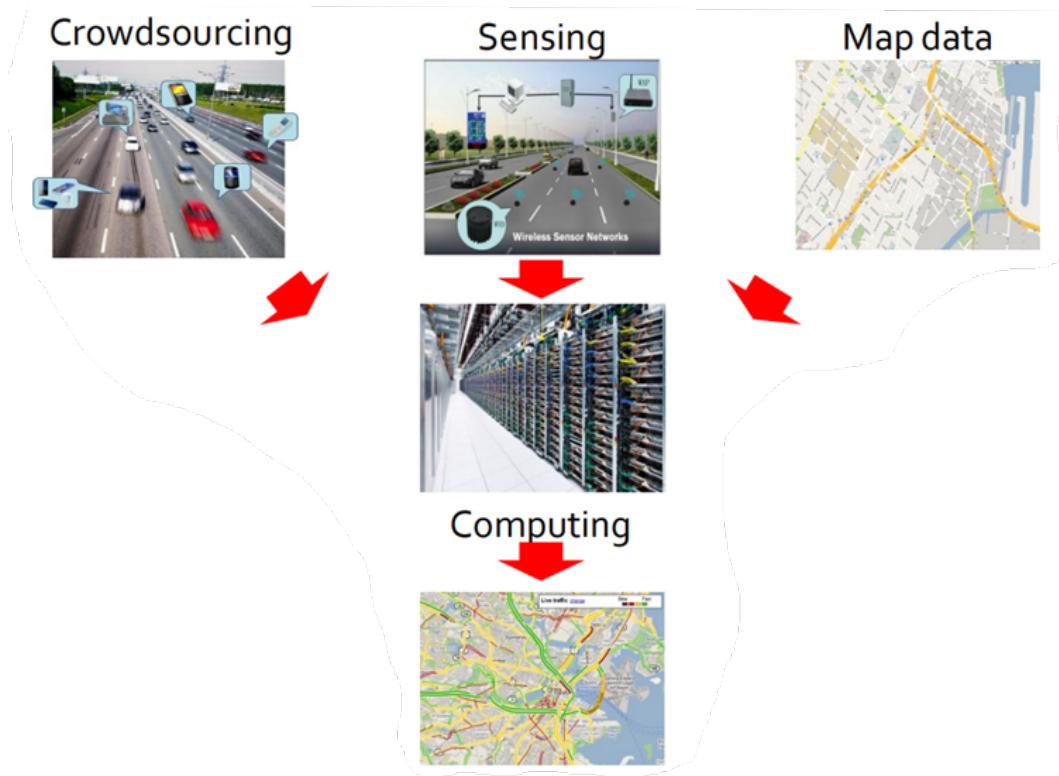
2 Introduction to Big data

2.1 Who generates 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

Example of Big Data at work

Figure 2.1: Bigdata example



2.2 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

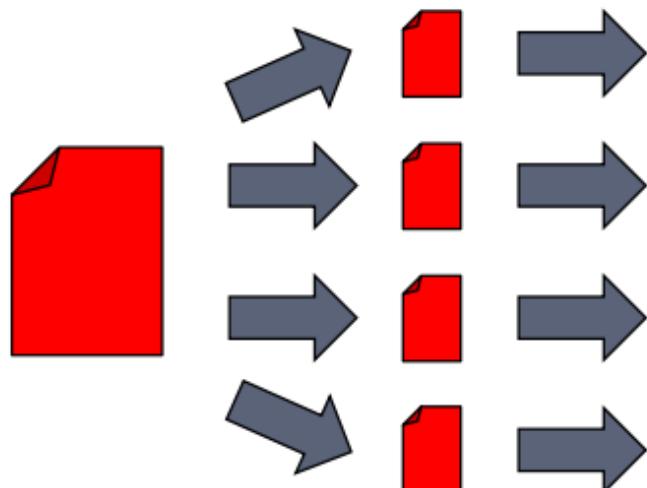
2.3 The bottleneck and the solution

Bottleneck

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

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

3 Big data architectures

3.1 Definition of Big data architecture

💡 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

3.2 Lambda architecture

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

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.

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

Queries

query = function(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)

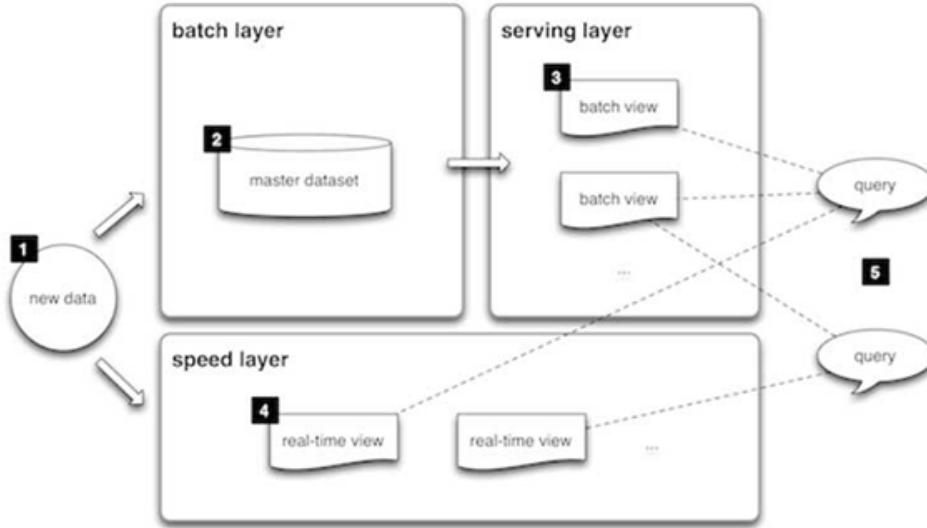
It is based on two data paths:

- 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

Basic structure

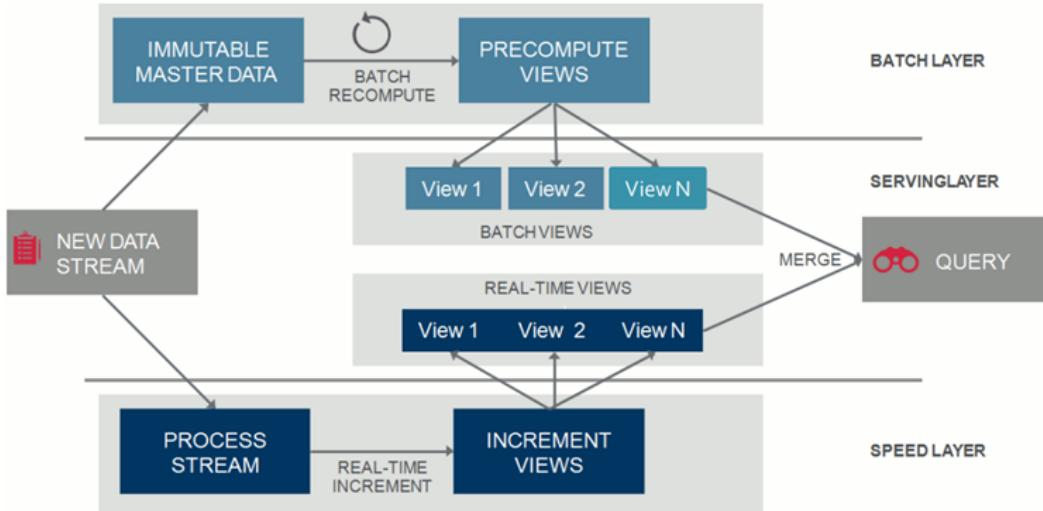
Figure 3.1: General Lambda architecture



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).

Detailed view

Figure 3.2: More detailed of Lambda architecture

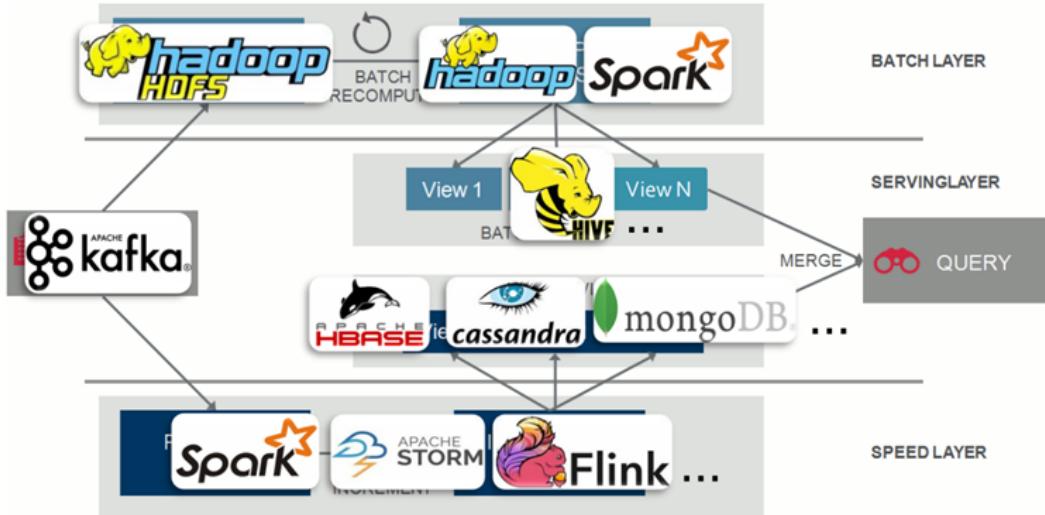


Structure similar to the one described before

0. Data stream
1. Batch layer
 - immutable data
 - precompute views
2. Real-time layer
 - process stream
 - increment views
3. Serving layer

Possible instances

Figure 3.3: 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.

4 HDFS and Hadoop: command line commands

4.1 HDFS

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).

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`

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.

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

Content of a file

To show the content of a file in the HDFS file system, use `hdfs dfs -cat file_name`

i Example

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

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`

i 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.

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`

i 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/`.

Delete a file

To delete a file from the HDFS file system, use `hdfs dfs -rm HDFS_path`

i Example

The command `hdfs dfs -rm /user/garza/document.txt` deletes from HDFS the file `/user/garza/document.txt`

Other commands

There are many other linux-like commands, for example

- `rmdir`
- `du`

- `tail`

See the [HDFS commands](#) guide for a complete list.

4.2 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

Example

The following command executes/submits a MapReduce application

```
1 hadoop jar MyApplication.jar \
2 it.polito.bigdata.hadoop.DriverMyApplication \
3 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/`)

5 Introduction to Hadoop and MapReduce

5.1 Motivations of Hadoop and Big data frameworks

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: $10 \text{ billion} \times 20\text{KBs} = 200\text{TB}$
- 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

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)

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 servers, instead of huge amount of data.

5.2 Architectures

Single node architecture

Figure 5.1: Single node architecture

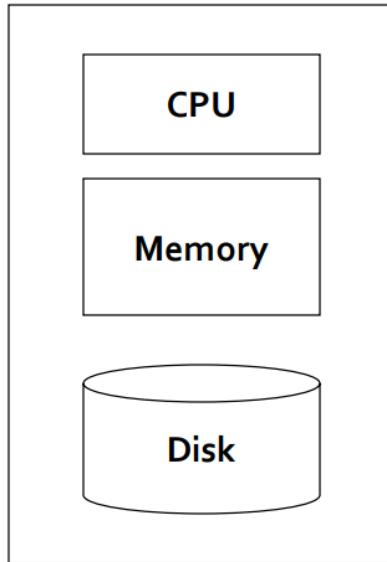
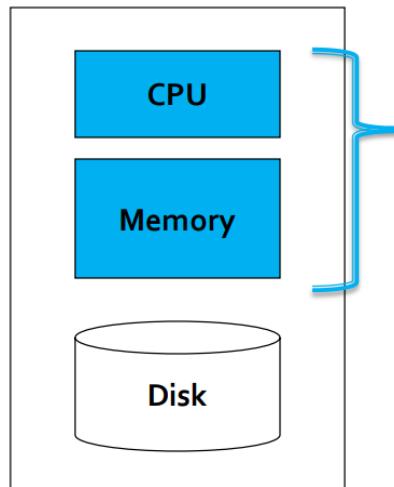
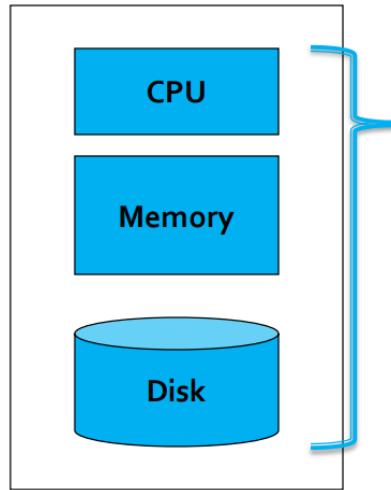


Figure 5.2: Single node architecture: Machine Learning and Statistics



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

Figure 5.3: Single node architecture: “Classical” data mining”



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

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:

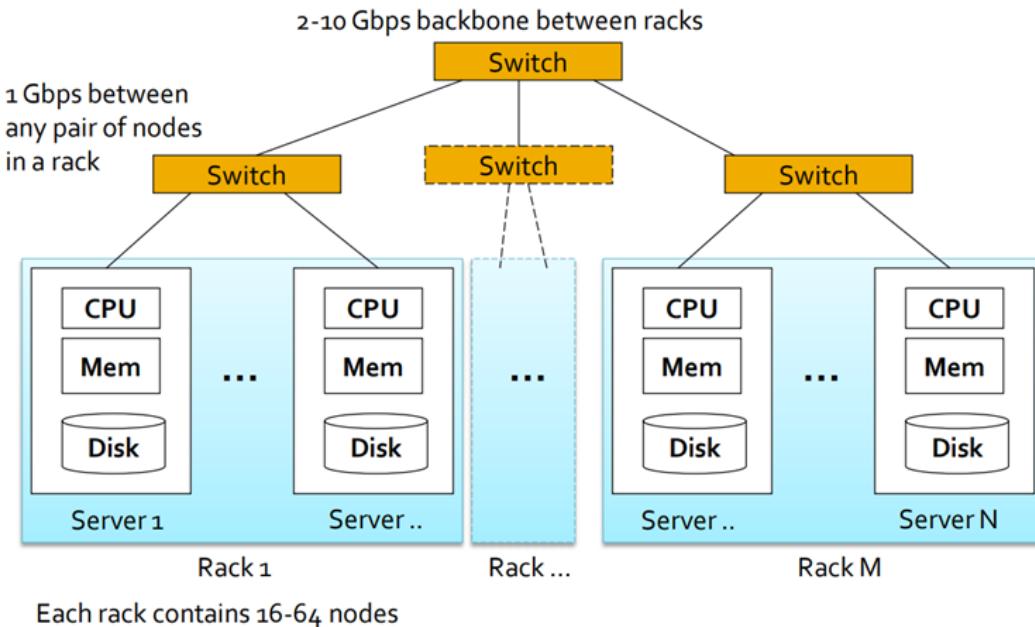
- Computation is distributed across servers
- Data are stored/distributed across servers

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

Commodity cluster architecture

Figure 5.4: 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

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)

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.

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
4. Network bottleneck

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.

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.

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)

5.3 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).

Hadoop history

Table 5.1: 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!

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

Main components

Core components of Hadoop:

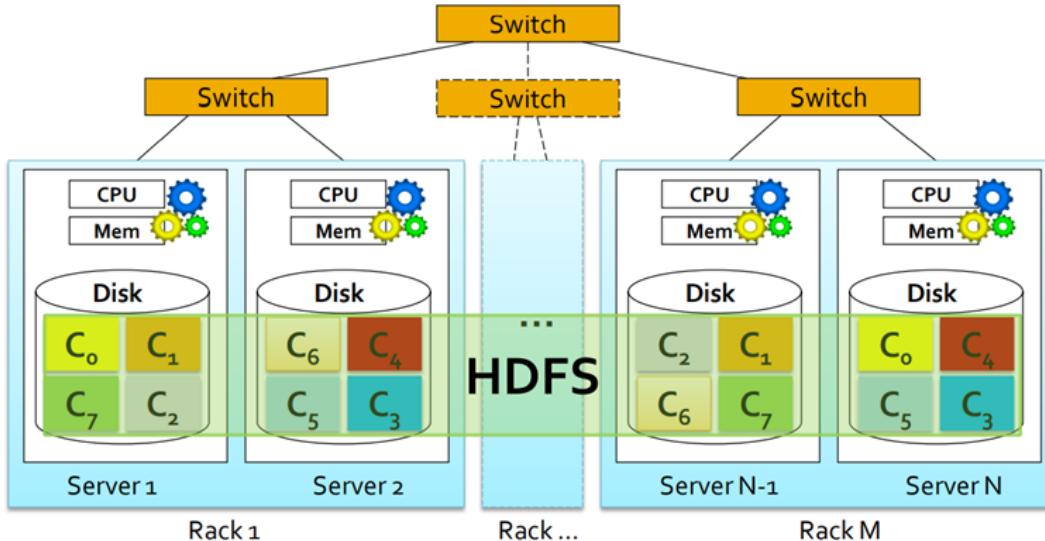
1. Distributed Big Data Processing Infrastructure based on the MapReduce programming paradigm
 - Provides a high-level abstraction view: programmers do not need to care about task scheduling and synchronization
 - Fault-tolerant: node and task failures are automatically managed by the Hadoop system
2. 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 5.5: Hadoop main components



Example with number of replicas per chunk = 2

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 HDFS can sustain with no repercussions is equal to (**number of copies**) – 1.

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

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.

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:

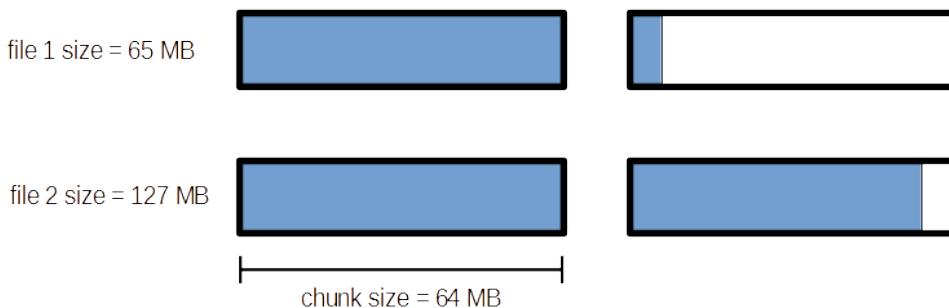
- huge files (GB to TB);
- data is rarely updated (create new files or append to existing ones);
- reads and appends are common, and random read/write operations are not performed.

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

- Each chunk 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 5.6: 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/chunk 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.

5.4 MapReduce: introduction

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

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).

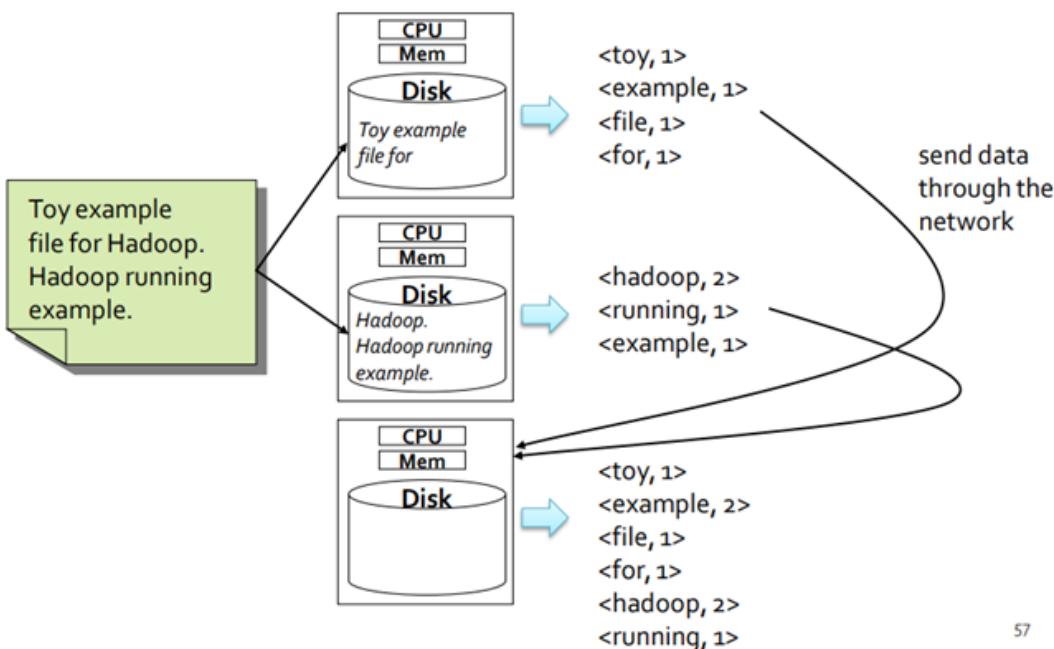
Case 2: File too large to fit in main memory

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

Figure 5.7: Word count solution



57

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: asynchronous synchronization is not needed in this phase
 - 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)

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.

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.

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

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”.

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 fit well

- Iterative problems
- Recursive problems
- Stream data processing
- Real-time processing

5.5 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.

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

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.

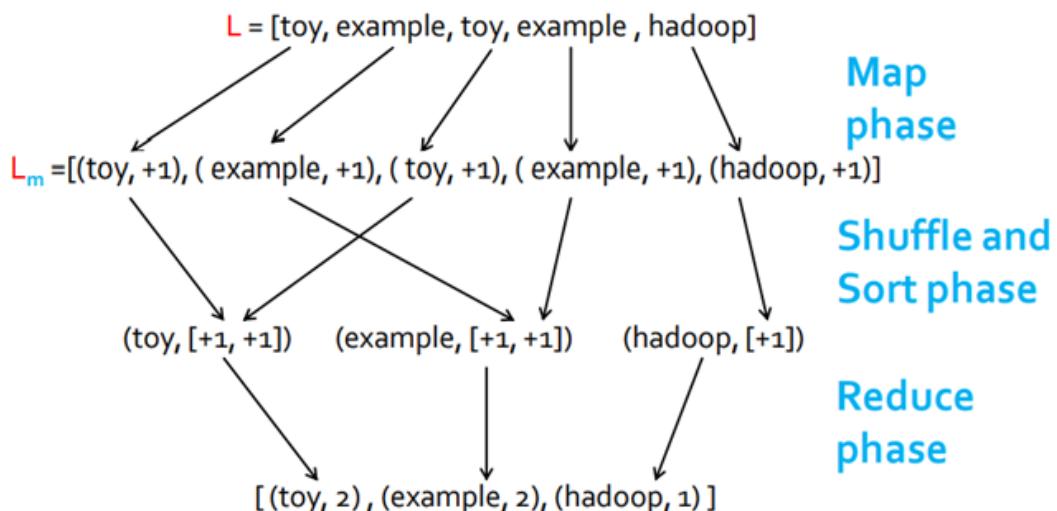
Solving the word count problem

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 5.8: Word count running example



[...] denotes a list. (k, v) denotes a key-value pair.

77

- 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).

Map

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

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

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 $[1, 1, 1, \dots]$, and, given a group G_w , the number of ones in $[1, 1, 1, \dots]$ 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.

Reduce

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

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

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

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.

MapReduce Phases

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.

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.

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.

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).

Pseudocode of word count solution using MapReduce

Map

```
1 def map(key, value):
2     """
3         :key: offset of the word in the file
4         :value: a word of the input document
5     """
6     return (value, 1)
```

Reduce

```
1 def reduce(key, values):
2     """
3         :key: a word
4         :values: a list of integers
5     """
6     occurrences = 0
7     for c in values:
8         occurrences = occurrences + c
9     return (key, occurrences)
```

6 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

6.1 The components: summary

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

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)

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.

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.

6.2 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 file 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

Figure 6.1: MapReduce data flow with a single reducer

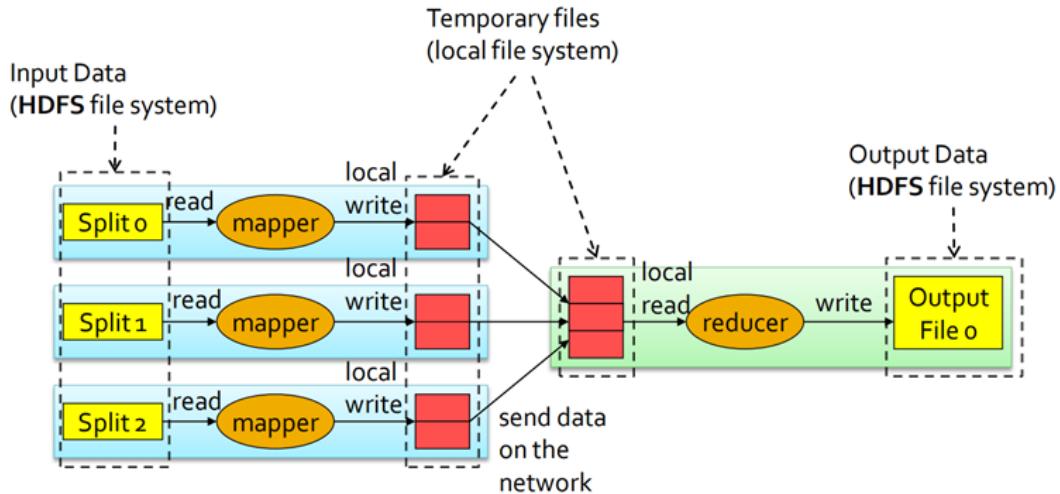
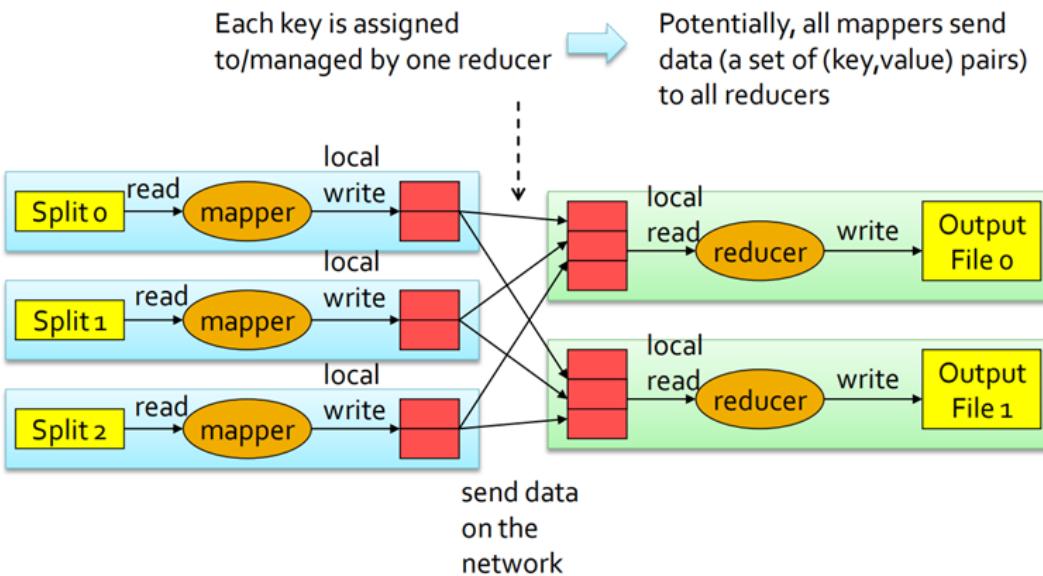


Figure 6.2: MapReduce data flow with multiple reducers



Driver class

The Driver class extends the `org.apache.hadoop.conf.Configured` class and implements the `org.apache.hadoop.util.Tool` interface¹.

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 Reducers²

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

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

²Setting 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

Reducer class

The Reducer class extends the

```
1 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

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.

Input: InputFormat

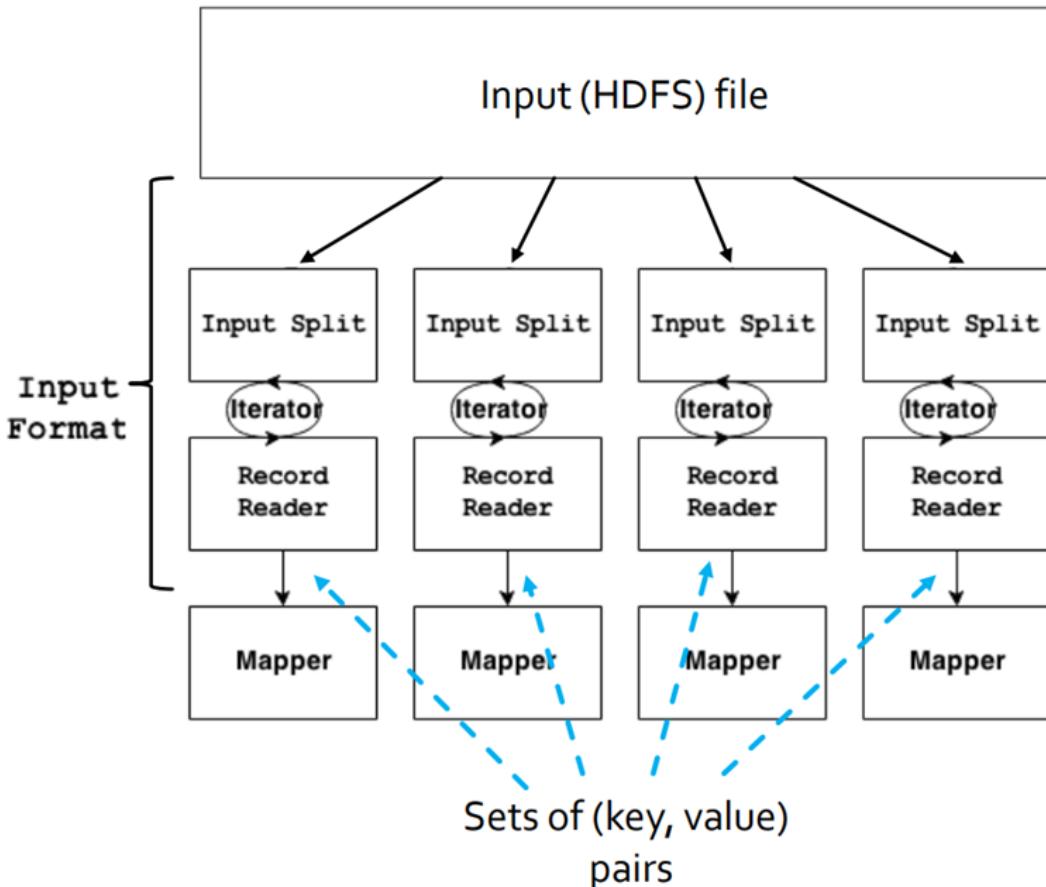
The input of the MapReduce program is an HDFS file (or an HDFS folder), but the input of the Mapper is a set of `(key, value)` pairs.

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

Figure 6.3: Getting data to 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
- ...

`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:

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

Example

Figure 6.4: Getting data to the Mapper

Input HDFS file

Toy example file for Hadoop.
 Hadoop running example.
 TextInputFormat is used to split data.



(key, value) pairs generated by using `TextInputFormat`

(0, "Toy example file for Hadoop.")
 (31, "Hadoop running example.")
 (56, "TextInputFormat is used to split data.")

`KeyValueTextInputFormat`

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

```
1 key<separator>value
```

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

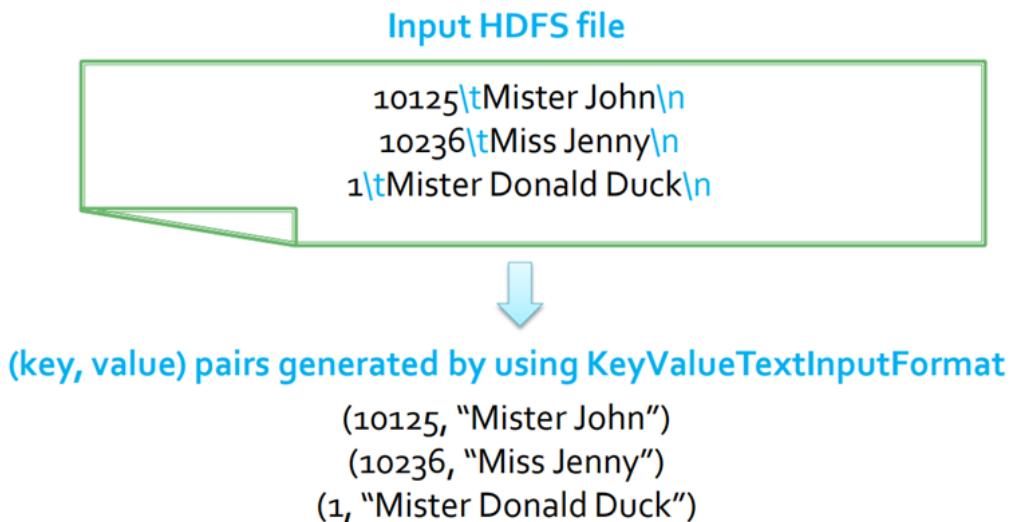
Files are broken into lines, and either linefeed or carriage-return are used to signal end of line, and each line is split into key and value parts by considering the separator symbol/character.

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 6.5: Getting data to the Mapper



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 `OutputFormat` for plain text files
- `SequenceFileOutputFormat`: an `OutputFormat` for sequential/binary files
- ...

`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"
```

6.3 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.

Driver

```
1 * Set package */ // <1>
2 package it.polito.bigdata.hadoop.mypackage;
3
4 /* Import libraries */
5 import java.io.IOException;
6
7 import org.apache.hadoop.mapreduce.Job;
8 import org.apache.hadoop.util.Tool;
9 import org.apache.hadoop.util.ToolRunner;
10 import org.apache.hadoop.conf.Configuration;
11 import org.apache.hadoop.conf.Configured;
12 import org.apache.hadoop.io.*;
13 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
14 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
15
16 /* Driver class */
17 public class MapReduceAppDriver extends Configured implements Tool {(2)
18     @Override
19     public int run(String[] args) throws Exception {
20         /* variables */
21         int exitCode;
22         //...
23
24         // Parse parameters
25         numberOfReducers = Integer.parseInt(args[0]); // Number of instances of the Reducer class
26         inputPath = new Path(args[1]); // Can be the path to a folder or to a file. If this is a
27         outputDir = new Path(args[2]); // This is always the path to a folder
28
29         // ****
30         //      JOB
31         // ****
32         // Define and configure a new job
33         Configuration conf = this.getConf(); // Create a configuration object to design in it
34         Job job = Job.getInstance(conf); // Creation of the job, that is the application instance
```

```

36   // Assign a name to the job
37   job.setJobName("My First MapReduce program"); (3)
38
39   // Set path of the input file/folder (if it is a folder, the job reads all the files in it)
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; the input types are (keys: LongWritable, values: text)
47   // KeyValueTextInputFormat = textual files; the input types are (keys: text, values: text)
48   job.setInputFormatClass(TextInputFormat.class); (4)
49
50   // Set job output format
51   job.setOutputFormatClass(TextOutputFormat.class); (5)
52
53   // *****
54   // DRIVER
55   // *****
56   // Specify the class of the Driver for this job
57   job.setJarByClass(MapReduceAppDriver.class); (6)
58
59   // *****
60   // MAPPER
61   // *****
62   // Set mapper class
63   job.setMapperClass(MyMapperClass.class); (7)
64
65   // Set map output key and value classes; these are also the key - value types of the reduce output
66   job.setMapOutputKeyClass(output key type.class); (8)
67   job.setMapOutputValueClass(output value type.class); (9)
68
69   // *****
70   // REDUCER
71   // *****
72   // Set reduce class
73   job.setReducerClass(MyReducerClass.class); (10)
74
75   // Set reduce output key and value classes
76   job.setOutputKeyClass(output key type.class); (11)
77   job.setOutputValueClass(output value type.class); (12)
78
79   // Set number of reducers
80   job.setNumReduceTasks(numberOfReducers);
81

```

```

82     // ****
83     // OTHER
84     // ****
85     // Execute the job and wait for completion
86     if (job.waitForCompletion(true)==true) // with this method the application is run
87         exitCode=0;
88     else
89         exitCode=1;
90     return exitCode;
91 } // End of the run method
92
93 /* main method of the driver class */
94 public static void main(String args[]) throws Exception { // This part of the code is always
95     /* Exploit the ToolRunner class to "configure" and run the Hadoop application */
96     int res = ToolRunner.run(
97         new Configuration(),
98         new MapReduceAppDriver(),
99         args
100    );
101    System.exit(res);
102 } // End of the main method
103 } // End of public class MapReduceAppDriver

```

(13)

- 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

Mapper

```

1  /* Set package */
2  package it.polito.bigdata.hadoop.mypackage;
3
4  /* Import libraries */
5  import java.io.IOException;
6

```

(1)

```

7 import org.apache.hadoop.mapreduce.Mapper;
8 import org.apache.hadoop.io.*;
9
10 /* Mapper Class */
11 class myMapperClass extends Mapper<          ②
12     MapperInputKeyType,                      ③
13     MapperInputValueType,                   ④
14     MapperOutputKeyType,                  ⑤
15     MapperOutputValueType                ⑥
16 >{
17     /* Implementation of the map method */
18     protected void map(
19         MapperInputKeyType key,            ⑦
20         MapperInputValueType value,      ⑧
21         Context context // This is an object containing the write method, that has to be invoked
22     ) throws IOException, InterruptedException {
23
24         /*
25             Process the input (key, value) pair and emit a set of (key,value) pairs.
26             context.write(...) is used to emit (key, value) pairs context.write(new outputkey, new
27             */
28
29         context.write(new outputkey, new outputvalue);          ⑨
30         // Notice context.write(...) has to be invoked a number of times equal to the number of
31
32         // In the mapper instance also setup and cleanup methods can be implemented, but are not
33
34     } // End of the map method
35 } // End of class myMapperClass

```

- ① mypackage
- ② myMapperClass
- ③ MapperInputKeyType
- ④ MapperInputValueType
- ⑤ MapperOutputKeyType
- ⑥ MapperOutputValueType
- ⑦ MapperInputKeyType
- ⑧ MapperInputValueType
- ⑨ outputkey and outputvalue

Reducer

```

1  /* Set package */
2  package it.polito.bigdata.hadoop.mypackage;          ①
3

```

```

4  /* Import libraries */
5  import java.io.IOException;
6  import org.apache.hadoop.mapreduce.Reducer;
7  import org.apache.hadoop.io.*;
8
9  /* Reducer Class */
10 class myReducerClass extends Reducer<②
11     ReducerInputKeyType, ③
12     ReducerInputValueType, ④
13     ReducerOutputKeyType, ⑤
14     ReducerOutputValueType ⑥
15 >{
16     /* Implementation of the reduce method */
17     protected void reduce( ⑦
18         ReducerInputKeyType key, ⑧
19         Iterable<ReducerInputValueType> values,
20         Context context
21     ) throws IOException, InterruptedException {
22
23         /*
24             Process the input (key, [list of values]) pair and emit a set of (key,value) pairs.
25             context.write(...) is used to emit (key, value) pairs context.write(new outputkey, new
26             */
27
28         context.write(new outputkey, new outputvalue); ⑨
29         // Notice context.write(...) has to be invoked a number of times equal to the number o
30         // "new" has to be always specified
31
32     } // 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

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)
```

Driver

```

1  /* Set package */
2  package it.polito.bigdata.hadoop.wordcount;
3
4  /* Import libraries */
5  import org.apache.hadoop.conf.Configuration;
6  import org.apache.hadoop.conf.Configured;
7  import org.apache.hadoop.fs.Path;
8  import org.apache.hadoop.io.IntWritable;
9  import org.apache.hadoop.io.Text;
10 import org.apache.hadoop.mapreduce.Job;
11 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
12 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
14 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
15 import org.apache.hadoop.util.Tool;
16 import org.apache.hadoop.util.ToolRunner;
17
18 /* Driver class */
19 public class WordCount extends Configured implements Tool {
20     @Override
21     public void run(String[] args) throws Exception {
22         Path inputPath;
23         Path outputDir;
24         int numberReducers;
```

```
25     int exitCode;  
26  
27     // Parse input parameters  
28     numberReducers = Integer.parseInt(args[0]);  
29     inputPath = new Path(args[1]);  
30     outputDir = new Path(args[2]);  
31  
32     // Define and configure a new job  
33     Configuration conf = this.getConf();  
34     Job job = Job.getInstance(conf);  
35  
36     // Assign a name to the job  
37     job.setJobName("WordCounter");  
38  
39     // Set path of the input file/folder (if it is a folder, the job reads all the files in  
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(numberReducers);  
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 */
81 public static void main(String args[]) throws Exception {
82
83     /* Exploit the ToolRunner class to "configure" and run the
84      Hadoop application */
85
86     intres = ToolRunner.run(
87         new Configuration(),
88         new WordCount(),
89         args
90     );
91     System.exit(res);
92 } // End of the main method
93 } // End of public class WordCount

```

Mapper

```

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.LongWritable;
8  import org.apache.hadoop.io.Text;
9  import org.apache.hadoop.mapreduce.Mapper;
10
11 /* MapperClass */
12 class WordCountMapper extends Mapper<
13     LongWritable, // Input key type
14     Text, // Input value type
15     Text, // Output key type
16     IntWritable // Output value type
17 >{
18     /* Implementation of the map method */
19     protected void map(
20         LongWritable key, // Input key type

```

```

21     Text value, // Input value type
22     Context context
23 ) throws IOException, InterruptedException {
24     // Split each sentence in words. Use whitespace(s) as delimiter
25     // The split method returns an array of strings
26     String[] words = value.toString().split("\\s+");
27
28     // Iterate over the set of words
29     for(String word : words) {
30         // Transform word case
31         String cleanedWord = word.toLowerCase();
32
33         // emit one pair (word, 1) for each input word
34         context.write(new Text(cleanedWord), new IntWritable(1));
35     }
36 } // End map method
37 } // End of class WordCountMapper

```

Reducer

```

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 /* Reducer Class */
11 class WordCountReducer 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 occurrences= 0;
24
25         // Iterate over the set of values and sum them

```

```

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

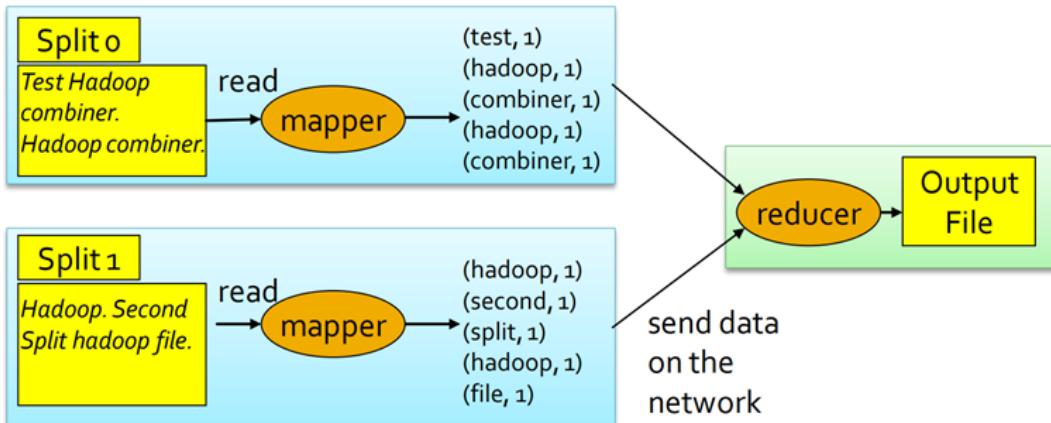
```

6.4 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”).

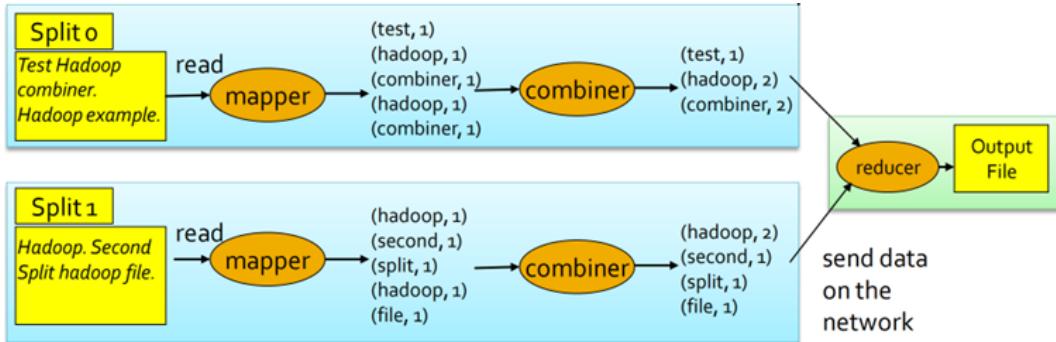
Consider the standard word count problem, and suppose that the input file is split in two input splits, hence, two Mappers are instantiated (one for each split).

Figure 6.6: 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 6.7: 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.

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

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

Notice that the output data types are the same as the output data types.

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).

Example: adding the Combiner to the Word Count problem

Consider the word count problem (see Section 6.3 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 the application.

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);
```

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 occurrences= 0;
24     // Iterate over the set of values and sum them
25     for (IntWritable value : values) {
26         occurrences = occurrences+ value.get();
27     }
28     // Emit the total number of occurrences of the current word
29     context.write(key, new IntWritable(occurrences));
30 } // End reduce method
31 } // End of class WordCountCombiner

```

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

```

1     // Set combiner class
2     job.setCombinerClass(WordCountReducer.class);
3     // "WordCountReducer.class" instead of "WordCountCombiner.class"

```

In 99% of the Hadoop applications the same class can be used to implement both combiner and reducer.

6.5 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. To properly serialize the input-output data, the following methods must be implemented

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

To properly format the output of the job (i.e., the output of the reducer) usually also the following method is “redefined”

- `public String toString()`

If also a constructor is defined, remember to define also an empty constructor, otherwise the system will raise an error at runtime.

Example

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.

```

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
8  public class SumAndCountWritable implements
9  org.apache.hadoop.io.Writable {
10     /* Private variables */
11     private float sum = 0;
12     private int count = 0;
13
14     /* Methods to get and set private variables of the class */
15     public float getSum() {
16         return sum;
17     }
18
19     public void setSum(float sumValue) {
20         sum=sumValue;
21     }
22
23     public int getCount() {
24         return count;
25     }
26
27     public void setCount(int countValue) {
28         count=countValue;
29     }
30
31     /* Methods to serialize and deserialize the contents of the
32     instances of this class */
33     @Override /* Serialize the fields of this object to out */
34     public void write(DataOutput out) throws IOException {
35         out.writeFloat(sum);
36         out.writeInt(count);
37     }
38
39     @Override /* Deserialize the fields of this object from in */
40     public void readFields(DataInput in) throws IOException {
41         sum=in.readFloat();
42         count=in.readInt(); 
```

(1)

```

43 }
44
45     /* Specify how to convert the contents of the instances of this
46     class to a String
47     * Useful to specify how to store/write the content of this class
48     * in a textual file */
49     public String toString()
50 {
51         String formattedString=
52             new String("sum="+sum+",count="+count);
53         return formattedString;
54     }
55 }
```

- ① Notice that the order of the read must be coherent with the order of the write.

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; it is used by the Combiner locally.

Example

```

1  @Override
2  public int compareTo(IntPair ip) {
3      int cmp = compare(first, ip.first);
4      if (cmp != 0) {
5          return cmp;
6      }
7      return compare(second, ip.second);
8  }
9 /**
10 * Convenience method for comparing two ints.
11 */
12 public static int compare(int a, int b) {
13     return (a < b ? -1 : (a == b ? 0 : 1));
14 }
```

- Split in groups: it is possible by implementing the `hashCode()` method; it is used by the Reducer on the networks.

6.6 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.

How to use these parameters

In the driver

1. Retrieve the configuration object

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

2. Set personalized properties

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

In the Mapper and/or Reducer

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

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

6.7 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.

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

```
1 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.

Example

In the driver, add

```
1 public static enum COUNTERS {
2     ERROR_COUNT,
3     MISSING_FIELDS_RECORD_COUNT
4 }
```

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

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

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

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

6.8 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.

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);
```

6.9 In-Mapper combiner

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

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.

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).

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 in 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})

```

6.10 Maven project

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

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.

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”

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

```
1 hadoop jar Exercise1-1.0.0.jar \
2 it.polito.bigdata.hadoop.exercise1.DriverBigData \
3 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>)

7 MapReduce patterns - 1

7.1 Summarization Patterns

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

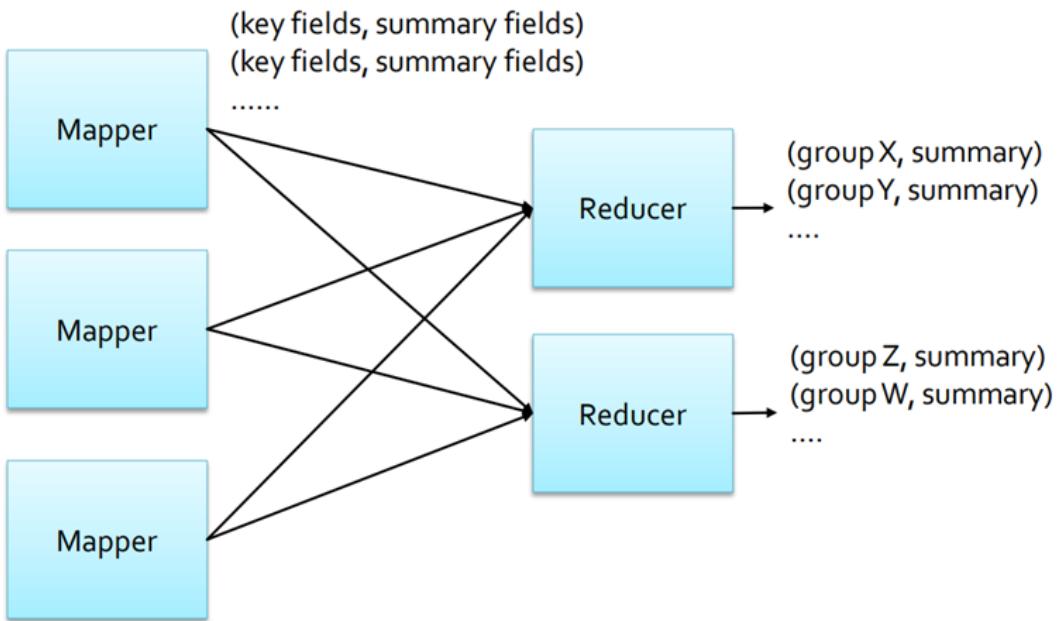
Numerical summarizations

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.

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).

Figure 7.1: Numerical summarization structure



💡 Use cases

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

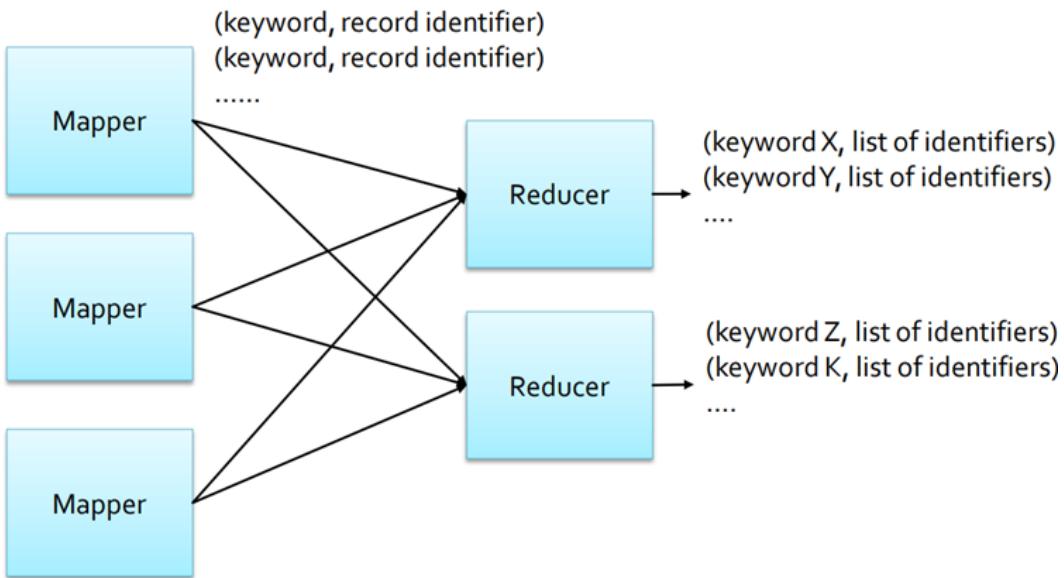
Inverted index summarization

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.

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

Figure 7.2: Numerical summarization structure



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

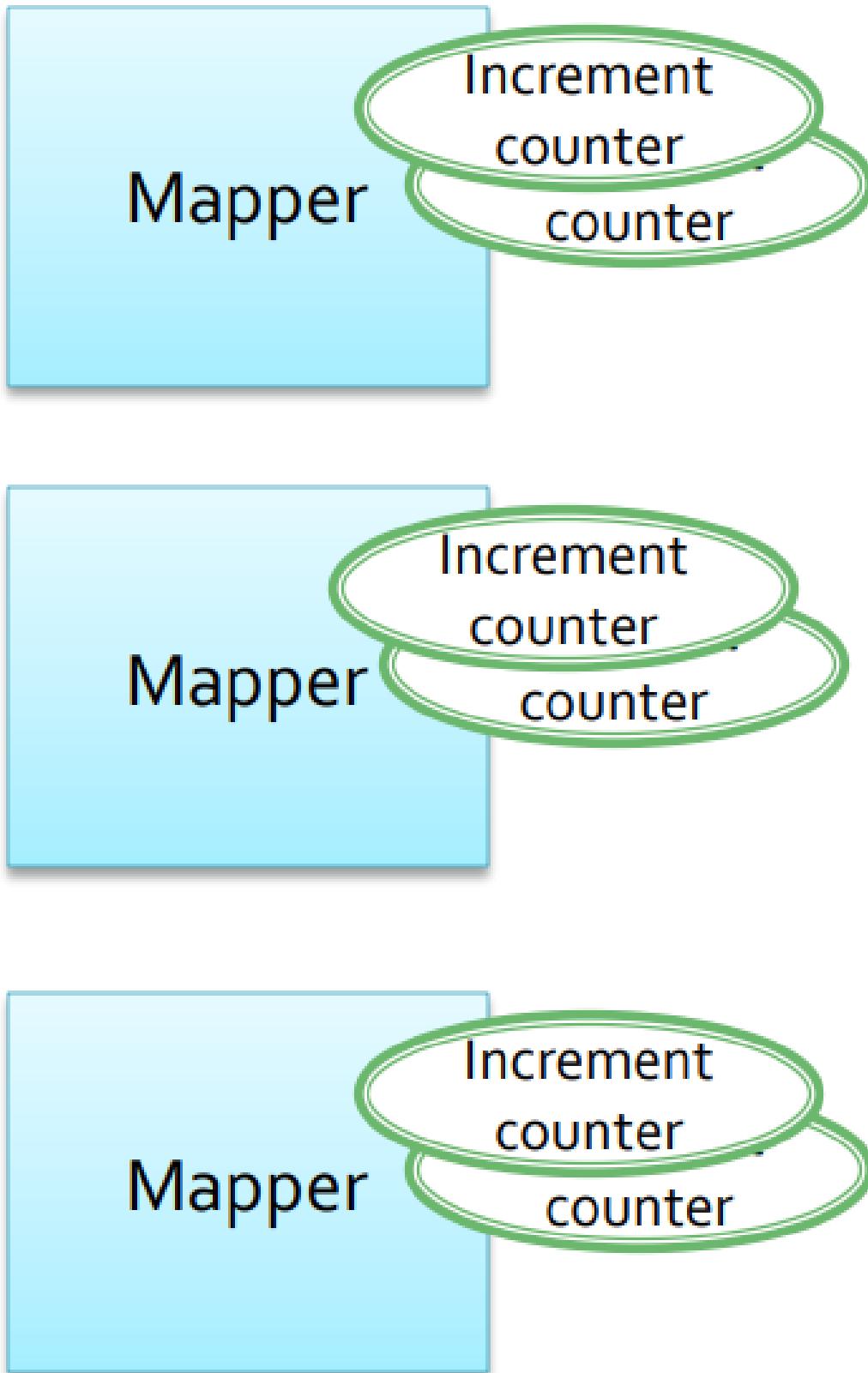
Counting with counters

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, ...

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

Figure 7.3: Numerical summarization structure



💡 Use cases

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

7.2 Filtering patterns

Are used to select the subset of input records of interest

- Filtering
- Top K
- Distinct

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.

Structure

The input of the mapper is a set of records

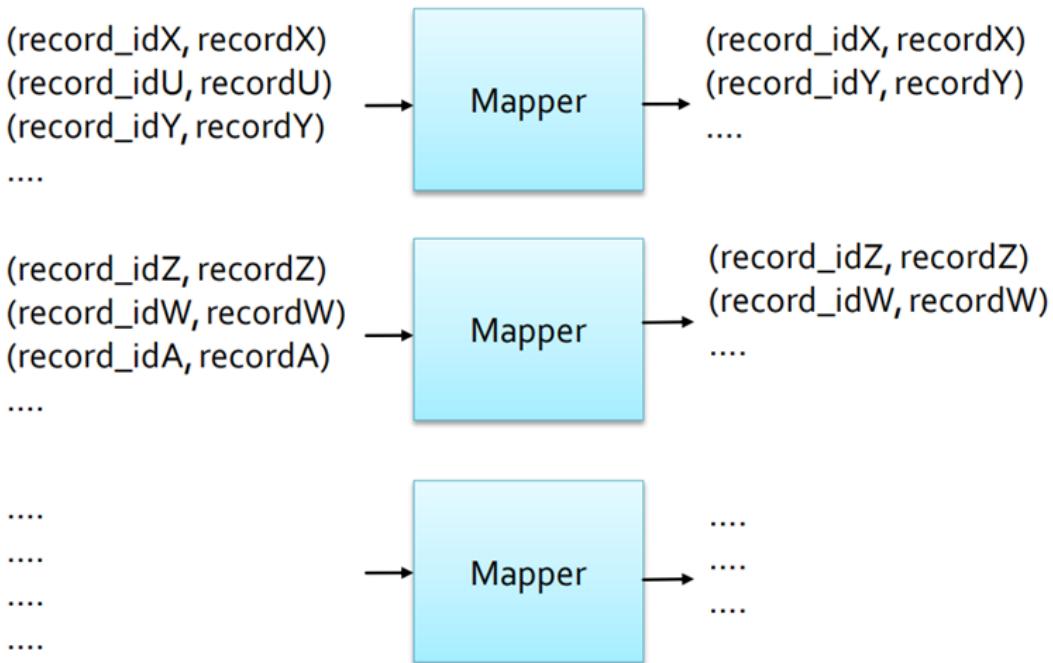
- Key = primary key
- Value = record

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).

Figure 7.4: Numerical summarization structure



💡 Use cases

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

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).

Structure

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

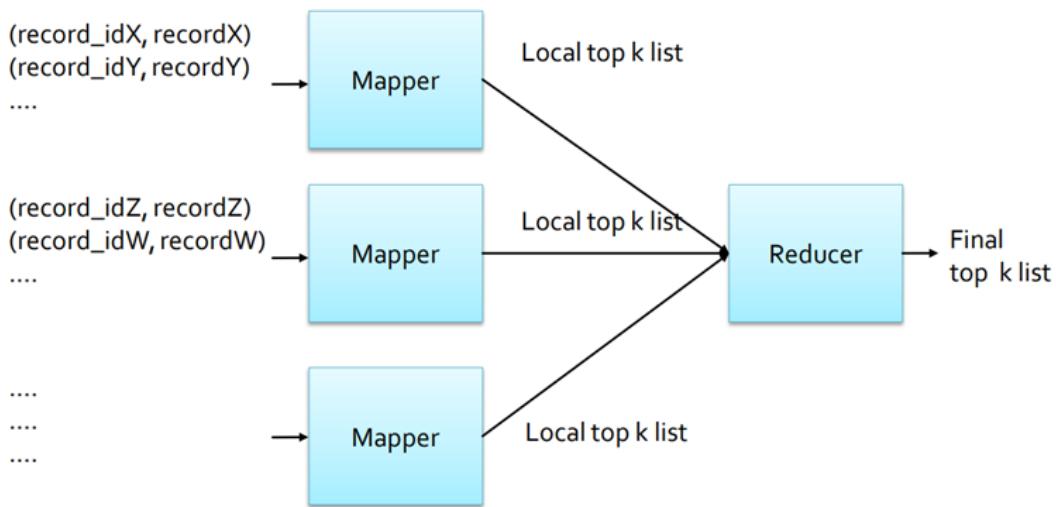
The cleanup method emits the k (`key, value`) pairs associated with the in-mapper local top k records

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

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 7.5: Numerical summarization structure



💡 Use cases

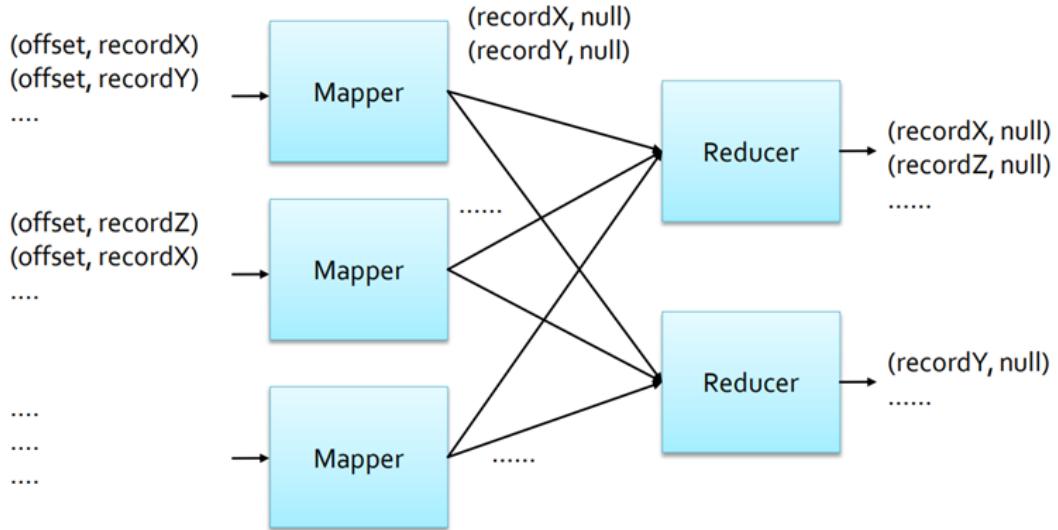
- Outlier analysis (based on a ranking function)
- Select interesting data (based on a ranking function)

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

Figure 7.6: Numerical summarization structure



💡 Use cases

- Duplicate data removal
- Distinct value selection

8 MapReduce and Hadoop Advanced Topics

8.1 Multiple inputs

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.

💡 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

Example: multiple inputs

```
1  MultipleInputs.addInputPath(
2      job,
3      new Path(args[1]),
4      TextInputFormat.class,
5      Mapper1.class
6  );
7
8  MultipleInputs.addInputPath(
9      job,
10     new Path(args[2]),
11     TextInputFormat.class,
12     Mapper2.class
13 );
```

- ① Specify two input paths (`args[1]` and `args[2]`)
② The data of both paths are read by using the `TextInputFormat` class
③ `Mapper1` is the class used to manage the input key-value pairs associated with the first path

- ④ Mapper2 is the class used to manage the input key-value pairs associated with the second path

8.2 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 mapper for map-only jobs)

Driver

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

- 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”

Example: multiple outputs

```
1  MultipleOutputs.addNamedOutput(
2      job,
3      "hightemp",
4      TextOutputFormat.class,
5      Text.class,
6      NullWritable.class
7  );
8
9  MultipleOutputs.addNamedOutput(
10     job,
11     "normaltemp",
```

```

12     TextOutputFormat.class,
13     Text.class,
14     NullWritable.class
15 );

```

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"

Map-only

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

```

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

```

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

```

1 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: map-only

```

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

```

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

```

1 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)

```

1 mos.close();

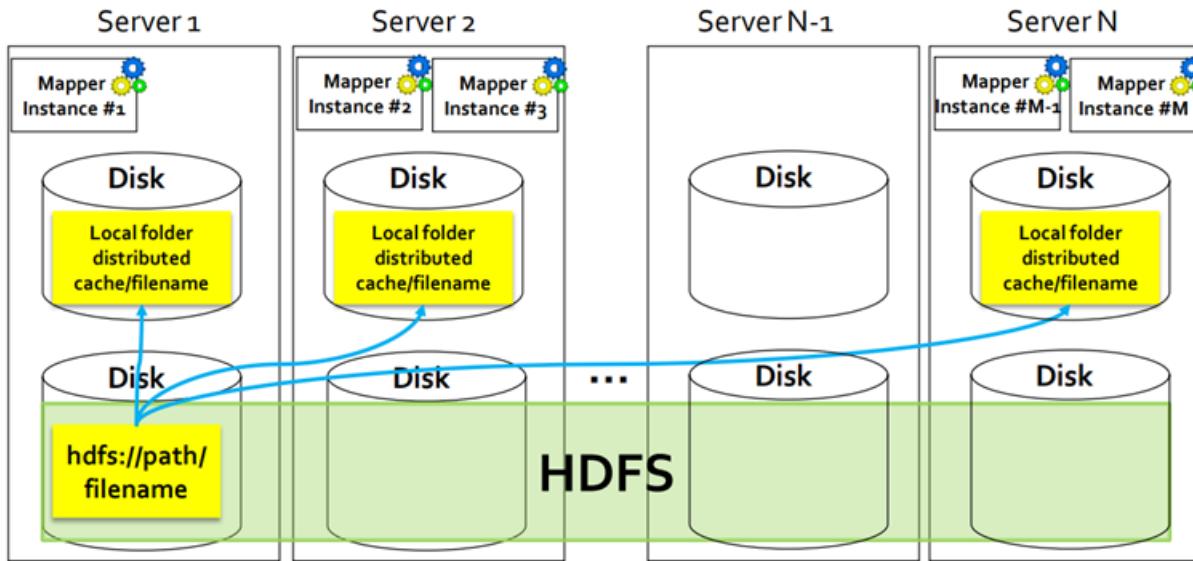
```

8.3 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 8.1: Distributed cache structure



In image Figure 8.1, 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 specified by 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).

Example: distributed cache

Driver

```

1  public int run(String[] args) throws Exception {
2      //...
3
4      // Add the shared/cached HDFS file in the
5      // distributed cache
6      job.addCacheFile(new Path("hdfs path/filename").toUri());
7
8      //...
9  }

```

Mapper/Reducer

```

1  protected void setup(Context context) throws IOException, InterruptedException{
2
3      String line;
4      // Retrieve the (original) paths of the distributed files
5      URI[] urisCachedFiles = context.getCacheFiles();
6
7      // Read the content of the cached file and process it.
8      // In this example the content of the first shared file is opened.
9      BufferedReader file = new BufferedReader(
10          new FileReader(
11              new File(
12                  new Path(urisCachedFiles[0].getPath()).getName()
13              )
14          )
15      );
16
17      // Iterate over the lines of the file
18      while ((line = file.readLine()) != null) {
19          // process the current line
20          //...
21      }
22      file.close();
23  }

```

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.

9 MapReduce patterns - 2

9.1 Data organization patterns

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)

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.

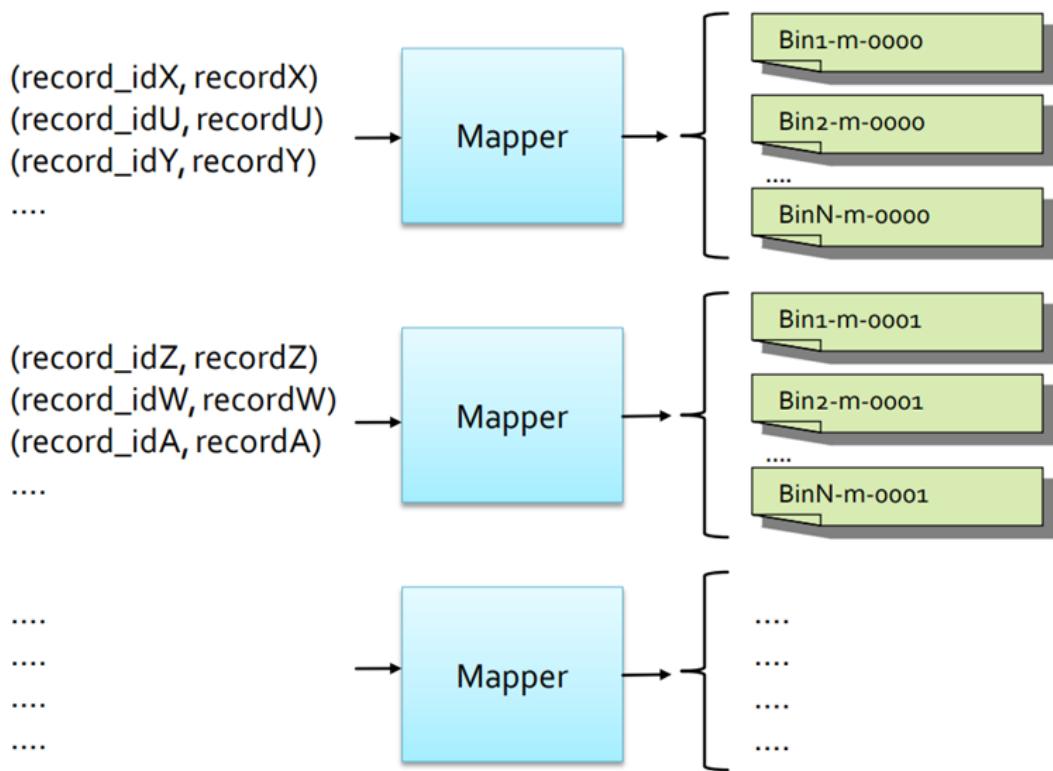
This is done because the input data set contains heterogenous data, but each data analysis usually is focused only on a specific subsets of the data.

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

Figure 9.1: Binning structure



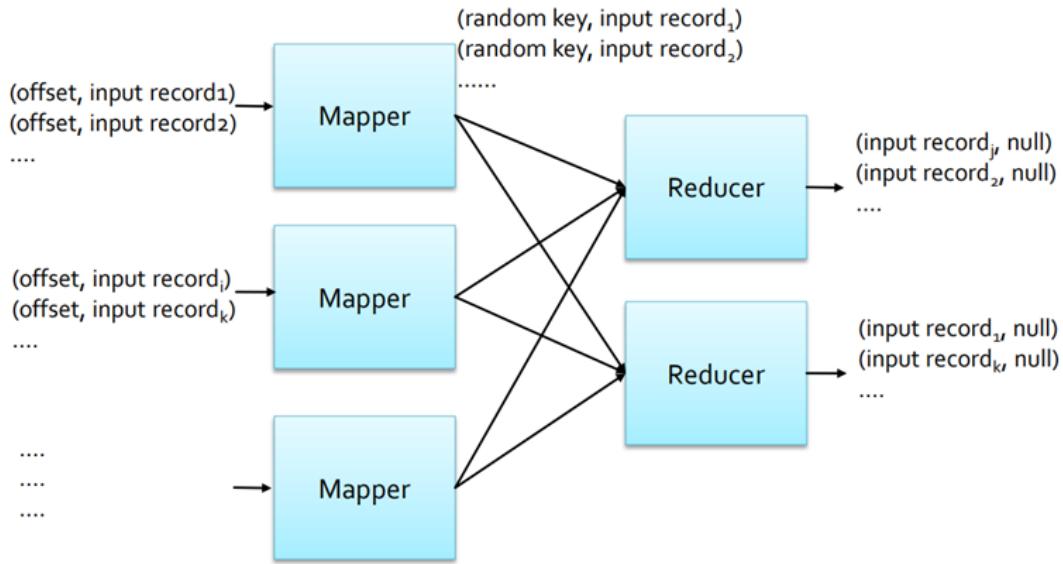
Shuffling

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

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 9.2: Shuffling structure



9.2 Metapatterns

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

- Job Chaining

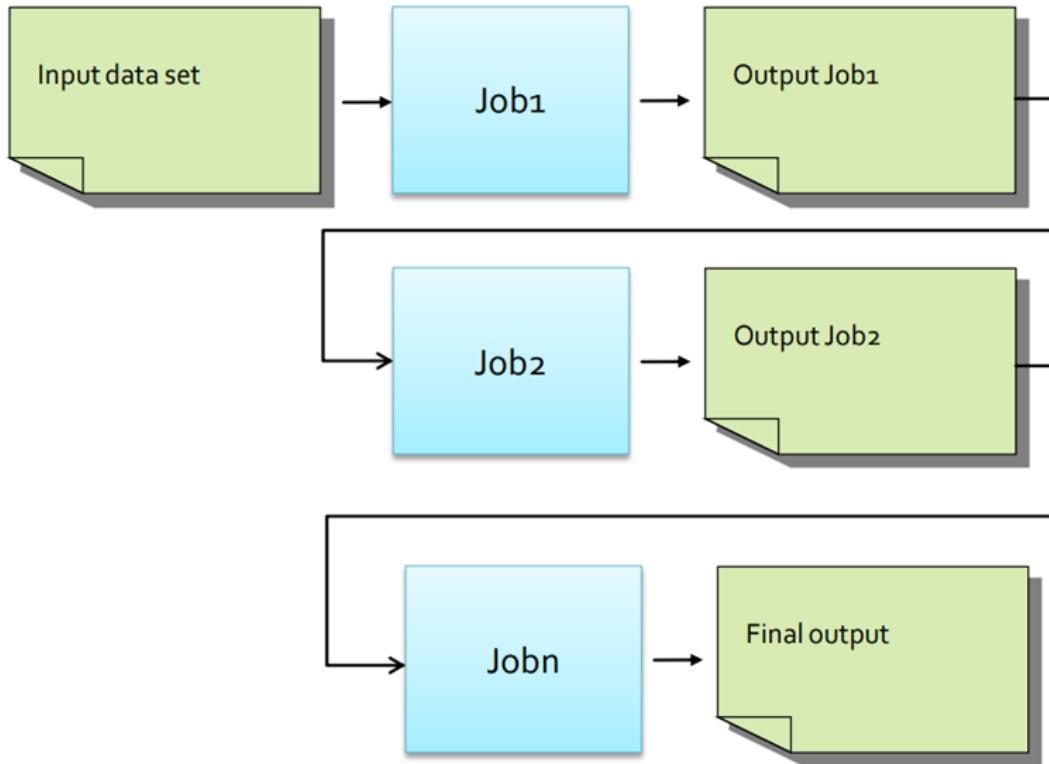
Job Chaining

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.

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 implemented by a MapReduce Job (i.e., it is associated with a mapper, a reducer, and a combiner, if it is useful)

Figure 9.3: Job chaining structure



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

9.3 Join patterns

Are used 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).

Reduce side natural join

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

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: join pattern

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

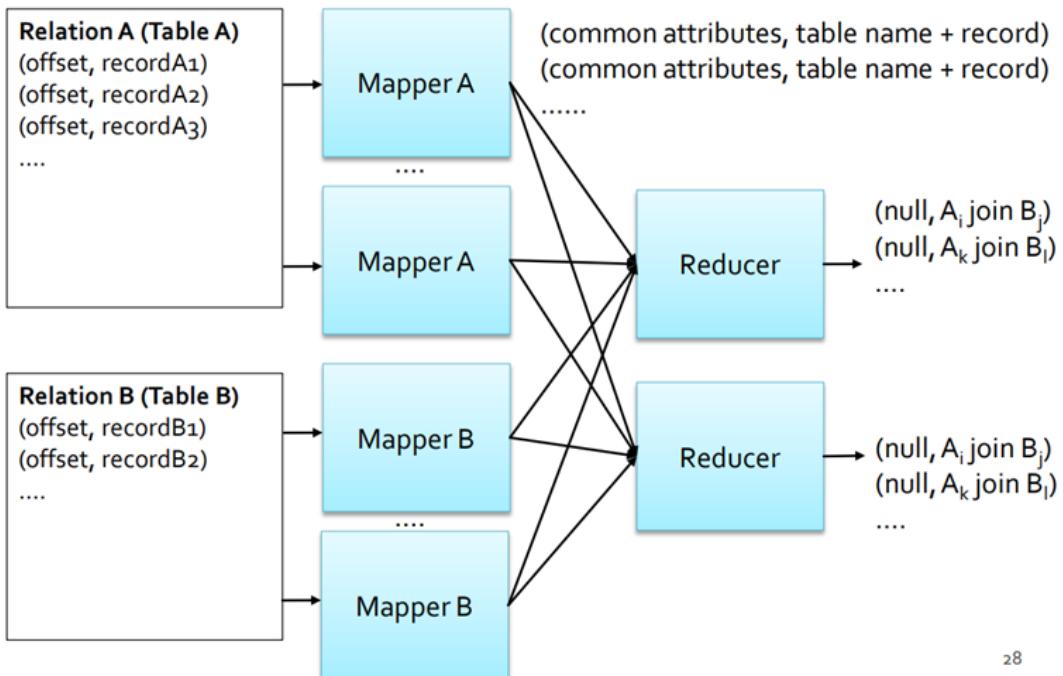
The (`key, [list of values]`) pair

```
1 (userid=u1, ["User:name=Paolo, surname=Garza", "Likes:movieGenre=horror", "Likes:movieGenre=adventure"])
```

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

```
1 (userid=u1, "name=Paolo, surname=Garza, genre=horror")
2
3 (userid=u1, "name=Paolo, surname=Garza, genre=adventure")
```

Figure 9.4: Reduce side natural join structure



28

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).

Structure

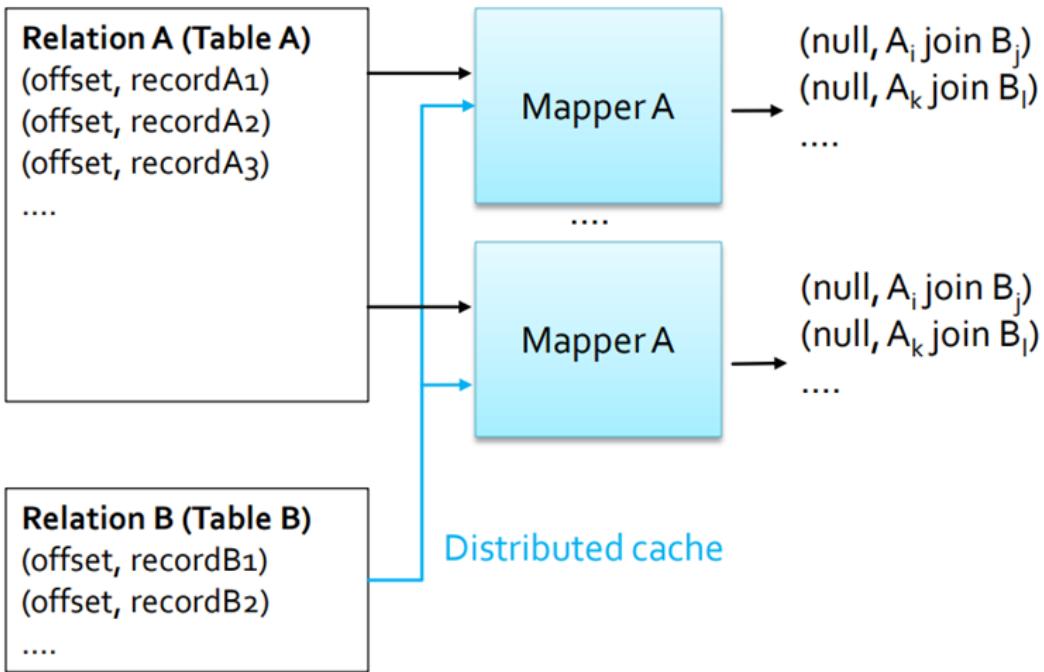
This is a Map-only job

- 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 9.5: Map side natural join structure



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).

10 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.

10.1 Selection

$$\sigma_C(R)$$

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.

i 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

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

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 $(\text{key}, \text{value})$ pair with $\text{key}=\text{record}$ and $\text{value}=\text{null}$, otherwise, it discards the record.

10.2 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), and it removes duplicates, if any.

i 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.

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

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 $(\text{key}, \text{value})$ pair with $\text{key}=r'$ and $\text{value}=\text{null}$.
- Each reducer emits one $(\text{key}, \text{value})$ pair for each input $(\text{key}, [\text{list of values}])$ pair with $\text{key}=r'$ and $\text{value}=\text{null}$.

10.3 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.

i 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.

$$\text{DegreeCourseProf} \cup \text{MasterCourseProf}$$

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 $(\text{key}, \text{value})$ pair with $\text{key}=t$ and $\text{value}=\text{null}$, and for each input record t in S , emit one $(\text{key}, \text{value})$ pair with $\text{key}=t$ and $\text{value}=\text{null}$.

- 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).

10.4 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.

$$\text{DegreeCourseProf} \cap \text{MasterCourseProf}$$

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 .

10.5 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 .

i 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.

$$\text{DegreeCourseProf} - \text{MasterCourseProf}$$

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=null` 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 .

10.6 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.

10.7 Aggregations and Group by

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

11 How to submit/execute a Spark application

11.1 Spark submit

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.

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

```
1 --master
```

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

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

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

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

It specifies where the Driver is launched/executed

<code>client</code>	The driver is launched locally (in the “local” PC executing <code>spark-submit</code>)
<code>cluster</code>	The driver is launched on one node of the cluster

💡 Deployment mode: cluster and client

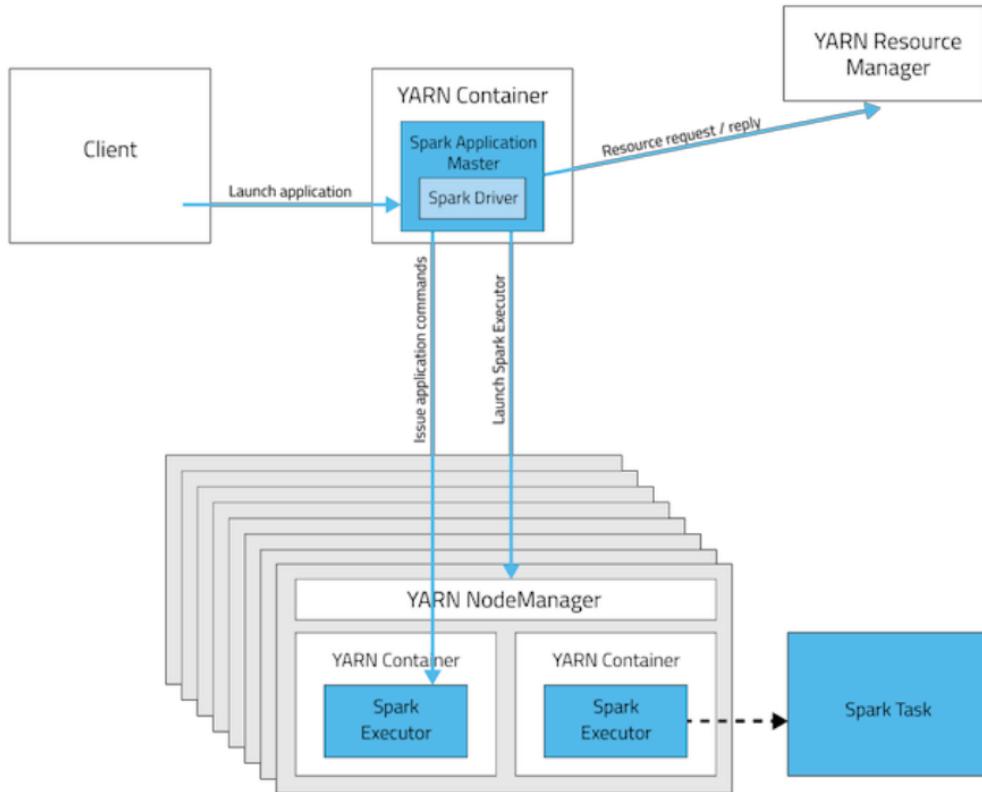
In **cluster** mode

- 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.

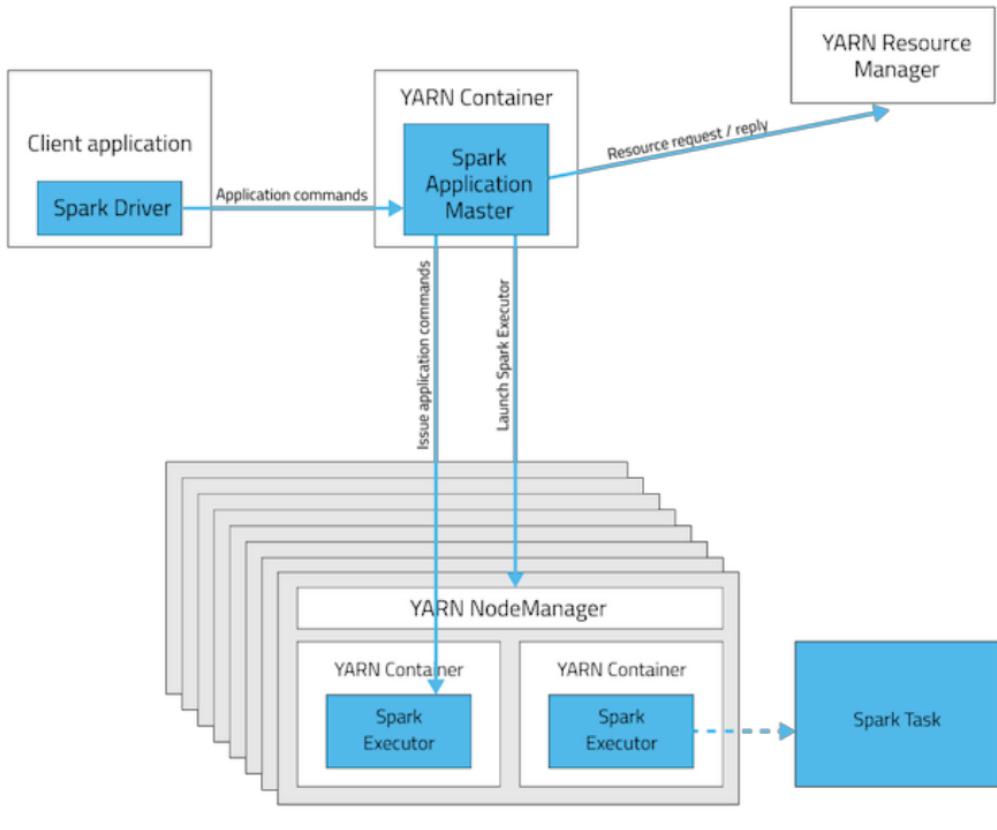
Figure 11.1: Cluster deployment mode



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 11.2: Client deployment mode



Setting the executors

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

option	meaning	default value
<code>--num-executors</code>	The number of executors	2 executors
<code>--executor-cores</code>	The number of cores per executor	1 core
<code>--executor-memory</code>	Main memory per executor	1 GB

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

Setting the drivers

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

option	meaning	default value
<code>--driver-cores</code>	The number of cores for the driver	1 core
<code>--driver-memory</code>	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`.

Execution examples

The following command submits a Spark application on a Hadoop cluster

```
1 spark-submit \
2 --deploy-mode cluster \
3 --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

```
1 spark-submit \
2 --deploy-mode client \
3 --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

12 Introduction to Spark

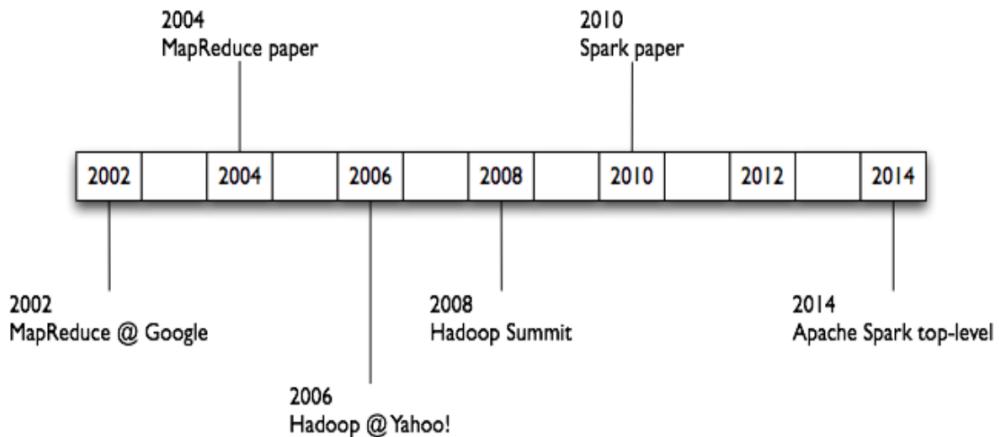
Apache Spark™ is a fast and general-purpose engine for large-scale data processing. Spark aims at achieving the following goals in the Big data context:

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

💡 History

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

Figure 12.1: Spark history

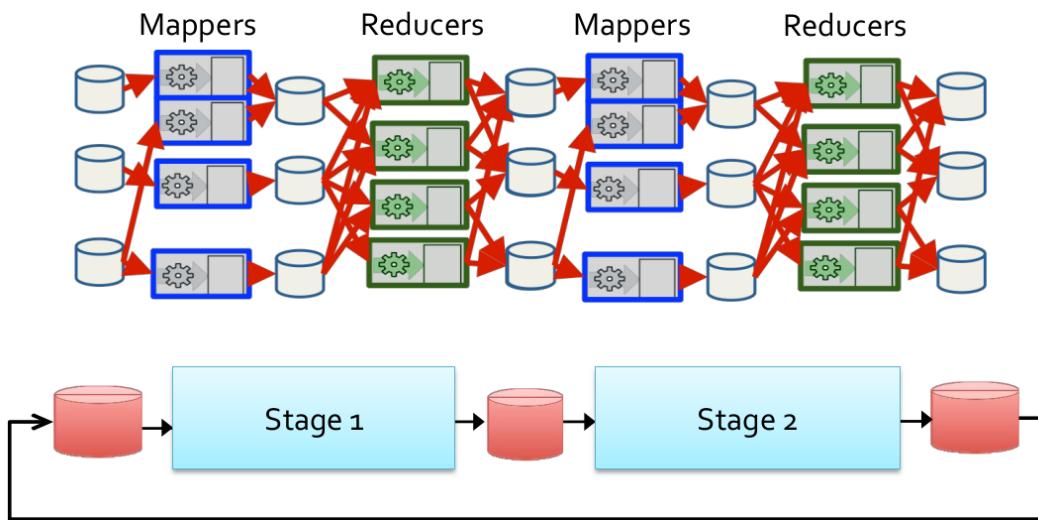


12.1 Motivations

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)

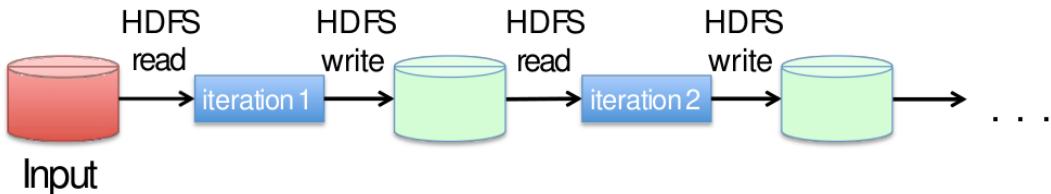
Figure 12.2: Iterative jobs



- 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

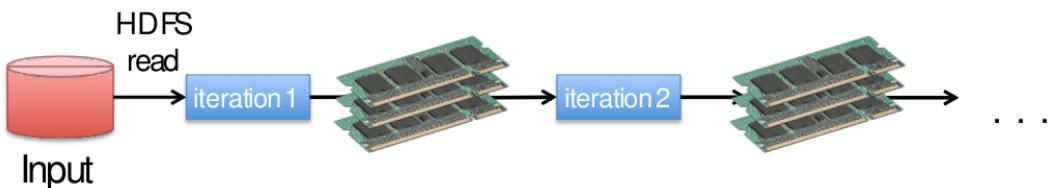
So an iterative job in MapReduce makes wide use of disk reading/writing

Figure 12.3: Iterative jobs in MapReduce



Instead, an iterative job in Spark uses the main memory

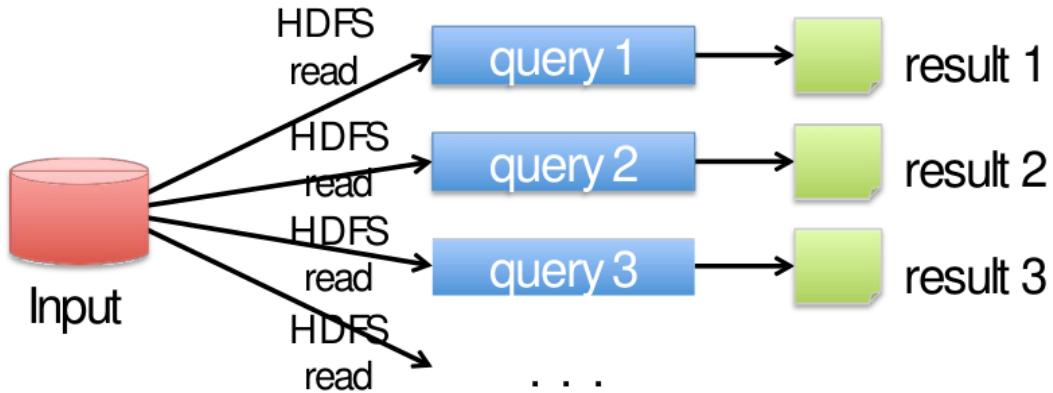
Figure 12.4: Iterative jobs in Spark



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.

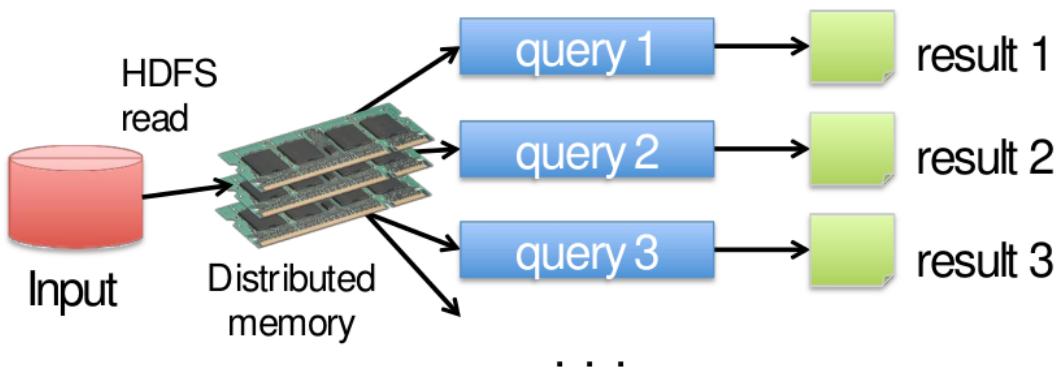
Moreover, to run multiple queries on the same data, in MapReduce the data must be read multiple times (once for each query)

Figure 12.5: Analysing the same data in MapReduce



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

Figure 12.6: 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.

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

MapReduce vs Spark

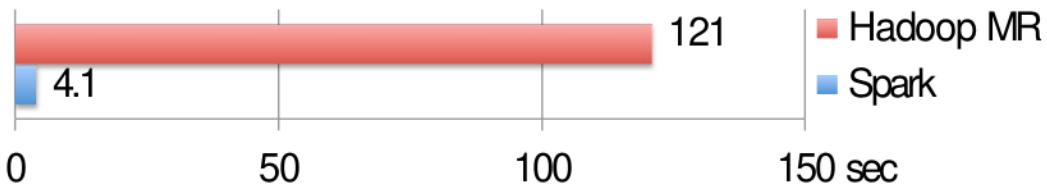
	Hadoop MapReduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, ...
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, Python, R

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 12.7: Performance comparison

K-means Clustering

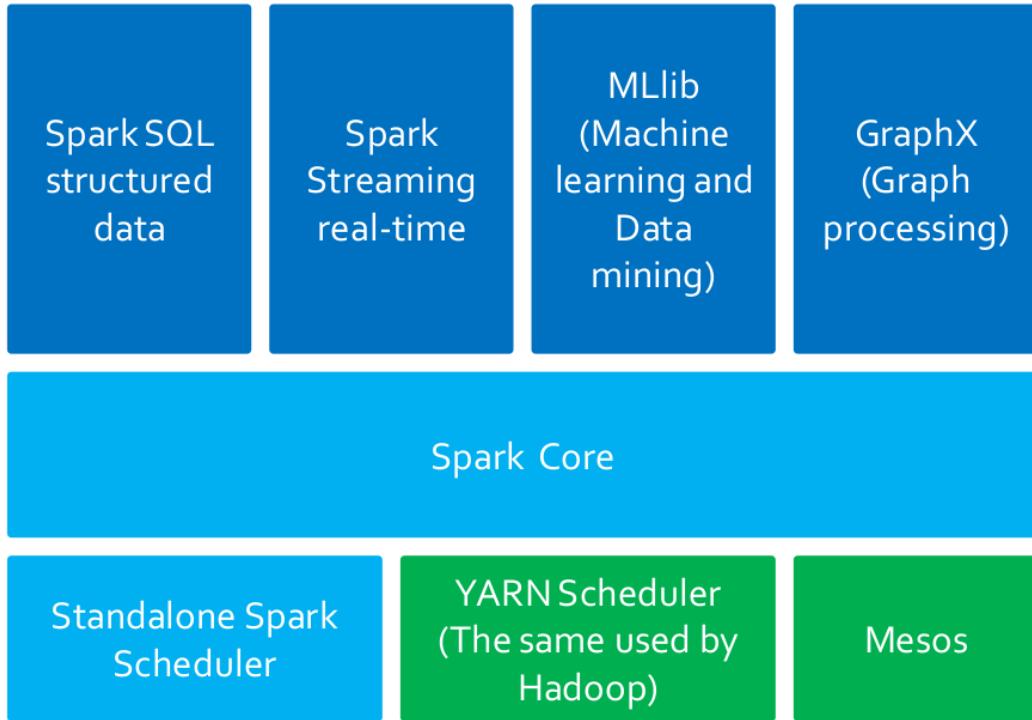


Logistic Regression



12.2 Main components

Figure 12.8: 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.

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.

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).

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.

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
- ...

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.

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

12.3 Basic concepts

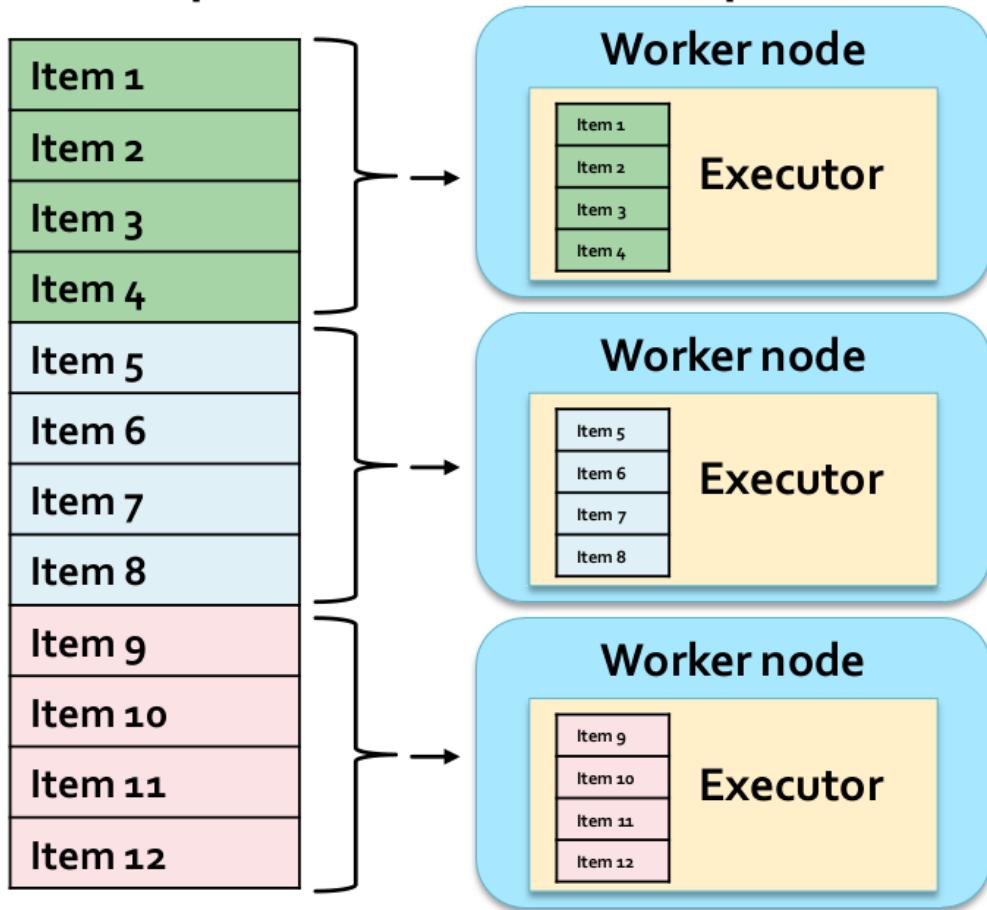
Resilient Distributed Data sets (RDDs)

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 each node: each executor of a worker node runs the specified code on its partition of the RDD.

i Example of an RDD split in 3 partitions

Figure 12.9: 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 rebuild 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, Python, 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)

12.4 Spark Programs

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

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 <code>main()</code> function of the application and creating the <code>SparkContext</code> .
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.

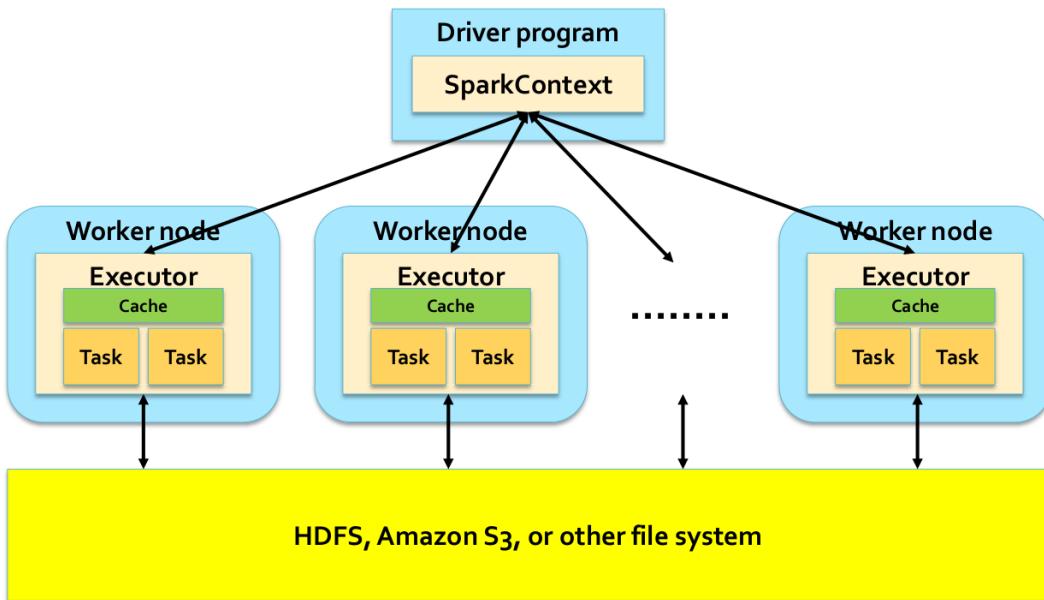
The Driver program defines Resilient Distributed Datasets (RDDs) that are allocated in the nodes of the cluster, and invokes parallel operations on RDDs.

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.

Figure 12.10: Distributed execution of Spark

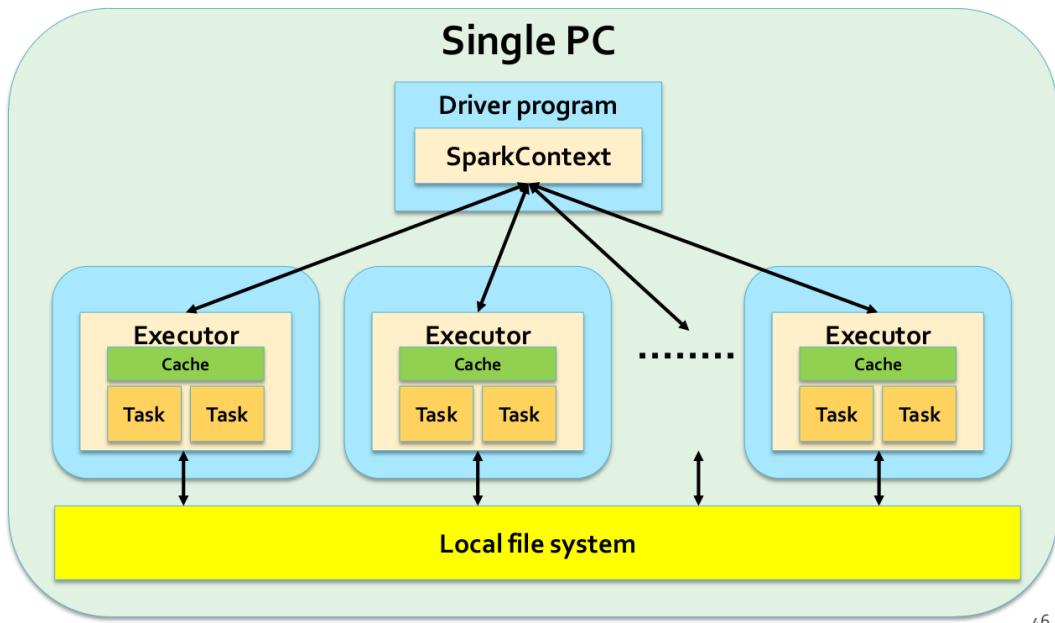


RDDs are distributed across executors (each RDD is split in partitions that are spread across the available executors).

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 as “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 12.11: Distributed execution of Spark



12.5 Spark program examples

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

```

1  from pyspark import SparkConf, SparkContext
2
3  if __name__ == "__main__":
4
5      ## Create a configuration object and
6      ## set the name of the application
7      conf = SparkConf().setAppName("Spark Line Count")          ①
8
9      ## Create a Spark Context object
10     sc = SparkContext(conf=conf)
11
12    ## Store the path of the input file in inputFile
13    inputFile= "myfile.txt"
14
15    ## Build an RDD of Strings from the input textual file
16    ## Each element of the RDD is a line of the input file
17    linesRDD = sc.textFile(inputFile)                            ②

```

```

18
19     ## Count the number of lines in the input file
20     ## Store the returned value in the local variable numLines
21     numLines = linesRDD.count() (1)
22
23     ## Print the output in the standard output
24     print("NumLines:", numLines)
25
26     ## Close the Spark Context object
27     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

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.

```

1  from pyspark import SparkConf, SparkContext
2  import sys
3
4  if __name__ == "__main__":
5      """
6          Word count example
7      """
8      inputFile= sys.argv[1]
9      outputPath = sys.argv[2]
10
11     ## Create a configuration object and
12     ## set the name of the application
13     conf = SparkConf().setAppName("Spark Word Count")
14
15     ## Create a Spark Context object
16     sc = SparkContext(conf=conf)
17

```

```
18     ## Build an RDD of Strings from the input textual file
19     ## Each element of the RDD is a line of the input file
20     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()
```

13 RDD based programming

13.1 Spark Context

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

Example

```
1 ## Create a configuration object and
2 ## set the name of the application
3 conf = SparkConf().setAppName("Application name")
4
5 ## Create a Spark Context object
6 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.

Example

```
1 ## Create a configuration object and
2 ## set the name of the application
3 conf = SparkConf().setAppName("Application name")
4
5 ## Retrieve the current SparkContext object or
6 ## create a new one
7 sc = SparkContext.getOrCreate(conf=conf)
```

13.2 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.

13.3 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)

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

```

1 ## Build an RDD of strings from the input textual file
2 ## myfile.txt
3 ## Each element of the RDD is a line of the input file
4 inputFile = "myfile.txt"
5 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

```

1 ## Build an RDD of strings from all the files stored in
2 ## myfolder
3 ## Each element of the RDD is a line of the input files
4 inputFolder = "myfolder/"
5 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

```

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

```

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

```

1 ## Create a local python list
2 inputList = [
3     'First element',
4     'Second element',
5     'Third element'
6 ]
7
8 ## Build an RDD of Strings from the local list.
9 ## The number of partitions is set automatically by Spark
10 ## There is one element of the RDD for each element
11 ## of the local list
12 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 `distrRDDlist`, 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

```

1 ## Create a local python list
2 inputList = [
3     'First element',
4     'Second element',
5     'Third element'
6 ]
7
8 ## Build an RDD of Strings from the local list.
9 ## The number of partitions is set to 3
10 ## There is one element of the RDD for each element
11 ## of the local list
12 distrRDDList = sc.parallelize(inputList, 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

```

1 ## Store the content of linesRDD in the output folder
2 ## Each element of the RDD is stored in one line
3 ## of the textual files of the output folder
4 outputPath="risFolder/"
5 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.

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

```
1 ## Retrieve the content of the linesRDD and store it
2 ## in a local python list
3 ## The local python list contains a copy of each
4 ## element of linesRDD
5 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

13.4 Transformations and Actions

RDD support two types of operations

- Transformations
- Actions

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.

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.

Example of lineage graph (DAG)

Consider the following code

```

1  from pyspark import SparkConf, SparkContext
2  import sys
3
4  if __name__ == "__main__":
5      conf = SparkConf().setAppName("Spark Application")
6      sc = SparkContext(conf=conf)
7
8      ## Read the content of a log file
9      inputRDD = sc.textFile("log.txt")
10
11     ## Select the rows containing the word "error"
12     errorsRDD = inputRDD.filter(lambda line: line.find('error')>=0)
13
14     ## Select the rows containing the word "warning"
15     warningRDD = inputRDD.filter(lambda line: line.find('warning')>=0)
16
17     ## Union of errorsRDD and warningRDD
18     ## The result is associated with a new RDD: badLinesRDD
19     badLinesRDD = errorsRDD.union(warningRDD)

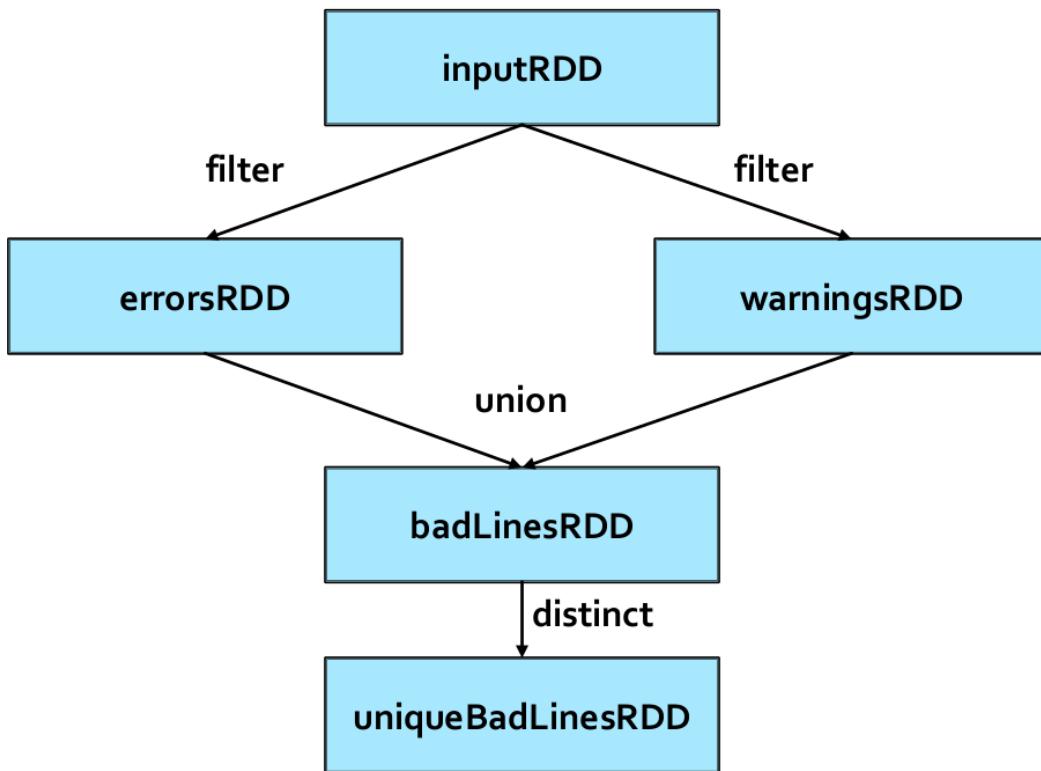
```

```

20
21 ## Remove duplicates lines (i.e., those lines containing
22 ## both "error" and "warning")
23 uniqueBadLinesRDD = badLinesRDD.distinct()
24
25 ## Count the number of bad lines by applying
26 ## the count() action
27 numBadLines = uniqueBadLinesRDD.count()
28
29 ## Print the result on the standard output of the driver
30 print("Lines with problems:", numBadLines)

```

Figure 13.1: 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 perform this kind of optimizations only on particular types of RDDs: Datasets and DataFrames.

13.5 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

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.

Solution based on lambda expressions (`lambda`)

```
1 ## Read the content of a log file
2 inputRDD = sc.textFile("log.txt")
3
4 ## Select the rows containing the word "error"
5 errorsRDD = inputRDD.filter(lambda l: l.find('error')>=0)
```

`lambda l:` 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.

Solution based on function (def)

```
1 ## Define the content of the Boolean function that is applied
2 ## to select the elements of interest
3 def myFunction(l):
4     if l.find('error')>=0: return True
5     else: return False
6
7 ## Read the content of a log file
8 inputRDD = sc.textFile("log.txt")
9
10 ## Select the rows containing the word "error"
11 errorsRDD = inputRDD.filter(myFunction)
```

def When it is invoked, this function analyses the value of the parameter line and
myFunction(l): 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.

Solution based on function (def)

```
1 ## Define the content of the Boolean function that is applied
2 ## to select the elements of interest
3 def myFunction(l):
4     return l.find('error')>=0
5
6 ## Read the content of a log file
7 inputRDD = sc.textFile("log.txt")
8
9 ## Select the rows containing the word "error"
10 errorsRDD = inputRDD.filter(myFunction)
```

return This part of the code is the same used in the lambda-based solution.
l.find('error')>=0
.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.

Solution comparison

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

Lambda function-based code (<code>lambda</code>)	Local user defined functions (local <code>def</code>)
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)

13.6 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()`, `subtract()`, `cartesian()`)

Single input RDD transformations

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.

i 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”.

```

1 ## Read the content of a log file
2 inputRDD = sc.textFile("log.txt")
3
4 ## Select the rows containing the word "error"
5 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`

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputList = [1, 2, 3, 3]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Select the values greater than 2
6 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`

```

1 ## Define the function to be applied in the filter transformation
2 def greaterThan2(num):
3     return num>2
4
5 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
6 inputList = [1, 2, 3, 3]
7 inputRDD = sc.parallelize(inputList)
8
9 ## Select the values greater than 2
10 greaterRDD = inputRDD.filter(greaterThan2)

```

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

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`.

i 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.

```

1 ## Read the content of the input textual file
2 inputRDD = sc.textFile("usernames.txt")
3
4 ## Compute the lengths of the input surnames
5 lengthsRDD = 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.

i 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.

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputList = [1, 2, 3, 3]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Compute the square of each input element
6 squaresRDD = inputRDD.map(lambda element: element*element)

```

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.

i 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.

```

1 ## Read the content of the input textual file
2 inputRDD = sc.textFile("document.txt")
3
4 ## Compute/identify the list of words occurring in document.txt
5 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.

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 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.

i 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.

```

1 ## Read the content of a textual input file
2 inputRDD = sc.textFile("names.txt")
3
4 ## Select the distinct names occurring in inputRDD
5 distinctNamesRDD = inputRDD.distinct()

```

i 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.

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputList = [1, 2, 3, 3]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Compute the set of distinct words occurring in inputRDD
6 distinctIntRDD = inputRDD.distinct()

```

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

i 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).

```

1 ## Read the content of a textual input file
2 inputRDD = sc.textFile("names.txt")
3
4 ## Sort the content of the input RDD by name.
5 ## Store the sorted result in a new RDD
6 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.

i 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)`).

```

1 ## Read the content of a textual input file
2 inputRDD = sc.textFile("names.txt")
3
4 ## Sort the content of the input RDD by name.
5 ## Store the sorted result in a new RDD
6 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).

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]).

i 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).

```

1 ## Read the content of a textual input file
2 inputRDD = sc.textFile("sentences.txt")
3
4 ## Create a random sample of sentences
5 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).

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputList = [1, 2, 3, 3]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Create a sample of the inputRDD
6 randomSentencesRDD = inputRDD.sample(True,0.2)

```

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.

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.

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.

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.

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.

Examples of set transformations

i 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`

```

1 ## Create two RDD of integers
2 inputList1 = [1,2,2,3,3]
3 inputRDD1 = sc.parallelize(inputList1)
4
5 inputList2 = [3,4,5]
6 inputRDD2 = sc.parallelize(inputList2)
7
8 ## Create four new RDDs by using union, intersection,
9 ## subtract, and cartesian
10 outputUnionRDD = inputRDD1.union(inputRDD2)
11
12 outputIntersectionRDD = inputRDD1.intersection(inputRDD2)
13
14 outputSubtractRDD = inputRDD1.subtract(inputRDD2)
15
16 outputCartesianRDD = inputRDD1.cartesian(inputRDD2)

```

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

i 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`

```
1 ## Create an RDD of Integers and an RDD of Strings
2 inputList1 = [1,2,3]
3 inputRDD1 = sc.parallelize(inputList1)
4
5 inputList2 = ["A","B"]
6 inputRDD2 = sc.parallelize(inputList2)
7
8 ## Compute the cartesian product
9 outputCartesianRDD = inputRDD1.cartesian(inputRDD2)
```

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

Summary

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].

Purposes

Transformation	Purpose
<code>filter(f)</code>	Return an RDD consisting only of the elements of the input RDD that pass the condition passed to <code>filter()</code> . The input RDD and the new RDD have the same data type.
<code>map(f)</code>	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.
<code>flatMap(f)</code>	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.
<code>distinct()</code>	Remove duplicates.
<code>sortBy(keyfunc)</code>	Return a new RDD containing the same values of the input RDD sorted in ascending order.
<code>sample(withReplacement, fraction)</code>	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.

Examples

Transformation	Example function	Example result
<code>filter(f)</code>	<code>filter(lambda x: x != 1)</code>	<code>[2, 3, 3]</code>
<code>map(f)</code>	<code>map(lambda x: x+1)</code>	<code>[2, 3, 4, 4]</code>
	For each input element <code>x</code> , the element with value <code>x+1</code> is included in the new RDD	
<code>flatMap(f)</code>	<code>flatMap(lambda x: list(range(x,4))</code>	<code>[1, 2, 3, 2, 3, 3, 3]</code>
	For each input element <code>x</code> , the set of elements with values from <code>x</code> to 3 are returned	
<code>distinct()</code>	<code>distinct()</code>	<code>[1, 2, 3]</code>
<code>sortBy(keyfunc)</code>	<code>sortBy(lambda v: v)</code>	<code>[1, 2, 3, 3]</code>
	Sort the input integer values in ascending order by using the standard integer sort order	
<code>sample(withReplacement, fraction)</code>	<code>sample(True, 0.2)</code>	Non deterministic

Two input RDD transformations

All the examples reported in the following tables are applied on the following two RDDs of integers

- `inputRDD1: [1,2,2,3,3]`
- `inputRDD2: [3,4,5]`

Purposes	Transformation	Purpose
	<code>union(other)</code>	Return a new RDD containing the union of the elements of the input RDD and the elements of the one passed as parameter to <code>union()</code> . Duplicate values are not removed. All the RDDs have the same data type.
	<code>intersection(other)</code>	Return a new RDD containing the intersection of the elements of the input RDD and the elements of the one passed as parameter to <code>intersection()</code> . All the RDDs have the same data type.
	<code>subtract(other)</code>	Return a new RDD the elements appearing only in the input RDD and not in the one passed as parameter to <code>subtract()</code> . All the RDDs have the same data type.
	<code>cartesian(other)</code>	Return a new RDD containing the cartesian product of the elements of the input RDD and the elements of the one passed as parameter to <code>cartesian()</code> . All the RDDs have the same data type.

Examples

Transformation	Example function	Example result
union(other)	inputRDD1.union(inputRDD2)	[1, 2, 2, 3, 3, 4, 5]
intersection(other)	inputRDD1.intersection(inputRDD2)	[3]
subtract(other)	inputRDD1.subtract(inputRDD2)	[1, 2, 2]
cartesian(other)	inputRDD1.cartesian(inputRDD2)	[(1, 3), (1, 4), ..., (3, 5)]

13.7 Basic Actions

Spark actions can retrieve the content of an RDD or the result of a function applied on an RDD and

- Store it in a local Python variable of the Driver program
 - Pay attention to the size of the returned value
 - Pay attentions that date are sent on the network from the nodes containing the content of RDDs and the executor running the Driver
- Store the content of an RDD in an output folder or database

The Spark actions that return a result that is stored in local (Python) variables of the Driver 1. Are executed locally on each node containing partitions of the RDD on which the action is invoked, and so local results are generated in each node; 2. Local results are sent on the network to the Driver that computes the final result and store it in local variables of the Driver.

The basic actions returning (Python) objects to the Driver are

- `collect()`
- `count()`
- `countByValue()`
- `take()`
- `top()`
- `takeSample()`
- `reduce()`
- `fold()`
- `aggregate()`
- `foreach()`

Collect action

The collect action 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 RDD cannot be memorized in a local variable of the Driver.

The collect action is based on the `collect()` method of the `RDD` class.

Example

1. Create an RDD of integers containing the values [1,2,3,3];
2. Retrieve the values of the created RDD and store them in a local python list that is instantiated in the Driver.

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputList = [1,2,3,3]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Retrieve the elements of the inputRDD and store them in
6 ## a local python list
7 retrievedValues = inputRDD.collect()

```

<code>inputRDD</code>	It is distributed across the nodes of the cluster. It can be large and it is stored in the local disks of the nodes if it is needed.
<code>retrievedValues</code>	It is a local python variable. It can only be stored in the main memory of the process/task associated with the Driver. Pay attention to the size of the list. Use the <code>collect()</code> action if and only if sure that the list is small. Otherwise, store the content of the RDD in a file by using the <code>saveAsTextFile</code> method.

Count action

Count the number of elements of an RDD.

The count action is based on the `count()` method of the `RDD` class: it returns the number of elements of the input RDD.

i Example

1. Consider the textual files “document1.txt” and “document2.txt”;
2. Print the name of the file with more lines.

```

1 ## Read the content of the two input textual files
2 inputRDD1 = sc.textFile("document1.txt")
3 inputRDD2 = sc.textFile("document2.txt")
4
5 ## Count the number of lines of the two files = number of elements
6 ## of the two RDDs
7 numLinesDoc1 = inputRDD1.count()
8 numLinesDoc2 = inputRDD2.count()
9
10 if numLinesDoc1 > numLinesDoc2:
11     print("document1.txt")
12 elif numLinesDoc2 > numLinesDoc1:
13     print("document2.txt")
14 else:
15     print("Same number of lines")

```

CountByValue action

The `countByValue` action returns a local python dictionary containing the information about the number of times each element occurs in the RDD

- The keys of the dictionary are associated with the input elements
- The values are the frequencies of the elements

The `countByValue` action is based on the `countByValue()` method of the `RDD` class. The amount of used main memory in the Driver is related to the number of distinct elements/keys.

Example

1. Create an RDD from a textual file containing the first names of a list of users (each line contain one name);
2. Compute the number of occurrences of each name and store this information in a local variable of the Driver.

```
1 ## Read the content of the input textual file
2 namesRDD = sc.textFile("names.txt")
3
4 ## Compute the number of occurrences of each name
5 namesOccurrences = namesRDD.countByValue()
```

```
namesOccurrences =
namesRDD.countByValue()
```

Also in this case, pay attention to the size of the returned dictionary (that is related to the number of distinct names in this case). Use the `countByValue()` action if and only if you are sure that the returned dictionary is small. Otherwise, use an appropriate chain of Spark's transformations and write the final result in a file by using the `saveAsTextFile` method.

Take action

The `take(num)` action returns a local python list of objects containing the first `num` elements of the considered RDD. The order of the elements in an RDD is consistent with the order of the elements in the file or collection that has been used to create the RDD.

The take action is based on the `take(num)` method of the `RDD` class.

Example

1. Create an RDD of integers containing the values [1,5,3,3,2];
2. Retrieve the first two values of the created RDD and store them in a local python list that is instantiated in the Driver.

```

1 ## Create an RDD of integers. Load the values 1, 5, 3, 3,2 in this RDD
2 inputList = [1,5,3,3,2]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Retrieve the first two elements of the inputRDD and store them in
6 ## a local python list
7 retrievedValues = inputRDD.take(2)

```

First action

The `first()` action returns a local python object containing the first element of the considered RDD. The order of the elements in an RDD is consistent with the order of the elements in the file or collection that has been used to create the RDD.

The first action is based on the `first()` method of the RDD class.

Notice that the only difference between `first()` and `take(1)` is given by the fact that `first()` returns a single element (the returned element is the first element of the RDD), while `take(1)` returns a list of elements containing one single element (the only element of the returned list is the first element of the RDD).

Top action

The `top(num)` action returns a local python list of objects containing the top num (largest) elements of the considered RDD. The ordering is the default one of class associated with the objects stored in the RDD (the descending order is used).

The top action is based on the `top(num)` method of the RDD class.

Example

1. Create an RDD of integers containing the values [1,5,3,4,2];
2. Retrieve the top-2 greatest values of the created RDD and store them in a local python list that is instantiated in the Driver.

```

1 ## Create an RDD of integers. Load the values 1, 5, 3, 4,2 in this RDD
2 inputList = [1,5,3,4,2]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Retrieve the top-2 elements of the inputRDD and store them in
6 ## a local python list
7 retrievedValues = inputRDD.top(2)

```

The `top(num, key)` action returns a local python list of objects containing the num largest elements of the considered RDD sorted by considering a user specified sorting function.

The top action is based on the `top(num, key)` method of the RDD class

- `num` is the number of elements to be selected;
- `key` is a function that is applied on each input element before comparing them.

The comparison between elements is based on the values returned by the invocations of this function.

i Example

1. Create an RDD of strings containing the values `['Paolo', 'Giovanni', 'Luca']`;
2. Retrieve the 2 longest names (longest strings) of the created RDD and store them in a local python list that is instantiated in the Driver.

```

1 ## Create an RDD of strings. Load the values 'Paolo', 'Giovanni', 'Luca'
2 ## in the RDD
3 inputList = ['Paolo','Giovanni','Luca']
4 inputRDD = sc.parallelize(inputList)
5
6 ## Retrieve the 2 longest names of the inputRDD and store them in
7 ## a local python list
8 retrievedValues = inputRDD.top(2,lambda s:len(s))

```

TakeOrdered action

The `takeOrdered(num)` action returns a local python list of objects containing the `num` smallest elements of the considered RDD. The ordering is the default one of class associated with the objects stored in the RDD (the ascending order is used).

The `takeOrdered` action is based on the `takeOrdered(num)` method of the RDD class.

i Example

1. Create an RDD of integers containing the values `[1,5,3,4,2]`;
2. Retrieve the 2 smallest values of the created RDD and store them in a local python list that is instantiated in the Driver.

```

1 ## Create an RDD of integers. Load the values 1, 5, 3, 4,2 in this RDD
2 inputList = [1,5,3,4,2]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Retrieve the 2 smallest elements of the inputRDD and store them in
6 ## a local python list
7 retrievedValues = inputRDD.takeOrdered(2)

```

The `takeOrdered(num, key)` action returns a local python list of objects containing the `num` smallest elements of the considered RDD sorted by considering a user specified sorting function.

The `takeOrdered` action is based on the `takeOrdered(num, key)` method of the `RDD` class

- `num` is the number of elements to be selected;
- `key` is a function that is applied on each input element before comparing them.

The comparison between elements is based on the values returned by the invocations of this function.

i Example

1. Create an `RDD` of strings containing the values `['Paolo', 'Giovanni', 'Luca']`;
2. Retrieve the 2 shortest names (shortest strings) of the created `RDD` and store them in a local python list that is instantiated in the Driver.

```

1 ## Create an RDD of strings. Load the values 'Paolo', 'Giovanni', 'Luca'
2 ## in the RDD
3 inputList = ['Paolo','Giovanni','Luca']
4 inputRDD = sc.parallelize(inputList)
5
6 ## Retrieve the 2 shortest names of the inputRDD and store them in
7 ## a local python list
8 retrievedValues = inputRDD.takeOrdered(2,lambda s:len(s))

```

TakeSample action

The `takeSample(withReplacement, num)` action returns a local python list of objects containing `num` random elements of the considered `RDD`.

The `takeSample` action is based on the `takeSample(withReplacement, num)` method of the `RDD` class, where `withReplacement` specifies if the random sample is with replacement (True) or not (False).

The `takeSample(withReplacement, num, seed)` method of the `RDD` class is used when the seed has to be set.

i Example

1. Create an `RDD` of integers containing the values `[1,5,3,3,2]`;
2. Retrieve randomly, without replacement, 2 values from the created `RDD` and store them in a local python list that is instantiated in the Driver.

```

1 ## Create an RDD of integers. Load the values 1, 5, 3, 3,2 in this RDD
2 inputList = [1,5,3,3,2]
3 inputRDD = sc.parallelize(inputList)
4
5 ## Retrieve randomly two elements of the inputRDD and store them in
6 ## a local python list
7 randomValues= inputRDD.takeSample(True, 2)

```

Reduce

Return a single python object obtained by combining all the objects of the input RDD by using a user provide function. The provided function must be associative and commutative, otherwise the result depends on the content of the partitions and the order used to analyze the elements of the RDD's partitions. The returned object and the ones of the input RDD are all instances of the same data type/class.

The reduce action is based on the `reduce(f)` method of the RDD class: a function `f` is passed to the reduce method

- Given two arbitrary input elements, `f` is used to combine them in one single value
- `f` is recursively invoked over the elements of the input RDD until the input values are reduced to one single value

Suppose L contains the list of elements of the input RDD. To compute the final element/value, the reduce action operates as follows

1. Apply the user specified function on a pair of elements e_1 and e_2 occurring in L and obtain a new element e_{new} ;
2. Remove the original elements e_1 and e_2 from L and then insert the element e_{new} in L ;
3. If L contains only one value, then return it as final result of the reduce action, otherwise, return to step 1.

Function f must be associative and commutative, so that the computation of the reduce action can be performed in parallel without problems, otherwise the result depends on how the input RDD is partitioned (i.e., for the functions that are not associative and commutative the output depends on how the RDD is split in partitions and how the content of each partition is analyzed).

Example 1

1. Create an RDD of integers containing the values [1,2,3,3];
2. Compute the sum of the values occurring in the RDD and store the result in a local python integer variable in the Driver.

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputListReduce = [1,2,3,3]
3 inputRDDReduce = sc.parallelize(inputListReduce)
4
5 ## Compute the sum of the values
6 sumValues = inputRDDReduce.reduce(lambda e1, e2: e1+e2)

```

`lambda e1, e2: e1+e2` This lambda function combines two input integer elements at a time and returns theirs sum.

Example 2

1. Create an RDD of integers containing the values [1,2,3,3];
2. Compute the maximum value occurring in the RDD and store the result in a local python integer variable in the Driver.

Solution 1

```

1 ## Define the function for the reduce action
2 def computeMax(v1,v2):
3     if v1>v2:
4         return v1
5     else:
6         return v2
7
8 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
9 inputListReduce = [1,2,3,3]
10 inputRDDReduce = sc.parallelize(inputListReduce)
11
12 ## Compute the maximum value
13 maxValue = inputRDDReduce.reduce(computeMax)

```

Solution 2

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputListReduce = [1,2,3,3]
3 inputRDDReduce = sc.parallelize(inputListReduce)
4
5 ## Compute the maximum value
6 maxValue = inputRDDReduce.reduce(lambda e1, e2: max(e1, e2))

```

Fold action

Return a single python object obtained by combining all the objects of the input RDD and a “zero” value by using a user provided function, which must be associative (otherwise the result depends on how the RDD is partitioned), but it is not required to be commutative. An initial neutral “zero” value is also specified.

The fold action is based on the `fold(zeroValue, op)` method of the `RDD` class. A function `op` is passed to the `fold` method; given two arbitrary input elements, `op` is

- used to combine them in one single value
- used to combine input elements with the “zero” value
- recursively invoked over the elements of the input RDD until the input values are reduced to one single value

The “zero” value is the neutral value for the used function `op` (i.e., “zero” combined with any value v by using `op` is equal to v).

Example 1

1. Create an RDD of strings containing the values `['This ','is ','a ','test']`;
2. Compute the concatenation of the values occurring in the RDD (from left to right) and store the result in a local python string variable in the Driver.

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputListFold = ['This ','is ','a ','test']
3 inputRDDFold = sc.parallelize(inputListFold)
4
5 ## Concatenate the input strings
6 finalString = inputRDDFold.fold('', lambda s1, s2: s1+s2)
```

Fold vs Reduce

1. Fold is characterized by the “zero” value;
2. Fold can be used to parallelize functions that are associative but non-commutative (e.g., concatenation of a list of strings).

Aggregate action

Return a single python object obtained by combining the objects of the RDD and an initial “zero” value by using two user provide functions. The provided functions must be associative, otherwise the result depends on how the RDD is partitioned. The returned objects and the ones of the input RDD can be instances of different classes (this is the main difference with respect to `reduce()` and `fold()`).

The aggregate action is based on the `aggregate(zeroValue,seqOp,combOp)` method of the `RDD` class. The input RDD contains objects of type `T` while the returned object is of type `U`, with `U` different from `T`.

- One function is needed for merging an element of type `T` with an element of type `U` to return a new element of type `U`: it is used to merge the elements of the input RDD and the accumulator of each partition;
- One function is needed for merging two elements of type `U` to return a new element of type `U`: it is used to merge two elements of type `U` obtained as partial results generated by two different partitions.

The `seqOp` function contains the code that is applied to combine the accumulator value (one accumulator for each partition) with the elements of each partition: one local result per partition is computed by recursively applying `seqOp`.

The `combOp` function contains the code that is applied to combine two elements of type `U` returned as partial results by two different partitions: the global final result is computed by recursively applying `combOp`.

How it works

Suppose that L contains the list of elements of the input RDD and this RDD is split in a set of partitions (i.e., a set of lists L_1, \dots, L_n). The aggregate action computes a partial result in each partition and then combines/merges the results. It operates as follows:

1. Aggregate the partial results in each partition, obtaining a set of partial results (of type U) $P = p_1, \dots, p_n$;
2. Apply the `combOp` function on a pair of elements p_1 and p_2 in P and obtain a new element p_{new} ;
3. Remove the original elements p_1 and p_2 from P and then insert the element p_{new} in P ;
4. If P contains only one value then return it as final result of the aggregate action. Otherwise, return to step 2.

Suppose that L_i is the list of elements on the i -th partition of the input RDD, and `zeroValue` is the initial zero value. To compute the partial result over the elements in L_i the aggregate action operates as follows

1. Set `accumulator` to `zeroValue` (`accumulator=zeroValue`);
2. Apply the `seqOp` function on `accumulator` and an elements e_j in L_i and update `accumulator` with the value returned by `seqOp`;
3. Remove the original elements e_j from L_i ;
4. If L_i is empty return `accumulator` as (final) partial result p_i of the i -th partition. Otherwise, return to step 2.

Note

1. Create an RDD of integers containing the values [1,2,3,3];
2. Compute both
 - the sum of the values occurring in the input RDD
 - and the number of elements of the input RDD
3. Store in a local python variable of the Driver the average computed over the values of the input RDD.

```

1 ## Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
2 inputListAggr = [1,2,3,3]
3 inRDD = sc.parallelize(inputListAggr)
4
5 ## Instantiate the zero value
6 ## We use a tuple containing two values:
7 ## (sum, number of represented elements)
8 zeroValue = (0,0)
9
10 ## Compute the sum of the elements in inputRDDAggr and count them
11 sumCount = inRDD.aggregate(
12     zeroValue,
13     lambda acc, e: (acc[0]+e, acc[1]+1),
14     lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1])
15 )

```

<code>zeroValue(0,0)</code>	Instantiate the “zero” value
<code>lambda acc, e: (acc[0]+e, acc[1]+1)</code>	Given a partition p of the input RDD, this is the function that is used to combine the elements of partition p with the accumulator of partition p
<code>acc</code>	It is a tuple object, initialized to the “zero” value
<code>e</code>	It is an integer
<code>lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1])</code>	This is the function that is used to combine the partial results emitted by the RDD partitions
<code>p1</code> and <code>p2</code>	These are tuple objects

```

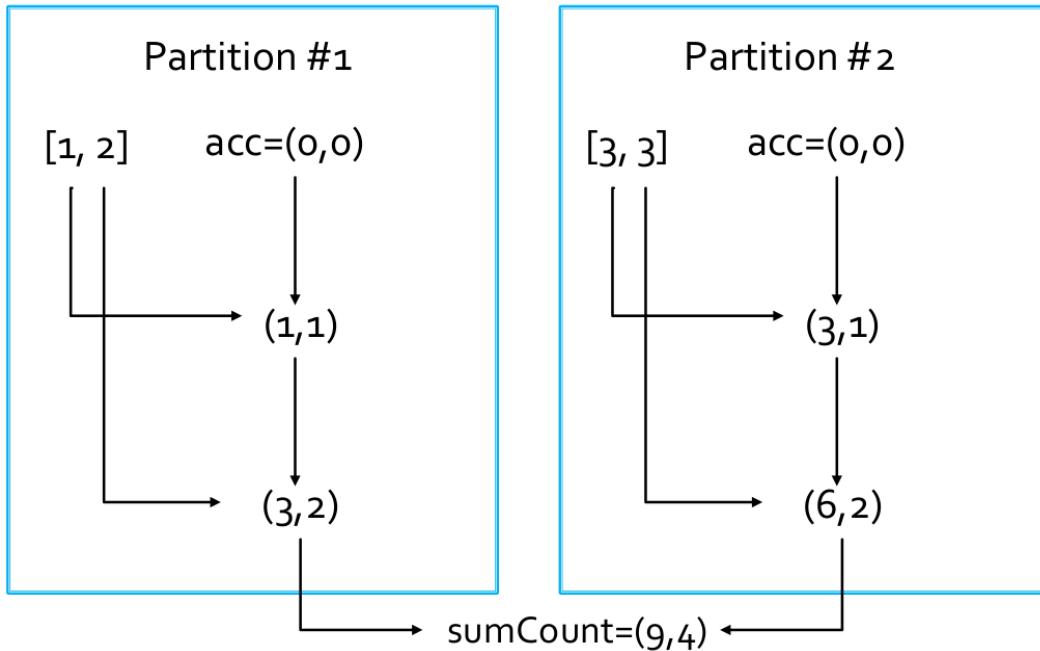
1 ## Compute the average value
2 myAvg = sumCount[0]/sumCount[1]
3
4 ## Print the average on the standard output of the driver
5 print('Average:', myAvg)

```

Aggregate action: visual simulation

- `inRDD = [1,2,3,3];`
- Suppose `inRDD` is split in the following two partitions, `[1,2]` and `[3,3]`.

Figure 13.2: Visual simulation of the aggregate action



Summary

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

Purposes

Action	Purpose
<code>collect()</code>	Return a python list containing all the elements of the RDD on which it is applied. The objects of the RDD and objects of the returned list are objects of the same class.
<code>count()</code>	Return the number of elements of the RDD.
<code>countByValue()</code>	Return a Map object containing the information about the number of times each element occurs in the RDD.
<code>take(num)</code>	Return a Python list containing the first <code>num</code> elements of the RDD. The objects of the RDD and objects of the returned list are objects of the same class.
<code>first()</code>	Return the first element of the RDD.
<code>top(num)</code>	Return a Python list containing the top <code>num</code> elements of the RDD based on the default sort order/comparator of the objects. The objects of the RDD and objects of the returned list are objects of the same class.
<code>takeSample(withReplacement, num)</code>	Return a (Python) List containing a random sample of size <code>n</code> of the RDD. The objects of the RDD and objects of the returned list are objects of the same class.
<code>takeSample(withReplacement, num, seed)</code>	Return a single Python object obtained by combining the values of the objects of the RDD by using a user provide function. The provided function must be associative and commutative. The object returned by the method and the objects of the RDD belong to the same class.
<code>reduce(f)</code>	Same as <code>reduce</code> but with the provided zero value.
<code>fold(zeroValue, op)</code>	Similar to <code>reduce()</code> but used to return a different type.
<code>Aggregate(zeroValue, seqOp, combOp)</code>	

Examples

Action	Example	Result
collect()	inputRDD.collect()	[1,2,3,3]
count()	inputRDD.count()	4
countByValue()	inputRDD.countByValue()	[(1,1), (2,1), (3,2)]
take(num)	inputRDD.take(2)	[1,2]
first()	inputRDD.first()	1
top(num)	inputRDD.top(2)	[3,3]
takeSample(withReplacement, num)	inputRDD.takeSample(False, 1)	Nondeterministic
takeSample(withReplacement, num, seed)	inputRDD.takeSample(False, 1)	9
reduce(f)	inputRDD.reduce(lambda e1, e2: e1+e2)	9
fold(zeroValue, op)	inputRDD.fold(0, lambda v1, v2: v1+v2)	9
Aggregate(zeroValue, seqOp, combOp)	inputRDD.aggregate(zeroValue, lambda acc, e: (acc[0]+e, acc[1]+1), lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))	(9, 4)
	Compute a pair of integers where the first one is the sum of the values of the RDD and the second the number of elements	

14 RDDs and key-value pairs

Spark supports also RDDs of key-value pairs. Key-value pairs in python are represented by means of python tuples:

- The first value is the key part of the pair
- The second value is the value part of the pair

RDDs of key-value pairs are sometimes called “pair RDDs”.

RDDs of key-value pairs are characterized by

- specific operations (e.g., `reduceByKey()`, `join()`), which analyze the content of one group (key) at a time;
- operations available for the standard RDDs (e.g., `filter()`, `map()`, `reduce()`).

Many applications are based on RDDs of key-value pairs: the operations available for RDDs of key-value pairs allow grouping data by key and performing computation by key (i.e., by group). The basic idea is similar to the one of the MapReduce-based programs in Hadoop, but there are more operations already available.

14.1 Creating RDDs of key-value pairs

RDDs of key-value pairs can be built

- From other RDDs by applying the `map()` or the `flatMap()` transformation on other RDDs;
- From a Python in-memory collection of tuple (key-value pairs) by using the `parallelize()` method of the `SparkContext` class.

Key-value pairs are represented as standard built-in Python tuples composed of two elements

- Key
- Value

14.2 RDDs of key-value pairs by using the Map transformation

The goal is to define an RDD of key-value pairs by using the map transformation: apply a function `f` on each element of the input RDD that returns one tuple for each input element. The new RDD of key-value pairs contains one tuple `y` for each element `x` of the input RDD.

The standard `map(f)` transformation is used, and the new RDD of key-value pairs contains one tuple `y` for each input element `x` of the input RDD ($y = f(x)$).

i Example

- Create an RDD from a textual file containing the first names of a list of users; each line of the file contains one first name;
- Create an RDD of key-value pairs containing a list of pairs (`first name, 1`).

```

1 ## Read the content of the input textual file
2 namesRDD = sc.textFile("first_names.txt")
3
4 ## Create an RDD of key-value pairs
5 nameOnePairRDD = namesRDD.map(lambda name: (name, 1))

```

`nameOnePairRDD` It contains key-value pairs (i.e., tuples) of type (string, integer)

14.3 RDDs of key-value pairs by using the flatMap transformation

Define an RDD of key-value pairs by using the flatMap transformation: apply a function f on each element of the input RDD that returns a list of tuples for each input element. The new PairRDD contains all the pairs obtained by applying f on each element x of the input RDD.

The standard `flatMap(f)` transformation is used, and the new RDD of key-value pairs contains the tuples returned by the execution of f on each element x of the input RDD.

$$[y] = f(x)$$

- Given a element x of the input RDD, f applied on x returns a list of pairs $[y]$;
- The new RDD is a list of pairs contains all the pairs of the returned list of pairs. It is not an RDD of lists.

$[y]$ can be the empty list.

i Example

1. Create an RDD from a textual file; each line of the file contains a set of words;
2. Create a PairRDD containing a list of pairs (`word, 1`): one pair for each word occurring in the input document (with repetitions).

Version 1

```

1 ## Define the function associated with the flatMap transformation
2 def wordsOnes(line):
3     pairs = []
4     for word in line.split(' '):
5         pairs.append( (word, 1))
6     return pairs
7
8 ## Read the content of the input textual file
9 linesRDD = sc.textFile("document.txt")
10
11 ## Create an RDD of key-value pairs based on the input document
12 ## One pair (word,1) for each input word
13 wordOnePairRDD = linesRDD.flatMap(wordsOnes)

```

Version 2

```

1 ## Read the content of the input textual file
2 linesRDD = sc.textFile("document.txt")
3
4 ## Create an RDD of key-value pairs based on the input document
5 ## One pair (word,1) for each input word
6 wordOnePairRDD = linesRDD.flatMap(
7     lambda line: map(lambda w: (w, 1), line.split(' '))
8 )

```

`map(lambda w: (w, 1), line.split(' '))`

14.4 RDDs of key-value pairs by using parallelize

Use the `parallelize` method to create an RDD of key-value pairs from a local python in-memory collection of tuples.

It is based on the standard `parallelize(c)` method of the `SparkContext` class: each element (tuple) of the local python collection becomes a key-value pair of the returned RDD.

i Example

Create an RDD from a local python list containing the following key-value pairs

- ("Paolo", 40)
- ("Giorgio", 22)
- ("Paolo", 35)

```

1 ## Create the local python list
2 nameAge = [
3     ("Paolo",40),
4     ("Giorgio",22),
5     ("Paolo",35)
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 nameAgePairRDD = sc.parallelize(nameAge)

```

nameAge This is a local in-memory python list of key-value pairs (tuples), that is stored in the main memory of the Driver.

nameAgePairRDD This is an RDD or key-value pairs based on the content of the local in-memory python list. The RDD is stored in the “distributed” main memory of the cluster servers

14.5 Transformations on RDDs of key-value pairs

All the standard transformations can be applied, where the specified functions operate on tuples, but also specific transformations are available (e.g., `reduceByKey()`, `groupByKey()`, `mapValues()`, `join()`).

ReduceByKey transformation

The goal is to create a new RDD of key-value pairs where there is one pair for each distinct key k of the input RDD of key-value pairs:

- The value associated with key k in the new RDD of key-value pairs is computed by applying a function f on the values associated with k in the input RDD of key-value pairs; the function f must be associative and commutative, otherwise the result depends on how data are partitioned and analyzed;
- The data type of the new RDD of key-value pairs is the same of the input RDD of key-value pairs.

The `reduceByKey` transformation is based on the `reduceByKey(f)` method of the `RDD` class. A function f is passed to the `reduceByKey` method

- Given the values of two input pairs, f is used to combine them in one single value;
- f is recursively invoked over the values of the pairs associated with one key at a time until the input values associated with one key are reduced to one single value.

The retuned RDD contains a number of key-value pairs equal to the number of distinct keys in the input key-value pair RDD.

Similarly to the `reduce()` action, the `reduceByKey()` transformation aggregate values, however `reduceByKey()` is executed on RDDs of key-value pairs and returns a set of key-value pairs, while

`reduce()` is executed on an RDD and returns one single value (stored in a local python variable). Moreover, `reduceByKey()` is a transformation, and so it is executed lazily and its result is stored in another RDD, whereas `reduce()` is an action.

A shuffle operation is executed for computing the result of the `reduceByKey()` transformation. The result/value for each group/key is computed from data stored in different input partitions.

i Example

1. Create an RDD from a local python list containing the pairs, where the key is the first name of a user and the value is his/her age
 - ("Paolo", 40)
 - ("Giorgio", 22)
 - ("Paolo", 35)
2. Create a new RDD of key-value pairs containing one pair for each name. In the returned RDD, associate each name with the age of the youngest user with that name.

```

1 ## Create the local python list
2 nameAge = [
3     ("Paolo",40),
4     ("Giorgio",22),
5     ("Paolo",35)
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 nameAgePairRDD = sc.parallelize(nameAge)
10
11 ## Select for each name the lowest age value
12 youngestPairRDD= nameAgePairRDD.reduceByKey(lambda age1, age2: min(age1, age2))

```

`youngestPairRDD` The returned RDD of key-value pairs contains one pair for each distinct input key (i.e., for each distinct name in this example)

FoldByKey transformation

The `foldByKey()` has the same goal of the `reduceByKey()` transformation, however

- It is characterized also by a “zero” value
- Functions must be associative but are not required to be commutative

The `foldByKey` transformation is based on the `foldByKey(zeroValue, op)` method of the `RDD` class. A function `op` is passed to the `fold` method:

- Given values of two input pairs, `op` is used to combine them in one single value
- `op` is also used to combine input values with the “zero” value

- `op` is recursively invoked over the values of the pairs associated with one key at a time until the input values are reduced to one single value

The “zero” value is the neutral value for the used function `op` (i.e., “zero” combined with any value v by using `op` is equal to v).

A shuffle operation is executed for computing the result of the `foldByKey()` transformation. The result/value for each group/key is computed from data stored in different input partitions.

Example

1. Create an RDD from a local python list containing the pairs, where the key is the first name of a user and the value is a message published by him/her
 - ("Paolo", "Message1")
 - ("Giorgio", "Message2")
 - ("Paolo", "Message3")
2. Create a new RDD of key-value pairs containing one pair for each name. In the returned RDD, associate each name the concatenation of its messages (preserving the order of the messages in the input RDD).

```

1 ## Create the local python list
2 nameMess = [
3     ("Paolo","Message1"),
4     ("Giorgio","Message2"),
5     ("Paolo","Message3")
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 nameMessPairRDD = sc.parallelize(nameMess)
10
11 ## Concatenate the messages of each user
12 concatPairRDD= nameMessPairRDD.foldByKey(' ', lambda m1, m2: m1+m2)
```

CombineByKey transformation

The goal is to create a new RDD of key-value pairs where there is one pair for each distinct key k of the input RDD of key-value pairs. The value associated with the key k in the new RDD of key-value pairs is computed by applying user-provided functions on the values associated with k in the input RDD of key-value pairs: the user-provided function must be associative, otherwise the result depends how data are partitioned and analyzed.

The data type of the new RDD of key-value pairs can be different with respect to the data type of the input RDD of key-value pairs.

The `combineByKey` transformation is based on the `combineByKey(createCombiner, mergeValue, mergeCombiner)` method of the `RDD` class

- The values of the input RDD of pairs are of type V
- The values of the returned RDD of pairs are of type U
- The type of the keys is K for both RDDs of pairs

The `createCombiner` function contains the code that is used to transform a single value (type V) of the input RDD of key-value pairs into a value of the data type (type U) of the output RDD of key-value pairs. It is used to transform the first value of each key in each partition to a value of type U .

The `mergeValue` function contains the code that is used to combine one value of type U with one value of type V : it is used in each partition to combine the initial values (type V) of each key with the intermediate ones (type U) of each key.

The `mergeCombiner` function contains the code that is used to combine two values of type U : it is used to combine intermediate values of each key returned by the analysis of different partitions.

The `combineByKey` function is more general than `reduceByKey` and `foldByKey` because the data types of the values of the input and the returned RDD of pairs can be different; for this reason, more functions must be implemented in this case.

A shuffle operation is executed for computing the result of the `combineByKey()` transformation: the result/value for each group/key is computed from data stored in different input partitions.

Example

1. Create an RDD from a local python list containing the the following pairs: the key is the first name of a user and the value is his/her age
 - ("Paolo", 40)
 - ("Giorgio", 22)
 - ("Paolo", 35)
2. Store the results in an output HDFS folder. The output contains one line for each name followed by the average age of the users with that name

```

1  ## Create the local python list
2  nameAge = [
3      ("Paolo",40),
4      ("Giorgio",22),
5      ("Paolo",35)
6  ]
7
8  ## Create the RDD of pairs from the local collection
9  nameAgePairRDD = sc.parallelize(nameAge)
10
11 ## Compute the sum of ages and
12 ## the number of input pairs for each name (key)
13 sumNumPerNamePairRDD = nameAgePairRDD.combineByKey(
14     lambda inputElem: (inputElem, 1),
15     lambda intermediateElem, inputElem: (
16         intermediateElem[0]+inputElem,
17         intermediateElem[1]+1
18     ),
19     lambda intermediateElem1, intermediateElem2: (
20         intermediateElem1[0]+intermediateElem2[0],
21         intermediateElem1[1]+intermediateElem2[1]
22     )
23 )
24
25 ## Compute the average for each name
26 avgPerNamePairRDD = sumNumPerNamePairRDD.map(
27     lambda pair: (pair[0], pair[1][0]/pair[1][1])
28 )
29
30 ## Store the result in an output folder
31 avgPerNamePairRDD.saveAsTextFile(outputPath)

```

-
- | | |
|--|--|
| lambda
inputElem:
lambda
intermediateElem, inputElem:
lambda
intermediateElem1, intermediateElem2:
lambda pair: | Given an input value (an age), it returns a tuple containing (age,1)
Given an input value (an age) and an intermediate value (<i>sum ages, num represented values</i>), it combines them and returns a new updated tuple (<i>sum ages, num represented values</i>)
Given two intermediate result tuples (<i>sum ages, num represented values</i>), it combines them and returns a new updated tuple (<i>sum ages, num represented values</i>)
Compute the average age for each key (i.e., for each name) by combining <i>sum ages</i> and <i>num represented values</i> . Each input pair is characterized by a value that is a tuple containing (<i>sum ages, num represented values</i>). |
|--|--|

GroupByKey transformation

The goal is to create a new RDD of key-value pairs where there is one pair for each distinct key k of the input RDD of key-value pairs: the value associated with key k in the new RDD of key-value pairs is the list of values associated with k in the input RDD of key-value pairs.

The groupByKey transformation is based on the `groupByKey()` method of the `RDD` class.

If grouping values per key to perform then an aggregation such as sum or average over the values of each key then `groupByKey` is not the right choice: `reduceByKey`, `aggregateByKey`, or `combineByKey` provide better performances for associative and commutative aggregations; `groupByKey` is useful if an aggregation or compute a function that is not associative must be applied.

A shuffle operation is executed for computing the result of the `groupByKey()` transformation: each group/key is associated with/is composed of values which are stored in different partitions of the input RDD.

Example

1. Create an RDD from a local python list containing the following pairs: the key is the first name of a user and the value is his/her age.
 - ("Paolo", 40)
 - ("Giorgio", 22)
 - ("Paolo", 35)
2. Store the results in an output HDFS folder. The output contains one line for each name followed by the ages of all the users with that name.

```

1 ## Create the local python list
2 nameAge = [
3     ("Paolo",40),
4     ("Giorgio",22),
5     ("Paolo",35)
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 nameAgePairRDD = sc.parallelize(nameAge)
10
11 ## Create one group for each name with the list of associated ages
12 agesPerNamePairRDD = nameAgePairRDD.groupByKey()
13
14 ## Store the result in an output folder
15 agesPerNamePairRDD\
16     .mapValues(lambda listValues: list(listValues))\
17     .saveAsTextFile(outputPath)

```

`agesPerNamePairRDD` In this RDD of key-value pairs each tuple is composed of a string (key of the pair) and a collection of integers (the value of the pair - a `ResultIterable` object)

`.mapValues(lambda listValues: list(listValues))` This part is used to format the content of the value part of each pair before storing the result in the output folder: this transforms a `ResultIterable` object to a Python list. Without this map the output will contain the pointers to `ResultIterable` objects instead of a readable list of integer values

MapValues transformation

The goal is to apply a function f over the value of each pair of an input RDD or key-value pairs and return a new RDD of key-value pairs: one pair is created in the returned RDD for each input pair

- The key of the created pair is equal to the key of the input pair
- The value of the created pair is obtained by applying the function f on the value of the input pair

The data type of the values of the new RDD of key-value pairs can be different from the data type of the values of the input RDD of key-value pairs. The data type of the key is the same.

The `mapValues` transformation is based on the `mapValues(f)` method of the `RDD` class: a function `f` is passed to the `mapValues` method, where `f` contains the code that is applied to transform each input value into the a new value that is stored in the `RDD` of key-value pairs. The retuned `RDD` of pairs contains a number of key-value pairs equal to the number of key-value pairs of the input `RDD` of pairs (the key part is not changed).

i Example

1. Create an `RDD` from a local python list containing the following pairs: the key is the first name of a user and the value is his/her age.
 - ("Paolo", 40)
 - ("Giorgio", 22)
 - ("Paolo", 35)
2. Increase the age of each user (+1 year) and store the result in the HDFS file system, one output line per user.

```

1 ## Create the local python list
2 nameAge = [
3     ("Paolo",40),
4     ("Giorgio",22),
5     ("Paolo",35)
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 nameAgePairRDD = sc.parallelize(nameAge)
10
11 ## Increment age of all users
12 plusOnePairRDD = nameAgePairRDD.mapValues(lambda age: age+1)
13
14 ## Save the result on disk
15 plusOnePairRDD.saveAsTextFile(outputPath)

```

FlatMapValues transformation

The goal is to apply a function f over the value of each pair of an input RDD or key-value pairs and return a new RDD of key-value pairs (f returns a list of values for each input value). A list of pairs is inserted in the returned RDD for each input pair

- The key of the created pairs is equal to the key of the input pair
- The values of the created pairs are obtained by applying the function f on the value of the input pair

The data type of the values of the new RDD of key-value pairs can be different from the data type of the values of the input RDD of key-value pairs. The data type of the key is the same.

The flatMapValues transformation is based on `flatMapValues(f)` method of the RDD class: a function `f` is passed to the `mapValues` method, where `f` contains the code that is applied to transform each input value into a set of new values that are stored in the new RDD of key-value pairs. The keys of the input pairs are not changed.

Example

1. Create an RDD from a local python list containing the pairs
 - ("Sentence#1", "Sentence test")
 - ("Sentence#2", "Sentence test number 2")
 - ("Sentence#3", "Sentence test number 3")
2. Select the words of each sentence and store in the HDFS file system one pair (sentenceId, word) per line.

```

1 ## Create the local python list
2 sentences = [
3     ("Sentence#1", "Sentence test"),
4     ("Sentence#2", "Sentence test number 2"),
5     ("Sentence#3", "Sentence test number 3")
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 sentPairRDD = sc.parallelize(sentences)
10
11 ## "Extract" words from each sentence
12 sentIdWord = sentPairRDD.flatMapValues(lambda s: s.split(' '))
13
14 ## Save the result on disk
15 sentIdWord.saveAsTextFile(outputPath)

```

Keys transformation

The goal is to return the list of keys of the input RDD of pairs and store them in a new RDD. The returned RDD is not an RDD of key-value pairs, instead it is a standard RDD of single elements, with duplicate keys not removed.

The keys transformation is based on the `keys()` method of the `RDD` class.

Example

1. Create an RDD from a local python list containing the following pairs: the key is the first name of a user and the value is his/her age
 - ("Paolo", 40)
 - ("Giorgio", 22)
 - ("Paolo", 35)
2. Store the names of the input users in an output HDFS folder. The output contains one name per line (duplicate names are removed).

```

1 ## Create the local python list
2 nameAge = [
3     ("Paolo",40),
4     ("Giorgio",22),
5     ("Paolo",35)
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 nameAgePairRDD = sc.parallelize(nameAge)
10
11 ## Select the key part of the input RDD of key-value pairs
12 namesRDD = nameAgePairRDD.keys().distinct()
13
14 ## Store the result in an output folder
15 namesRDD.saveAsTextFile(outputPath)

```

Values transformation

The goal is to return the list of values of the input RDD of pairs and store them in a new RDD. The returned RDD is not an RDD of key-value pairs, instead it is a standard RDD of single elements, with duplicate values are not removed.

The values transformation is based on the `values()` method of the RDD class.

Example

1. Create an RDD from a local python list containing the pairs: the key is the first name of a user and the value is his/her age
 - ("Paolo", 40)
 - ("Giorgio", 22)
 - ("Paolo", 22)
2. Store the ages of the input users in an output HDFS folder, containing one age per line and duplicate ages/values are not removed.

```

1 ## Create the local python list
2 nameAge = [
3     ("Paolo",40),
4     ("Giorgio",22),
5     ("Paolo",22)
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 nameAgePairRDD = sc.parallelize(nameAge)
10
11 ## Select the value part of the input RDD of key-value pairs
12 agesRDD = nameAgePairRDD.values()
13
14 ## Store the result in an output folder
15 agesRDD.saveAsTextFile(outputPath)

```

SortByKey transformation

The goal is to return a new RDD of key-value pairs obtained by sorting, in ascending order, the pairs of the input RDD by key (notice that the final order is related to the default sorting function of the data type of the input keys). The content of the new RDD of key-value pairs is the same of the input RDD but the pairs are sorted by key in the new returned RDD.

The `sortByKey` transformation is based on the `sortByKey()` method of the `RDD` class (pairs are sorted by key in ascending order). The `sortByKey(ascending)` method of the `RDD` class is also available: this method allows to specify if the sort order is ascending or descending by means of a Boolean parameter (`True` for ascending, `False` for descending).

A shuffle operation is executed for computing the result of the `sortByKey()` transformation, since pairs from different partitions of the input RDD must be compared to sort the input pairs by key.

i Example

1. Create an RDD from a local python list containing the pairs: the key is the first name of a user and the value is his/her age

- ("Paolo", 40)
- ("Giorgio", 22)
- ("Paolo", 35)

2. Sort the users by name and store the result in the HDFS file system.

```
1 ## Create the local python list
2 nameAge = [
3     ("Paolo",40),
4     ("Giorgio",22),
5     ("Paolo",35)
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 nameAgePairRDD = sc.parallelize(nameAge)
10
11 ## Sort by name the content of the input RDD of key-value pairs
12 sortedNameAgePairRDD = nameAgePairRDD.sortByKey()
13
14 ## Store the result in an output folder
15 sortedNameAgePairRDD.saveAsTextFile(outputPath)
```

Summary

All the examples reported in the following tables are applied on an RDD of pairs containing the following tuples (pairs): [("k1", 2), ("k3", 4), ("k3", 6)].

- The key of each tuple is a string
- The value of each tuple is an integer

Purposes

Transformation	Purpose
<code>reduceByKey(f)</code>	Return an RDD of pairs containing one pair for each key of the input RDD of pairs. The value of each pair of the new RDD of pairs is obtained by combining the values of the input RDD associated with the same key. The input RDD of pairs and the new RDD of pairs have the same data type.
<code>foldByKey(zeroValue, op)</code>	Similar to the <code>reduceByKey()</code> transformation, however <code>foldByKey()</code> is characterized also by a zero value.
<code>combineByKey(createCombiner, mergeValue, mergeCombiner)</code>	Return an RDD of key-value pairs containing one pair for each key of the input RDD of pairs. The value of each pair of the new RDD is obtained by combining the values of the input RDD associated with the same key. The values of the input RDD of pairs and the values of the new (returned) RDD of pairs can be characterized by different data types.
<code>groupByKey()</code>	Return an RDD of pairs containing one pair for each key of the input RDD of pairs. The value of each pair of the new RDD of pairs is a collection containing all the values of the input RDD associated with one of the input keys.
<code>mapValues(f)</code>	Apply a function over each pair of an RDD of pairs and return a new RDD of pairs. The applied function returns one pair for each pair of the input RDD of pairs. The function is applied only on the value part without changing the key. The values of the input RDD and the values of new RDD can have different data types.
<code>flatMapValues(f)</code>	Apply a function over each pair of an RDD of pairs and return a new RDD of pairs. The applied function returns a set of pairs (from 0 to many) for each pair of the input RDD of pairs. The function is applied only on the value part without changing the key. The values of the input RDD and the values of new RDD can have different data types.
<code>keys()</code>	Return an RDD containing the keys of the input pairRDD.
<code>values()</code>	Return an RDD containing the values of the input pairRDD.
<code>sortByKey()</code>	Return a PairRDD sorted by key. The input PairRDD and the new PairRDD have the same data type.

Examples

Transformation	Example	Result
<code>reduceByKey(f)</code>	<code>reduceByKey(1lambda v1, v2: v1+v2)</code> Sum values per key	<code>[("k1", 2), ("k3", 10)]</code>
<code>foldByKey(zeroValue, op)</code>	<code>foldByKey(0,1ambda v1, v2: v1+v2)</code> Sum values per key. The zero value is 0	<code>[("k1", 2), ("k3", 10)]</code>
<code>combineByKey(createCombiner, mergeValue, mergeCombiner)</code>	<code>combineByKey(lambda e: (e, 1), lambda c, e: (c [0]+e, c [1]+1), lambda c1, c2: (c1[0]+c2[0], c1 [1]+c2[1]))</code> Sum values by key and count the number of pairs by key in one single step	<code>[("k1", (2, 1)), ("k3", (10, 2))]</code>
<code>groupByKey() mapValues(f)</code>	<code>groupByKey()</code> <code>mapValues(1ambda v: v+1)</code> Increment the value part by 1	<code>[("k1", [2]), ("k3", [4, 6])] [("k1", 3), ("k3", 5), ("k3", 7)]</code>
<code>flatMapValues(f)</code>	<code>flatMapValues(1ambda v: list(range(v, 6)))</code>	<code>[("k1", 2), ("k1", 3), ("k1", 4), ("k1", 5), ("k3", 4), ("k3", 5)]</code>
<code>keys() values() sortByKey()</code>	<code>keys()</code> <code>values()</code> <code>sortByKey()</code>	<code>[["k1", "k3", "k3"]] [2, 4, 6] [("k1", 2), ("k3", 4), ("k3", 6)]</code>

14.6 RDD-based programming

Transformations on two RDDs of key-value pairs

Spark supports also some transformations that are applied on two RDDs of key-value pairs at the same time (e.g., `subtractByKey`, `join`, `coGroup`).

SubtractByKey transformation

The goal is to create a new RDD of key-value pairs containing only the pairs of the first input RDD of pairs associated with a key that is not appearing as key in the pairs of the second input RDD or pairs. The two input RDD of pairs must have the same type of keys, but the data type of the values can be different. The data type of the new RDD of pairs is the same of the first input RDD of pairs.

The `subtractByKey` transformation is based on the `subtractByKey(other)` method of the `RDD` class. The two input RDDs of pairs analyzed by `subtractByKey` are the one which the method is invoked on and the one passed as parameter (i.e., `other`).

A shuffle operation is executed for computing the result of the `subtractByKey()` transformation, since keys from different partitions of the two input RDDs must be compared.

Example

1. Create two RDDs of key-value pairs from two local python lists

- First list – Profiles of the users of a blog (`username,age`):
`[("PaoloG",40),("Giorgio",22),("PaoloB",35)]`
- Second list – Banned users (`username,motivation`):
`[("PaoloB","spam"),("Giorgio","Vandalism")]`

2. Create a new RDD of pairs containing only the profiles of the non-banned users.

```

1 ## Create the first local python list
2 profiles = [("PaoloG",40),("Giorgio",22),("PaoloB",35)]
3
4 ## Create the RDD of pairs from the profiles local list
5 profilesPairRDD = sc.parallelize(profiles)
6
7 ## Create the second local python list
8 banned = [("PaoloB","spam"),("Giorgio","Vandalism")]
9
10 ## Create the RDD of pairs from the banned local list
11 bannedPairRDD = sc.parallelize(banned)
12
13 ## Select the profiles of the "good" users
14 selectedProfiles = profilesPairRDD.subtractByKey(bannedPairRDD)

```

Join transformation

The goal is to Join the key-value pairs of two RDDs of key-value pairs based on the value of the key of the pairs: each pair of the input RDD of pairs is combined with all the pairs of the other RDD of pairs with the same key. The new RDD of key-value pairs has the same key data type of the input RDDs of pairs, and has a tuple as value (the pair of values of the two joined input pairs). The two input RDDs of key-value pairs must have the same type of keys, but the data types of the values can be different.

The join transformation is based on the `join(other)` method of the RDD class: the two input RDDs of pairs analyzed by join are the one which the method is invoked on and the one passed as parameter (i.e., `other`).

A shuffle operation is executed for computing the result of the `join()` transformation, since keys from different partitions of the two input RDDs must be compared and values from different partitions must be retrieved.

Example

1. Create two RDDs of key-value pairs from two local python lists
 - First list – List of questions (QuestionId, Text of the question): `[(1,"What is ...?"),(2,"Who is ...?")]`
 - Second list – List of answers (QuestionId, Text of the answer): `[(1,"It is a car"),(1,"It is a byke"),(2,"She is Jenny")]`
2. Create a new RDD of pairs to associate each question with its answers: one pair for each possible pair (question-answer)

```

1 ## Create the first local Python list
2 questions= [(1,"What is ...?"),(2,"Who is ...?")]
3
4 ## Create the RDD of pairs from the local list
5 questionsPairRDD = sc.parallelize(questions)
6
7 ## Create the second local python list
8 answers = [(1,"It is a car"),(1,"It is a byke"),(2,"She is Jenny")]
9
10 ## Create the RDD of pairs from the local list
11 answersPairRDD = sc.parallelize(answers)
12
13 ## Join questions with answers
14 joinPairRDD = questionsPairRDD.join(answersPairRDD)

```

`joinPairRDD`

The key part of the returned RDD of pairs is an integer number. The value part of the returned RDD of pairs is a tuple containing two values: (question,answer).

CoGroup transformation

The goal is to associate each key k of the two input RDDs of key-value pairs with

- The list of values associated with k in the first input RDD of pairs
- The list of values associated with k in the second input RDD of pairs

The new RDD of key-value pairs has the same key data type of the two input RDDs of pairs, and has a tuple as value (the two lists of values of the two input RDDs of pairs). The two input RDDs of key-value pairs must have the same type of keys, but the data types of the values can be different.

The cogroup transformation is based on the `cogroup(other)` method of the `RDD` class: the two input RDDs of pairs analyzed by cogroup are the one which the method is invoked on and the one passed as parameter (i.e., `other`).

A shuffle operation is executed for computing the result of the `cogroup()` transformation, since keys from different partitions of the two input RDDs must be compared and values from different partitions must be retrieved.

Example

1. Create two RDDs of key-value pairs from two local python lists

- First list – List of liked movies (`userId,likedMovies`): `[(1,"Star Trek"),(1,"Forrest Gump"),(2,"Forrest Gump")]`
- Second list – List of liked directors (`userId,likedDirector`): `[(1,"Woody Allen"),(2,"Quentin Tarantino"),(2,"Alfred Hitchcock")]`

2. Create a new RDD of pairs containing one pair for each `userId` (key) associated with

- The list of liked movies
- The list of liked directors

Inputs - `[(1,"Star Trek"),(1,"Forrest Gump"),(2,"Forrest Gump")]` - `[(1,"Woody Allen"),(2,"Quentin Tarantino"),(2,"Alfred Hitchcock")]`

Output - `(1,(["Star Trek","Forrest Gump"],["Woody Allen"]))` - `(2,(["Forrest Gump"],["Quentin Tarantino","Alfred Hitchcock"]))`

```

1 ## Create the first local python list
2 movies= [
3     (1,"Star Trek"),
4     (1,"Forrest Gump"),
5     (2,"Forrest Gump")
6 ]
7
8 ## Create the RDD of pairs from the first local list
9 moviesPairRDD = sc.parallelize(movies)
10
11 ## Create the second local python list
12 directors = [
13     (1,"Woody Allen"),
14     (2,"Quentin Tarantino"),
15     (2,"Alfred Hitchcock")
16 ]
17
18 ## Create the RDD of pairs from the second local list
19 directorsPairRDD = sc.parallelize(directors)
20
21 ## Cogroup movies and directors per user
22 cogroupPairRDD = moviesPairRDD.cogroup(directorsPairRDD)

```

cogroupPairRDD

Notice that the value part of the returned tuples is a tuple containing two lists: the first value contains the list of movies (iterable) liked by a user; the second value contains the list of directors (iterable) liked by a user.

Summary

All the examples reported in the following tables are applied on the following two RDDs of key-value pairs

- inputRDD1: [('k1',2),('k3',4),('k3',6)]
- inputRDD2: [('k3',9)]

Purposes

Transformation	Purpose
<code>subtractByKey(other)</code>	Return a new RDD of key-value pairs. The returned pairs are those of input RDD on which the method is invoked such that the key part does not occur in the keys of the RDD that is passed as parameter. The values are not considered to take the decision.
<code>join(other)</code>	Return a new RDD of pairs corresponding to join of the two input RDDs. The join is based on the value of the key. For each key k in one of the two input RDDs of pairs, return a pair $(k, tuple)$, where tuple contains:
<code>cogroup(other)</code>	<ul style="list-style-type: none"> • the list (iterable) of values of the first input RDD associated with key k; • the list (iterable) of values of the second input RDD associated with key k.

Examples

Transformation	Example	Result
subtractByKey(other)	inputRDD1.subtractByKey(inputRDD2)	[('k1', 2)]
join(other)	inputRDD1.join(inputRDD2)	[('k3', (4, 9)), ('k3', (6, 9))]
cogroup(other)	inputRDD1.cogroup(inputRDD2)	[('k1', ([2], [])), ('k3', ([4, 6], [9])))]

14.7 Actions on RDDs of key-value pairs

Spark supports also some specific actions on RDDs of key-value pairs (e.g., `countByKey`, `collectAsMap`, `lookup`).

CountByKey action

The `countByKey` action returns a local python dictionary containing the information about the number of elements associated with each key in the input RDD of key-value pairs (i.e., the number of times each key occurs in the input RDD).

Warning

Pay attention to the number of distinct keys of the input RDD of pairs: if the number of distinct keys is large, the result of the action cannot be stored in a local variable of the Driver.

The `countByKey` action is based on the `countByKey()` method of the `RDD` class. Notice that data are sent on the network to compute the final result.

Example

1. Create an RDD of pairs from the following python list (each pair contains a movie and the rating given by someone to that movie): `[("Forrest Gump",4),("Star Trek",5),("Forrest Gump",3)]`;
2. Compute the number of ratings for each movie.

```

1 ## Create the local python list
2 movieRating= [
3     ("Forrest Gump",4),
4     ("Star Trek",5),
5     ("Forrest Gump",3)
6 ]
7
8 ## Create the RDD of pairs from the local collection
9 movieRatingRDD = sc.parallelize(movieRating)
10
11 ## Compute the number of rating for each movie
12 movieNumRatings = movieRatingRDD.countByKey()
13
14 ## Print the result on the standard output
15 print(movieNumRatings)
```

`movieNumRatings`

Pay attention to the size of the returned local python dictionary (i.e., in this case, the number of distinct movies).

CollectAsMap action

The collectAsMap action returns a local dictionary containing the same pairs of the considered input RDD of pairs. Pay attention to the size of the returned RDD: data are sent on the network.

The collectAsMap action is based on the `collectAsMap()` method of the RDD class.

Warning

Pay attention that the collectAsMap action returns a dictionary object, and a dictionary cannot contain duplicate keys: each key can be associated with at most one value, and if the input RDD of pairs contains more than one pair with the same key, only one of those pairs is stored in the returned local python dictionary (usually, the last one occurring in the input RDD of pairs).

Use collectAsMap only if sure that each key appears only once in the input RDD of key-value pairs.

The `collectAsMap()` method returns a local dictionary while `collect()` return a list of key-value pairs (i.e., a list of tuples). Notice that the list of pairs returned by `collect()` can contain more than one pair associated with the same key.

Example

1. Create an RDD of pairs from the following python list (each pair contains a userId and the name of the user): `[("User1","Paolo"),("User2","Luca"),("User3","Daniele")]`;
2. Retrieve the pairs of the created RDD of pairs and store them in a local python dictionary that is instantiated in the Driver.

```

1 ## Create the local python list
2 users = [
3     ("User1","Paolo"),
4     ("User2","Luca"),
5     ("User3","Daniele")
6 ]
7
8 #Create the RDD of pairs from the local list
9 usersRDD = sc.parallelize(users)
10
11 ## Retrieve the content of usersRDD and store it in a
12 ## local python dictionary
13 retrievedPairs = usersRDD.collectAsMap()
14
15 ## Print the result on the standard output
16 print(retrievedPairs)
```

`retrievedPairs`

Pay attention to the size of the returned local python dictionary (i.e., in this case, the number of distinct users).

Lookup action

The `lookup(k)` action returns a local python list containing the values of the pairs of the input RDD associated with the key *k* specified as parameter.

The lookup action is based on the `lookup(key)` method of the RDD class.

Example

1. Create an RDD of pairs from the following python list (each pair contains a movie and the rating given by someone to that movie): `[("Forrest Gump",4),("Star Trek",5),("Forrest Gump",3)]`
2. Retrieve the ratings associated with the movie “Forrest Gump” and store them in a local python list in the Driver.

```

1 ## Create the local python list
2 movieRating= [("Forrest Gump",4),("Star Trek",5),("Forrest Gump",3)]
3
4 ## Create the RDD of pairs from the local collection
5 movieRatingRDD = sc.parallelize(movieRating)
6
7 ## Select the ratings associated with "Forrest Gump"
8 movieRatings = movieRatingRDD.lookup("Forrest Gump")
9
10 ## Print the result on the standard output
11 print(movieRatings)

```

`movieRatings`

Pay attention to the size of the returned local list (i.e., in this case, the number of ratings associated with “Forrest Gump”).

Summary

All the examples reported in the following tables are applied on the following RDD of key-value pairs:

- `inputRDD: [('k1',2),('k3',4),('k3',6)]`

Purposes

Transformation	Purpose
<code>countByKey()</code>	Return a local python dictionary containing the number of elements in the input RDD for each key of the input RDD of pairs.
<code>collectAsMap()</code>	Return a local python dictionary containing the pairs of the input RDD of pairs.
<code>lookup(key)</code>	Return a local python list containing all the values associated with the key specified as parameter.

Examples

Transformation	Example	Result
countByKey() collectAsMap()	inputRDD.countByKey() inputRDD.collectAsMap() Or {'k1' : 2}, {'k3' : 4} Depending on the order of the pairs in the input RDD of pairs [4, 6]	{('k1', 1), ('k3', 2)} {('k1', 2), ('k3', 6)}
lookup(key)	inputRDD.lookup('k3')	[4, 6]

15 RDD of numbers

Spark provides specific actions for RDD containing numerical values (integers or floats).

RDDs of numbers can be created by using the standard methods

- parallelize
- transformations that return an RDD of numbers

The following specific actions are also available on this type of RDDs

- `sum()`
- `mean()`
- `stdev()`
- `variance()`
- `max()`
- `min()`

15.1 Summary

All the examples reported in the following are applied on `inputRDD` that is an RDD containing the following double values: [1.5,3.5,2.0]

Action	Purpose	Example	Result
<code>sum()</code>	Return the sum of the values of the input RDD.	<code>inputRDD.sum()</code>	7.0
<code>mean()</code>	Return the mean computed over the values of the input RDD.	<code>inputRDD.mean()</code>	2.3333
<code>stdev()</code>	Return the standard deviation computed over the values of the input RDD.	<code>inputRDD.stdev()</code>	0.8498
<code>variance()</code>	Return the variance computed over the values of the input RDD.	<code>inputRDD.variance()</code>	0.7223
<code>max()</code>	Return the maximum value.	<code>inputRDD.max()</code>	3.5
<code>min()</code>	Return the minimum value.	<code>inputRDD.min()</code>	1.5

i Example

1. Create an RDD containing the following float values: [1.5,3.5,2.0]
2. Print on the standard output the following statistics

- sum
- mean
- standard deviation
- variance
- maximum value
- minimum value

```
1 ## Create an RDD containing a list of float values
2 inputRDD = sc.parallelize([1.5,3.5,2.0])
3
4 ## Compute the statistics of interest and print them on
5 ## the standard output
6 print("sum:", inputRDD.sum())
7 print("mean:", inputRDD.mean())
8 print("stdev:", inputRDD.stdev())
9 print("variance:", inputRDD.variance())
10 print("max:", inputRDD.max())
11 print("min:", inputRDD.min())
```

16 Cache, Accumulators, Broadcast Variables

16.1 Persistence and Cache

Spark computes the content of an RDD each time an action is invoked on it. If the same RDD is used multiple times in an application, Spark recomputes its content every time an action is invoked on the RDD, or on one of its descendants, but this is expensive, especially for iterative applications.

So, it is possible to ask Spark to persist/cache RDDs: in this way, each node stores the content of its partitions in memory and reuses them in other actions on that RDD/dataset (or RDDs derived from it).

- The first time the content of a persistent/cached RDD is computed in an action, it will be kept in the main memory of the nodes;
- The next actions on the same RDD will read its content from memory (i.e., Spark persists/caches the content of the RDD across operations). This allows future actions to be much faster, often by more than ten times faster.

Spark supports several storage levels, which are used to specify if the content of the RDD is stored

- In the main memory of the nodes
- On the local disks of the nodes
- Partially in the main memory and partially on disk

Table 16.1: Storage levels

Storage Level	Meaning
MEMORY_ONLY	Store RDD as serialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as serialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on (local) disk, and read them from there when they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY, but store the data in off-heap memory. This requires off-heap memory to be enabled.

See [here](#) for more details.

It is possible to mark an RDD to be persisted by using the `persist(storageLevel)` method of the RDD class. The parameter of persist can assume the following values

- `pyspark.StorageLevel.MEMORY_ONLY`
- `pyspark.StorageLevel.MEMORY_AND_DISK`
- `pyspark.StorageLevel.DISK_ONLY`
- `pyspark.StorageLevel.NONE`
- `pyspark.StorageLevel.OFF_HEAP`
- `pyspark.StorageLevel.MEMORY_ONLY_2`
- `pyspark.StorageLevel.MEMORY_AND_DISK_2`

The storage level *_2 replicate each partition on two cluster nodes, so that, if one node fails, the other one can be used to perform the actions on the RDD without recomputing the content of the RDD.

It is possible to cache an RDD by using the `cache()` method of the RDD class: it corresponds to persist the RDD with the storage level 'MEMORY_ONLY' (i.e., it is equivalent to `inRDD.persist(pyspark.StorageLevel.MEMORY_ONLY)`).

! Important

Notice that both persist and cache return a new RDD, since RDDs are immutable.

The use of the persist/cache mechanism on an RDD provides an advantage if the same RDD is used multiple times (i.e., multiple actions are applied on it or on its descendants).

The storage levels that store RDDs on disk are useful if and only if

- the size of the RDD is significantly smaller than the size of the input dataset;
- the functions that are used to compute the content of the RDD are expensive.

Otherwise, recomputing a partition may be as fast as reading it from disk.

Remove data from cache

Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion. It is also possible to manually remove an RDD from the cache by using the `unpersist()` method of the RDD class.

i Example

1. Create an RDD from a textual file containing a list of words (one word for each line);
2. Print on the standard output
 - The number of lines of the input file
 - The number of distinct words

```

1 ## Read the content of a textual file
2 ## and cache the associated RDD
3 inputRDD = sc.textFile("words.txt").cache()
4
5 print("Number of words: ",inputRDD.count())
6 print("Number of distinct words: ", inputRDD.distinct().count())

```

.cache()	The cache method is invoked, hence <code>inputRDD</code> is a cached RDD
inputRDD.count()	This is the first time an action is invoked on the <code>inputRDD</code> RDD. The content of the RDD is computed by reading the lines of the <code>words.txt</code> file and the result of the <code>count</code> action is returned. The content of <code>inputRDD</code> is also stored in the main memory of the nodes of the cluster.
inputRDD.distinct().count()	The content of <code>inputRDD</code> is in the main memory if the nodes of the cluster. Hence the computation of <code>distinct()</code> and <code>count()</code> is performed by reading the data from the main memory and not from the input (HDFS) file <code>words.txt</code> .

16.2 Accumulators

When a function passed to a Spark operation is executed on a remote cluster node, it works on separate copies of all the variables used in the function. These variables are copied to each node of the cluster, and no updates to the variables on the nodes are propagated back to the driver program.

Spark provides a type of shared variables called **accumulators**: accumulators are shared variables that are only “added” to through an associative operation and can therefore be efficiently supported in parallel, and they can be used to implement counters or sums.

Accumulators are usually used to compute simple statistics while performing some other actions on the input RDD, avoiding to use actions like `reduce()` to compute simple statistics (e.g., count the number of lines with some characteristics).

How to use accumulators

1. The driver defines and initializes the accumulator
2. The code executed in the worker nodes increases the value of the accumulator (i.e., the code in the functions associated with the transformations)
3. The final value of the accumulator is returned to the driver node
 - Only the driver node can access the final value of the accumulator
 - The worker nodes cannot access the value of the accumulator: they can only add values to it

! Important

Pay attention that the value of the accumulator is increased in the functions associated with transformations, and, since transformations are lazily evaluated, the value of the accumulator is computed only when an action is executed on the RDD on which the transformations increasing the accumulator are applied.

Spark natively supports numerical accumulators (integers and floats), but programmers can add support for new data types: accumulators are `pyspark.accumulators.Accumulator` objects.

Accumulators are defined and initialized by using the `accumulator(value)` method of the `SparkContext` class: the value of an accumulator can be increased by using the `add(value)` method of the `Accumulator` class, that adds `value` to the current value of the accumulator. The final value of an accumulator can be retrieved in the driver program by using `value` of the `Accumulator` class.

i Example

1. Create an RDD from a textual file containing a list of email addresses (one email for each line);
2. Select the lines containing a valid email and store them in an HDFS file (in this example, an email is considered a valid email if it contains the @ symbol);
3. Print also, on the standard output, the number of invalid emails.

```

1 ## Define an accumulator. Initialize it to 0
2 invalidEmails = sc.accumulator(0)

3

4 ## Read the content of the input textual file
5 emailsRDD = sc.textFile("emails.txt")

6

7 #Define the filtering function
8 def validEmailFunc(line):
9     if (line.find('@')<0):
10         invalidEmails.add(1)
11         return False
12     else:
13         return True
14
15 ## Select only valid emails
16 ## Count also the number of invalid emails
17 validEmailsRDD = emailsRDD.filter(validEmailFunc)

18

19 ## Store valid emails in the output file
20 validEmailsRDD.saveAsTextFile(outputPath)

21

22 ## Print the number of invalid emails
23 print("Invalid email addresses: ", invalidEmails.value)

```

<code>invalidEmails = sc.accumulator(0)</code>	Definition of an accumulator of type integer.
<code>invalidEmails.add(1)</code>	This function increments the value of the <code>invalidEmails</code> accumulator if the email is invalid.
<code>invalidEmails.value</code>	Read the final value of the accumulator. Pay attention that the value of the accumulator is correct only because an action (<code>saveAsTextFile</code>) has been executed on the <code>validEmailsRDD</code> and its content has been computed (the function <code>validEmailFunc</code> has been executed on each element of <code>emailsRDD</code>)

Personalized accumulators

Programmers can define accumulators based on new data types (different from integers and floats): to define a new accumulator data type of type T , the programmer must define a class subclassing the `AccumulatorParam` interface. The `AccumulatorParam` interface has two methods

- `zero` for providing a zero value for your data type
- `addInPlace` for adding two values together

16.3 Broadcast variables

Spark supports broadcast variables. A broadcast variable is a read-only (small/medium) shared variable

- It is instantiated in the driver: the broadcast variable is stored in the main memory of the driver in a local variable;
- It is sent to all worker nodes that use it in one or more Spark operations: the broadcast variable is also stored in the main memory of the executors (which are instantiated in the used worker nodes).

A copy each broadcast variable is sent to all executors that are used to run a task executing a Spark operation based on that variable (i.e., the variable is sent `num.executors` times). A broadcast variable is sent only one time to each executor that uses that variable in at least one Spark operation (i.e., in at least one of its tasks). Each executor can run multiples tasks associated with the same broadcast variable, but the broadcast variable is sent only one time for each executor, hence the amount of data sent on the network is limited by using broadcast variables instead of standard variables.

Broadcast variables are usually used to share (small/medium) lookup-tables, and, since they are stored in local variables, they must be small enough to be stored in the main memory of the driver and also in the main memory of the executors.

Broadcast variables are objects of type `Broadcast`: a broadcast variable (of type T) is defined in the driver by using the `broadcast(value)` method of the `SparkContext` class. The value of a broadcast variable (of type T) is retrieved (usually in transformations) by using `value` of the `Broadcast` class.

i Example

1. Create an RDD from a textual file containing a dictionary of pairs (word, integer value), one pair for each line. Suppose the content of this first file is large but can be stored in main-memory;
2. Create an RDD from a textual file containing a set of words, in particular a sentence (set of words) for each line; Transform the content of the second file mapping each word to an integer based on the dictionary contained in the first file; then, store the result in an HDFS file.
 - First file (dictionary)

```
java 1  
spark 2  
test 3
```

- Second file (the text to transform)

```
java spark  
spark test java
```

- Output file

```
12  
231
```

```

1  ## Read the content of the dictionary from the first file and
2  ## map each line to a pair (word, integer value)
3  dictionaryRDD = sc \
4      .textFile("dictionary.txt") \
5      .map(lambda line: (
6          line.split(" ")[0],
7          line.split(" ")[1]
8      ))
9
10 ## Create a broadcast variable based on the content of dictionaryRDD.
11 ## Pay attention that a broadcast variable can be instantiated only
12 ## by passing as parameter a local variable and not an RDD.
13 ## Hence, the collectAsMap method is used to retrieve the content of the
14 ## RDD and store it in the dictionary variable
15 dictionary = dictionaryRDD.collectAsMap()
16
17 ## Broadcast dictionary
18 dictionaryBroadcast = sc.broadcast(dictionary)
19
20 ## Read the content of the second file
21 textRDD = sc.textFile("document.txt")
22
23 ## Define the function that is used to map strings to integers
24 def myMapFunc(line):
25     transformedLine=''
26     for word in line.split(' '):
27         intValue = dictionaryBroadcast.value[word]
28         transformedLine = transformedLine+intValue+' '
29     return transformedLine.strip()
30
31 ## Map words in textRDD to the corresponding integers and concatenate
32 ## them
33 mappedTextRDD= textRDD.map(myMapFunc)
34
35 ## Store the result in an HDFS file
36 mappedTextRDD.saveAsTextFile(outputPath)

```

`sc.broadcast(dictionary)` Define a broadcast variable.
`dictionaryBroadcast.value[word]` Retrieve the content of the broadcast variable and use it.

16.4 RDDs and Partitions

The content of each RDD is split in partitions: the number of partitions and the content of each partition depend on how RDDs are defined/created. The number of partitions impacts on the maximum parallelization degree of the Spark application, but pay attention that the amount of resources is limited (there is a maximum number of executors and parallel tasks).

💡 How many partitions are good?

Disadvantages of too few partitions

- Less concurrency/parallelism: there could be worker nodes that are idle and could be used to speed up the execution of your application;
- Data skewing and improper resource utilization: data might be skewed on one partition (i.e., one partition with many data, many partitions with few data). The worker node that processes the largest partition needs more time than the other workers, becoming the bottleneck of your application.

Disadvantages of too many partitions

- Task scheduling may take more time than actual execution time if the amount of data in some partitions is too small

Only some specific transformations set the number of partitions of the returned RDD: `parallelize()`, `textFile()`, `repartition()`, `coalesce()`. The majority of the Spark transformations do not change the number of partitions, preserving the number of partitions of the input RDD (i.e., the returned RDD has the same number of partitions of the input RDD).

Transformation	Effect
<code>parallelize(collection)</code>	The number of partitions of the returned RDD is equal to <code>sc.defaultParallelism</code> . Sparks tries to balance the number of elements per partition in the returned RDD; notice that elements are not assigned to partitions based on their value.
<code>parallelize(collection, numSlices)</code>	The number of partitions of the returned RDD is equal to <code>numSlices</code> . Sparks tries to balance the number of elements per partition in the returned RDD; notice that elements are not assigned to partitions based on their value.
<code>textFile(pathInputData)</code>	The number of partitions of the returned RDD is equal to the number of input chunks/blocks of the input HDFS data. Each partition contains the content of one of the input blocks.
<code>textFile(pathInputData, minPartitions)</code>	The user specified number of partitions must be greater than the number of input blocks. The number of partitions of the returned RDD is greater than or equal to the specified value <code>minPartitions</code> , and each partition contains a part of one input blocks.

Transformation	Effect
<code>repartition(numPartitions)</code>	<p><code>numPartitions</code> can be greater or smaller than the number of partitions of the input RDD, and the number of partitions of the returned RDD is equal to <code>numPartitions</code>. Sparks tries to balance the number of elements per partition in the returned RDD; notice that elements are not assigned to partitions based on their value. A shuffle operation is executed to assign input elements to the partitions of the returned RDD.</p>
<code>coalesce(numPartitions)</code>	<p><code>numPartitions</code> is smaller than the number of partitions of the input RDD, and the number of partitions of the returned RDD is equal to <code>numPartitions</code>. Sparks tries to balance the number of elements per partition in the returned RDD; notice that elements are not assigned to partitions based on their value. Usually no shuffle operation is executed to assign input elements to the partitions of the returned RDD, so coalesce is more efficient than repartition to reduce the number of partitions.</p>

Spark allows specifying how to partition the content of RDDs of key-value pairs: the input tpairs are grouped in partitions based on the integer value returned by a function applied on the key of each input pair. This operation can be useful to improve the efficiency of the next transformations by reducing the amount of shuffle operations and the amount of data sent on the network in the next steps of the application (Spark can optimize the execution of the transformations if the input RDDs of pairs are properly partitioned).

`partitionBy(numPartitions)`

Partitioning is based on the `partitionBy()` transformation. The input pairs are grouped in partitions based on the integer value returned by a default hash function applied on the key of each input pair. A shuffle operation is executed to assign input elements to the partitions of the returned RDD.

Suppose that the number of partition of the returned Pair RDD is `numPart`. The default partition function is `portable_hash`: given an input pair $(key, value)$ a copy of that pair will be stored in the partition number n of the returned RDD, where

$$n = \text{portable_hash}(key) \% \text{numPart}$$

Notice that `portable_hash(key)` returns an integer.

`partitionBy(numPartitions, partitionFunc)`

The input pairs are grouped in partitions based on the integer value returned by the user provided `partitionFunc` function. A shuffle operation is executed to assign input elements to the partitions of the returned RDD.

Suppose that the number of partition of the returned Pair RDD is **numPart**. The custom partition function is **partitionFunc**: given an input pair $(key, value)$ a copy of that pair will be stored in the partition number n of the returned RDD, where

$$n = \text{partitionFunc}(key) \% \text{numPart}$$

💡 Use case scenario

Partitioning Pair RDDs by using **partitionBy()** is useful only when the same partitioned RDD is cached and reused multiple times in the application in time and network consuming key-oriented transformations.

For example, the same partitioned RDD is used in many **join()**, **cogroup()**, **groupByKey()**, ... transformations in different paths/branches of the application (different paths/branches of the DAG).

Pay attention to the amount of data that is actually sent on the network: **partitionBy()** can slow down the application instead of speeding it up.

ℹ️ Example

1. Create an RDD from a textual file containing a list of pairs (pageID, list of linked pages);
2. Implement the (simplified) PageRank algorithm and compute the pageRank of each input page;
3. Print the result on the standard output.

```

1 ## Read the input file with the structure of the web graph
2 inputData = sc.textFile("links.txt")
3
4 ## Format of each input line
5 ## PageId,LinksToOtherPages - e.g., P3 [P1,P2,P4,P5]
6 def mapToPairPageIDLinks(line):
7     fields = line.split(' ')
8     pageID = fields[0]
9     links = fields[1].split(',')
10    return (pageID, links)
11
12 links = inputData.map(mapToPairPageIDLinks) \
13     .partitionBy(inputData.getNumPartitions()) \
14     .cache()
15
16 ## Initialize each page's rank to 1.0; since we use mapValues,
17 ## the resulting RDD will have the same partitioner as links
18 ranks = links.mapValues(lambda v: 1.0)
19
20 ## Function that returns a set of pairs from each input pair
21 ## input pair: (pageid, (linked pages, current page rank of pageid) )
22 ## one output pair for each linked page. Output pairs:
23 ## (pageid linked page,
24 ## current page rank of the linking page pageid / number of linked pages)
25 def computeContributions(pageIDLinksPageRank):
26     pagesContributions = []
27     currentPageRank = pageIDLinksPageRank[1][1]
28     linkedPages = pageIDLinksPageRank[1][0]
29     numLinkedPages = len(linkedPages)
30     contribution = currentPageRank/numLinkedPages
31
32     for pageidLinkedPage in linkedPages:
33         pagesContributions.append((pageidLinkedPage, contribution))
34
35     return pagesContributions
36
37 ## Run 30 iterations of PageRank
38 for x in range(30):
39     ## Retrieve for each page its current pagerank and
40     ## the list of linked pages by using the join transformation
41
42     pageRankLinks = links.join(ranks)
43     ## Compute contributions from linking pages to linked pages
44     ## for this iteration
45
46     contributions = pageRankLinks.flatMap(computeContributions)
47     ## Update current pagerank of all pages for this iteration
48     ranks = contributions\
49         .reduceByKey(lambda contrib1, contrib2: contrib1+contrib2)
50
51 ## Print the result
52 ranks.collect()

```

```
.partitionBy()...cache()  
links
```

Notice that the returned Pair RDD is partitioned and cached. The join transformation is invoked many times on the links Pair RDD. The content of links is constant (it does not change during the loop iterations). Hence, caching it and also partitioning its content by key is useful: its content is computed one time and cached in the main memory of the executors, and it is shuffled and sent on the network only one time because `partitionBy` was applied on it

Default partitioning behavior of the main transformations

Transformation	Number of partitions	Partitioner
<code>sc.parallelize()</code>	<code>sc.defaultParallelism</code> the maximum between <code>sc.defaultParallelism</code> and the number of file blocks same as parent RDD	NONE NONE
<code>filter()</code> , <code>map()</code> , <code>flatMap()</code> , <code>distinct()</code>	NONE except filter preserve parent	
<code>rdd.union(otherRDD)</code>	<code>rdd.partitions.size +</code> <code>otherRDD.partitions.size</code>	RDD's partitioner
<code>rdd.intersection(otherRDD)</code>	<code>max(rdd.partitions.size,</code> <code>otherRDD.partitions.size)</code>	RDD's partitioner
<code>rdd.subtract(otherRDD)</code>	<code>rdd.partitions.size</code>	RDD's partitioner
<code>rdd.cartesian(otherRDD)</code>	<code>rdd.partitions.size * otherRDD.</code> <code>partitions.size</code>	NONE except filter preserve parent RDD's partitioner
<code>reduceByKey()</code> , <code>foldByKey()</code> , <code>combineByKey()</code> ,	same as parent RDD	HashPartitioner
<code>groupByKey()</code>		
<code>sortByKey()</code>	same as parent RDD	RangePartitioner
<code>mapValues()</code> , <code>flatMapValues()</code>	same as parent RDD	parent RDD's partitioner
<code>cogroup()</code> , <code>join()</code> , <code>leftOuterJoin()</code> ,	depends upon input properties of two involved RDDs	HashPartitioner
<code>rightOuterJoin()</code>		

16.5 Broadcast join

The join transformation is expensive in terms of execution time and amount of data sent on the network. If one of the two input RDDs of key-value pairs is small enough to be stored in the main memory, then it is possible to use a more efficient solution based on a broadcast variable.

- Broadcast hash join (or map-side join)
- The smaller the small RDD, the higher the speed up

Example

1. Create a large RDD from a textual file containing a list of pairs (`userID, post`); notice that each user can be associated to several posts;
2. Create a small RDD from a textual file containing a list of pairs (`userID, (name, surname, age)`); notice that each user can be associated to one single line in this second file;
3. Compute the join between these two files.

```

1 ## Read the first input file
2 largeRDD = sc \
3     .textFile("post.txt") \
4     .map(lambda line: (
5         int(line.split(',') [0]),
6         line.split(',') [1]
7     ))
8
9 ## Read the second input file
10 smallRDD = sc \
11     .textFile("profiles.txt") \
12     .map(lambda line: (
13         int(line.split(',') [0]),
14         line.split(',') [1]
15     ))
16
17 ## Broadcast join version
18 ## Store the "small" RDD in a local python variable in the driver
19 ## and broadcast it
20 localSmallTable = smallRDD.collectAsMap()
21 localSmallTableBroadcast = sc.broadcast(localSmallTable)
22
23 ## Function for joining a record of the large RDD with the matching
24 ## record of the small one
25 def joinRecords(largeTableRecord):
26     returnedRecords = []
27     key = largeTableRecord[0]
28     valueLargeRecord = largeTableRecord[1]
29
30     if key in localSmallTableBroadcast.value:
31         returnedRecords.append((
32             key,
33             (valueLargeRecord,localSmallTableBroadcast.value[key])
34         ))
35
36     return returnedRecords
37
38 ## Execute the broadcast join operation by using a flatMap
39 ## transformation on the "large" RDD
40 userPostProfileRDBBroadcatJoin = largeRDD.flatMap(joinRecords)

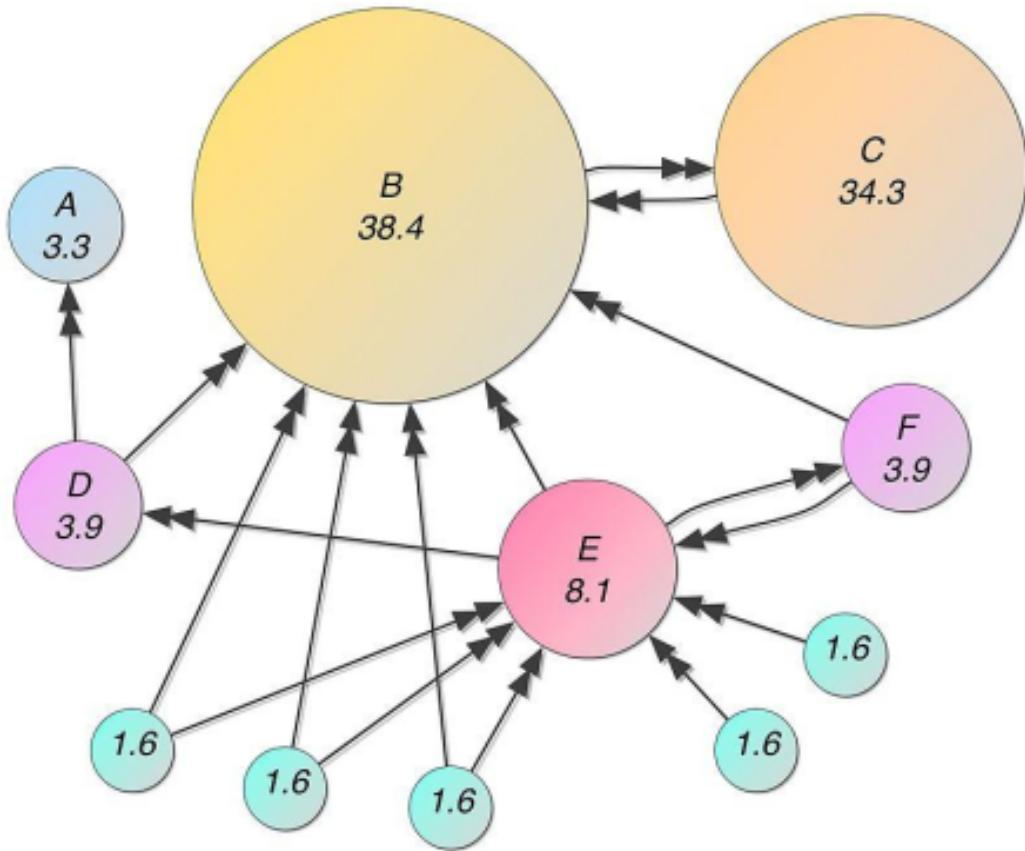
```

17 Introduction to PageRank

PageRank is the original famous algorithm used by the Google Search engine to rank vertexes (web pages) in a graph by order of importance. For the Google search engine, vertexes are web pages in the World Wide Web, and edges are hyperlinks among web pages: PageRank works by assigning a numerical weighting (importance) to each node.

In other words, it computes a likelihood that a person randomly clicking on links will arrive at any particular web page. So, to have a high PageRank, it is important to have many in-links, and be liked by relevant pages (pages characterized by a high PageRank).

Figure 17.1: PageRank basic idea



💡 Basic idea

- The vote of each link is proportional to the importance of its source page p ;
- If page p with importance $\text{PageRank}(p)$ has n out-links, each out-link gets $\frac{\text{PageRank}(p)}{n}$ votes;

- Page p importance is the sum of the votes on its in-links.

17.1 PageRank formulations

Simple recursive formulation

- Initialize each page rank to 1.0: for each p in pages set $\text{PageRank}(p)$ to 1.0
- Iterate for max iterations
 1. Page p sends a contribution $\frac{\text{PageRank}(p)}{\text{numOutLinks}(p)}$ to its neighbors (the pages it links);
 2. Update each page rank $\text{PageRank}(p)$ with the sum of the received contributions.

Random jumps formulation

The PageRank algorithm simulates the “random walk” of a user on the web. Indeed, at each step of the random walk, the random surfer has two options:

- with probability $1 - \alpha$, follow a link at random among the ones in the current page;
- with probability α , jump to a random page.
- Initialize each page rank to 1.0: for each p in pages set $\text{PageRank}(p)$ to 1.0
- Iterate for max iterations
 1. Page p sends a contribution $\frac{\text{PageRank}(p)}{\text{numOutLinks}(p)}$ to its neighbors (the pages it links);
 2. Update each page rank $\text{PageRank}(p)$ to $\alpha + (1 - \alpha)$ times the sum of the received contributions.

Example

- $\alpha = 0.15$
- Initialization: $\forall p, \text{PageRank}(p) = 1.0$

Figure 17.2: Initialization

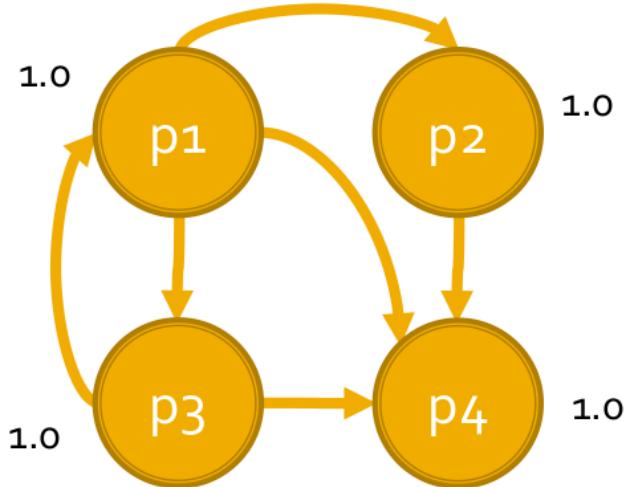
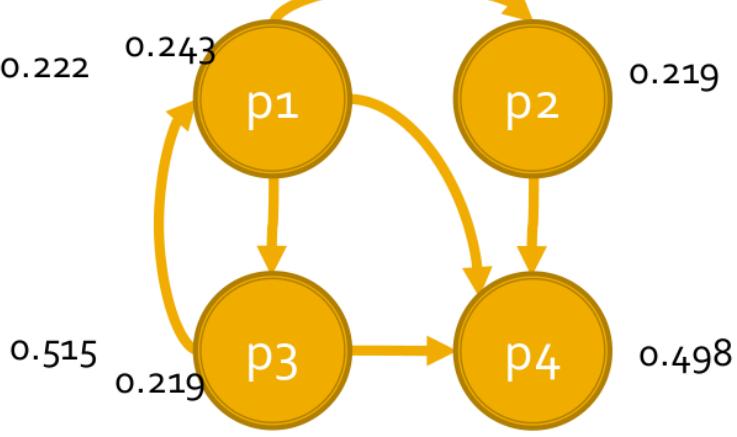
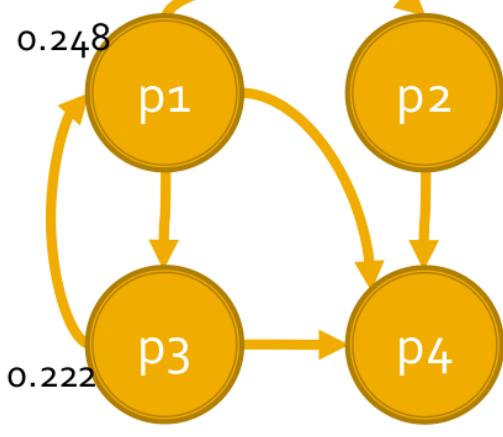
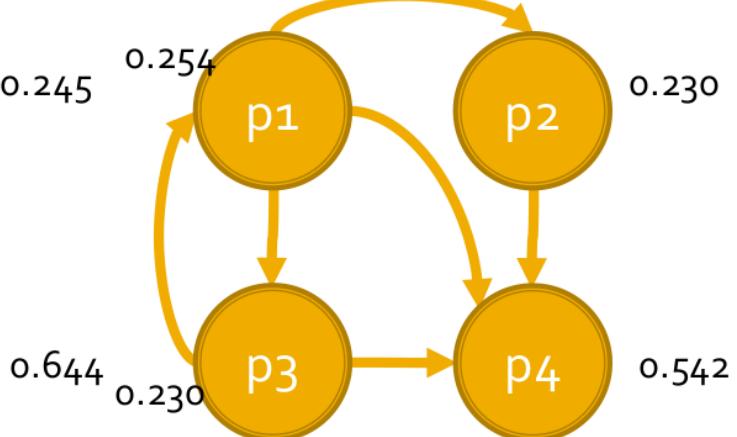
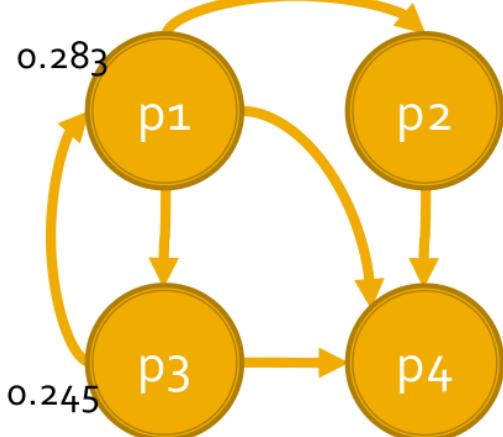
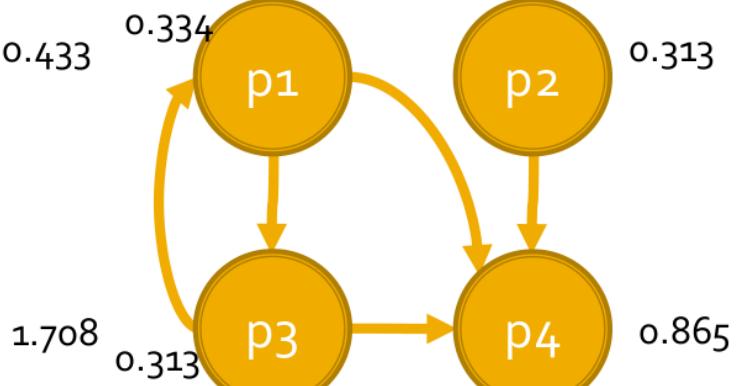
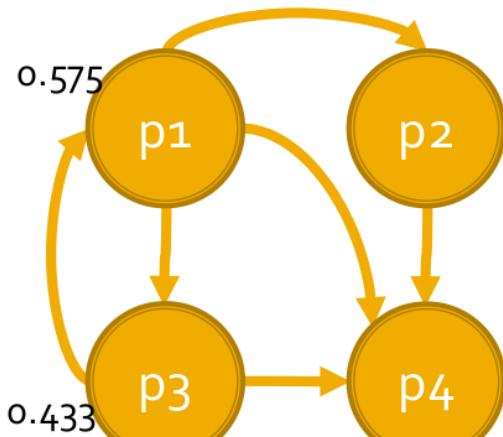


Figure 17.3: Iterations



18 Spark SQL and DataFrames

18.1 Spark SQL

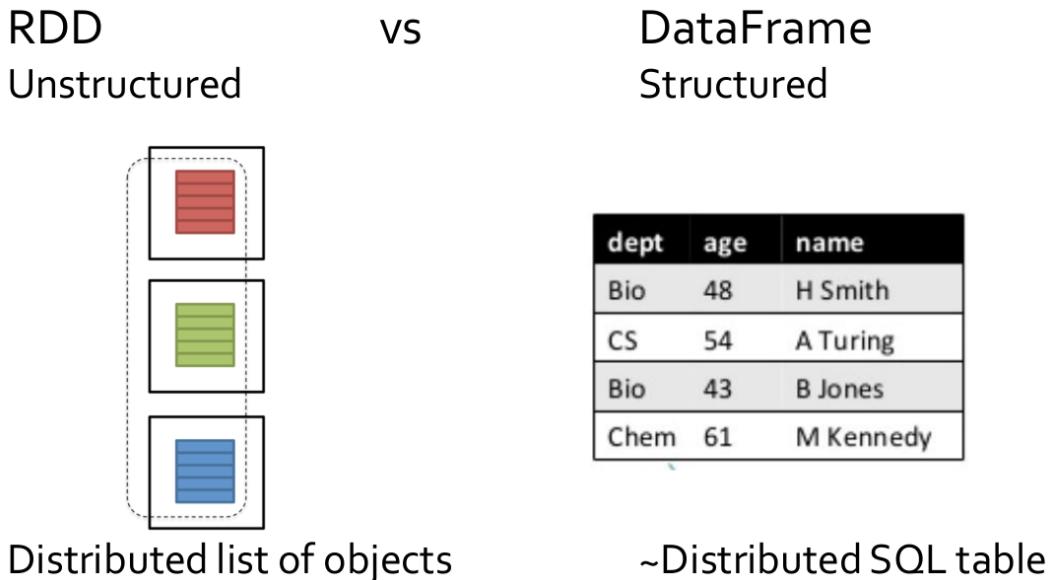
Spark SQL is the Spark component for structured data processing. It provides a programming abstraction called *Dataframe* and can act as a distributed SQL query engine: the input data can be queried by using

- Ad-hoc methods
- Or an SQL-like language

Spark SQL vs Spark RDD APIs

The interfaces provided by Spark SQL provide more information about the structure of both the data and the computation being performed. Spark SQL uses this extra information to perform extra optimizations based on a “SQL-like” optimizer called **Catalyst**, and so programs based on Dataframe are usually faster than standard RDD-based programs.

Figure 18.1: Spark SQL vs Spark RDD APIs



DataFrames

A DataFrame is a distributed collection of structured data. It is conceptually equivalent to a table in a relational database, and it can be created reading data from different types of external sources (CSV files,

JSON files, RDBMs,...). A DataFrame benefits from Spark SQL optimized execution engine exploiting the information about the data structure.

All the Spark SQL functionalities are based on an instance of the `pyspark.sql.SparkSession` class

To import it in a standalone application use

```
1 from pyspark.sql import SparkSession
```

To instance a `SparkSession` object use

```
1 spark = SparkSession.builder.getOrCreate()
```

To close a `SparkSession` use the `SparkSession.stop()` method

```
1 spark.stop()
```

18.2 DataFrames

A DataFrame is a distributed collection of data organized into named columns, equivalent to a relational table: DataFrames are lists of **Row objects**.

The classes used to define DataFrames are

- `pyspark.sql.DataFrame`
- `pyspark.sql.Row`

DataFrames can be created from different sources

- Structured (textual) data files (e.g., csv files, json files);
- Existing RDDs;
- Hive tables;
- External relational databases.

Creating DataFrames from csv files

Spark SQL provides an API that allows creating DataFrames directly from CSV files. The creation of a DataFrame from a csv file is based the `load(path)` method of the `pyspark.sql.DataFrameReader` class, where `path` is the path of the input file. To get a `DataFrameReader` using the the `read()` method of the `SparkSession` class.

```
1 df = spark.read.load(path, options)
```

Example

Create a DataFrame from a csv file (“people.csv”) containing the profiles of a set of people. Each line of the file contains name and age of a person, and age can assume the null value (i.e., it can be missing). The first line contains the header (i.e., the names of the attributes/columns).

Example of csv file

```
name,age
Andy,30
Michael,
Justin,19
```

Notice that the age of the second person is unknown.

```
1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
```

<code>format="csv"</code>	This is used to specify the format of the input file
<code>header=True</code>	This is used to specify that the first line of the file contains the name of the attributes/columns
<code>inferSchema=True</code>	This method is used to specify that the system must infer the data types of each column. Without this option all columns are considered strings

Creating DataFrames from JSON files

Spark SQL provides an API that allows creating a DataFrame directly from a textual file where each line contains a JSON object. Hence, the input file is not a standard JSON file: it must be properly formatted in order to have one JSON object (tuple) for each line. So, the format of the input file must be compliant with the **JSON Lines text format**, also called newline-delimited JSON.

The creation of a DataFrame from JSON files is based on the same method used for reading csv files, that is the `load(path)` method of the `pyspark.sql.DataFrameReader` class, where `path` is the path of the input file. To get a `DataFrameReader` use the `read()` method of the `SparkSession` class.

```
1 df = spark.read.load(path, format="json", options)
```

or

```
1 df = spark.read.json(path, options)
```

The same API allows also reading standard multiline JSON files by setting the multiline option to true by setting the argument `multiLine=True` on the defined `DataFrameReader` for reading standard JSON files (this feature is available since Spark 2.2.0).

Caution

Pay attention that reading a set of small JSON files from HDFS is very slow.

Example 1

Create a DataFrame from a JSON Lines text formatted file (“people.json”) containing the profiles of a set of people: each line of the file contains a JSON object containing name and age of a person. Age can assume the null value.

Example of JSON file

```
{"name": "Michael"}  
{"name": "Andy", "age": 30}  
{"name": "Justin", "age": 19}
```

Notice that the age of the first person is unknown.

```
1 ## Create a Spark Session object  
2 spark = SparkSession.builder.getOrCreate()  
3  
4 ## Create a DataFrame from people.csv  
5 df = spark.read.load(  
6     "people.json",  
7     format="json"  
8 )
```

`format="json"`

This method is used to specify the format of the input file.

Example 2

Create a DataFrame from a folder containing a set of standard multiline JSON files: each input JSON file contains the profile of one person, in particular each file contains name and age of a person. Age can assume the null value.

Example of JSON file

```
{"name": "Andy", "age": 30}
```

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "folder_JSONFiles/",
7     format="json",
8     multiLine=True
9 )

```

`multiLine=True` This multiline option is set to true to specify that the input files are standard multiline JSON files.

Creating DataFrames from other data sources

The `DataFrameReader` class (the same we used for reading a json file and store it in a DataFrame) provides other methods to read many standard (textual) formats and read data from external databases:

- Apache parquet files
- external relational database, through a JDBC connection
- Hive tables
- ...

Creating DataFrames from RDDs or Python lists

The content of an RDD of tuples or the content of a Python list of tuples can be stored in a DataFrame by using the `spark.createDataFrame(data,schema)` method, where `data` is a RDD of tuples or Rows, Python list of tuples or Rows, or pandas DataFrame, and `schema` is a list of string with the names of the columns/attributes. `schema` is optional, and if not specified the column names are set to `_1`, `_2`, ..., `_n` for input RDDs/lists of tuples.

i Example

Create a DataFrame from the following Python list

```

1 [
2     (19,"Justin"),
3     (30,"Andy"),
4     (None,"Michael")
5 ]

```

The column names must be set to “age” and “name”.

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a Python list of tuples
5 profilesList = [
6     (19,"Justin"),
7     (30,"Andy"),
8     (None,"Michael")
9 ]
10
11 ## Create a DataFrame from the profilesList
12 df = spark.createDataFrame(profilesList,["age","name"])

```

From DataFrame to RDD

The `rdd` method of the `DataFrame` class returns an RDD of Row objects containing the content of the DataFrame which it is invoked on. Each Row object is like a dictionary containing the values of a record: it contains column names in the keys and column values in the values.

Usage of the Row class

The fields in it can be accessed:

- like attributes: `row.key`, where `key` is a column name;
- like dictionary values: `row["key"]`;
- using `for key in row` will search through row keys.

Also the `asDict()` method returns the Row content as a Python dictionary.

Example

1. Create a DataFrame from a csv file containing the profiles of a set of people: each line of the file contains name and age of a person, but the first line contains the header (i.e., the name of the attributes/columns);
2. Transform the input DataFrame into an RDD, select only the name field/column and store the result in the output folder.

```

1  ## Create a Spark Session object
2  spark = SparkSession.builder.getOrCreate()
3
4  ## Create a DataFrame from people.csv
5  df = spark.read.load(
6      "people.csv",
7      format="csv",
8      header=True,
9      inferSchema=True
10 )
11
12 ## Define an RDD based on the content of
13 ## the DataFrame
14 rddRows = df.rdd
15
16 ## Use the map transformation to extract
17 ## the name field/column
18 rddNames = rddRows.map(lambda row: row.name)
19
20 ## Store the result
21 rddNames.saveAsTextFile(outputPath)

```

18.3 Operations on DataFrames

A set of specific methods are available for the `DataFrame` class (e.g., `show()`, `printSchema()`, `count()`, `distinct()`, `select()`, `filter()`), also the standard `collect()` and `count()` actions are available.

Show method

The `show(n)` method of the `DataFrame` class prints on the standard output the first `n` of the input DataFrame. Default value of `n` is 20.

Example

1. Create a DataFrame from a csv file containing the profiles of a set of people;
2. Print the content of the first 2 people (i.e., the first 2 rows of the DataFrame).

The content of `people.csv` is

```

name,age
Andy,30
Michael,
Justin,19

```

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 df.show(2)

```

PrintSchema method

The `printSchema()` method of the `DataFrame` class prints on the standard output the schema of the DataFrame (i.e., the name of the attributes of the data stored in the DataFrame).

Count method

The `count()` method of the `DataFrame` class returns the number of rows in the input DataFrame.

Distinct method

The `distinct()` method of the `DataFrame` class returns a new DataFrame that contains only the unique rows of the input DataFrame. A shuffle phase is needed.

Caution

Pay attention that the `distinct` operation is always an heavy operation in terms of data sent on the network.

Example

1. Create a DataFrame from a csv file containing the names of a set of people. The first line is the header.
2. Create a new DataFrame without duplicates.

The content of “names.csv” is

```

name
Andy
Michael
Justin

```

Michael

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from names.csv
5 df = spark.read.load(
6     "names.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 df_distinct = df.distinct()

```

Select method

The `select(col1, ..., coln)` method of the `DataFrame` class returns a new DataFrame that contains only the specified columns of the input DataFrame. Use `*` as special column to select all columns



Caution

Pay attention that the select method can generate errors at runtime if there are mistakes in the names of the columns.



Example

1. Create a DataFrame from the “people2.csv” file that scontains the profiles of a set of people
 - The first line contains the header;
 - The others lines contain the users’ profiles: one line per person, and each line contains name, age, and gender of a person.
2. Create a new DataFrame containing only name and age of the people.

The “people2.csv” has the following structure

```

name,age,gender
Paul,40,male
John,40,male

```

```

1  ## Create a Spark Session object
2  spark = SparkSession.builder.getOrCreate()
3
4  ## Create a DataFrame from people2.csv
5  df = spark.read.load(
6      "people2.csv",
7      format="csv",
8      header=True,
9      inferSchema=True
10 )
11
12 dfNamesAges = df.select("name", "age")

```

SelectExpr method

The `selectExpr(expression1, ..., expressionN)` method of the `DataFrame` class is a variant of the `select` method, where `expr` can be a SQL expression.

Example 1

1. Create a DataFrame from the “people2.csv” file that scontains the profiles of a set of people
 - The first line contains the header;
 - The others lines contain the users’ profiles: one line per person, and each line contains name, age, and gender of a person.
2. Create a new DataFrame containing only name and age of the people.
3. Create a new DataFrame containing only the name of the people and their age plus one. Call the age column as “new_age”.

The “people2.csv” has the following structure

```

name,age,gender
Paul,40,male
John,40,male

```

```
1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people2.csv
5 df = spark.read.load(
6     "people2.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 dfNamesAges = df.selectExpr("name", "age")
13
14 dfNamesAgesMod = df.selectExpr("name", "age + 1 AS new_age")
```

Example 2

1. Create a DataFrame from the “people2.csv” file that contains the profiles of a set of people
 - The first line contains the header;
 - The others lines contain the users’ profiles: each line contains name, age, and gender of a person.
2. Create a new DataFrame containing the same columns of the initial dataset plus an additional column called “newAge” containing the value of age incremented by one.

The “people2.csv” has the following structure

```
name,age,gender
Paul,40,male
John,40,male
```

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people2.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 ## Create a new DataFrame with four columns:
13 ## name, age, gender, newAge = age +1
14 dfNewAge = df.selectExpr(
15     "name",
16     "age",
17     "gender",
18     "age+1 as newAge"
19 )

```

"... as newAge"

This part of the expression is used to specify the name of the column associated with the result of the first part of the expression in the returned DataFrame. Without this part of the expression, the name of the returned column would be "age+1".

Filter method

The `filter(conditionExpr)` method of the `DataFrame` class returns a new DataFrame that contains only the rows satisfying the specified condition. The condition is expressed as a Boolean SQL expression.

Caution

Pay attention that this version of the filter method can generate errors at runtime if there are errors in the filter expression: the parameter is a string and the system cannot check the correctness of the expression at compile time.

Example

1. Create a DataFrame from the "people.csv" file that contains the profiles of a set of people
 - The first line contains the header;
 - The others lines contain the users' profiles: each line contains name and age of a person.
2. Create a new DataFrame containing only the people with age between 20 and 31.

```

1  ## Create a Spark Session object
2  spark = SparkSession.builder.getOrCreate()
3
4  ## Create a DataFrame from people.csv
5  df = spark.read.load(
6      "people.csv",
7      format="csv",
8      header=True,
9      inferSchema=True
10 )
11
12 df_filtered = df.filter("age>=20 and age<=31")

```

Where method

The `where(expression)` method of the `DataFrame` class is an alias of the `filter(conditionExpr)` method.

Join

The `join(right, on, how)` method of the `DataFrame` class is used to join two DataFrames. It returns a DataFrame that contains the join of the tuples of the two input DataFrames based on the `on` join condition.

`on` specifies the join condition. It can be:

- a string: the column to join
- a list of strings: multiple columns to join
- a condition/an expression on the columns (e.g., `joined_df = df.join(df2, df.name == df2.name)`)

`how` specifies the type of join

- `inner` (default type of join)
- `cross`
- `outer`
- `full`
- `full_outer`
- `left`
- `left_outer`
- `right`
- `right_outer`
- `left_semi`
- `left_anti`

 Caution

Pay attention that this method: can generate errors at runtime if there are errors in the join expression.

 Example 1

1. Create two DataFrames

- One based on the “people_id.csv” file that contains the profiles of a set of people, the schema is: uid, name, age;
- One based on the liked_sports.csv file that contains the liked sports for each person, the schema is: uid, sportname. 2.Join the content of the two DataFrames (uid is the join column) and show it on the standard output.

```

1  ## Create a Spark Session object
2  spark = SparkSession.builder.getOrCreate()
3
4  ## Read people_id.csv and store it in a DataFrame
5  dfPeople = spark.read.load(
6      "people_id.csv",
7      format="csv",
8      header=True,
9      inferSchema=True
10 )
11
12 ## Read liked_sports.csv and store it in a DataFrame
13 dfUidSports = spark.read.load(
14     "liked_sports.csv",
15     format="csv",
16     header=True,
17     inferSchema=True
18 )
19
20 ## Join the two input DataFrames
21 dfPersonLikes = dfPeople.join(
22     dfUidSports,
23     dfPeople.uid == dfUidSports.uid
24 )
25
26 ## Print the result on the standard output
27 dfPersonLikes.show()
```

dfPeople.uid == dfUidSports.uid

Specify the join condition on the uid columns.

Example 2

1. Create two DataFrames

- One based on the “people_id.csv” file that contains the profiles of a set of people, the schema is: uid, name, age;
- One based on the banned.csv file that contains the banned users, the schema is: uid, bannedmotivation.

2. Select the profiles of the non-banned users and show them on the standard output.

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Read people_id.csv and store it in a DataFrame
5 dfPeople = spark.read.load(
6     "people_id.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 ## Read banned.csv and store it in a DataFrame
13 dfBannedUsers = spark.read.load(
14     "banned.csv",
15     format="csv",
16     header=True,
17     inferSchema=True
18 )
19
20 ## Apply the Left Anti Join on the two input DataFrames
21 dfSelectedProfiles = dfPeople.join(
22     dfBannedUsers,
23     dfPeople.uid == dfBannedUsers.uid,
24     "left_anti"
25 )
26
27 ## Print the result on the standard output
28 dfSelectedProfiles.show()
```

dfPeople.uid == dfUidSports.uid
 "left_anti"

Specify the (anti) join condition on the uid columns.
 Use Left Anti Join.

Aggregate functions

Aggregate functions are provided to compute aggregates over the set of values of columns. Some of the provided aggregate functions/methods are

- `avg(column)`
- `count(column)`
- `sum(column)`
- `abs(column)`
- ...

Each aggregate function returns one value computed by considering all the values of the input column.

The `agg(expr)` method of the `DataFrame` class is used to specify which aggregate function we want to apply on one input column. The result is a `DataFrame` containing one single row and one single column, and the name of the return column is “`function_name(column)`”.



Caution

Pay attention that this methods can generate errors at runtime (e.g., wrong attribute name, wrong data type).



1. Create a `DataFrame` from the “`people.csv`” file that contains the profiles of a set of people (each line contains name and age of a person)
 - The first line contains the header;
 - The others lines contain the users’ profiles.
2. Create a `Dataset` containing the average value of age.

Input file example

```
name,age
Andy,30
Michael,15
Justin,19
Andy,40
```

Expected output example

```
avg(age)
26.0
```

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 ## Compute the average of age
13 averageAge = df.agg({"age": "avg"})

```

groupBy and aggregate functions

The method `groupBy(col1, ..., coln)` method of the `DataFrame` class combined with a set of aggregate methods can be used to split the input data in groups and compute aggregate function over each group.



Caution

Pay attention that this methods can generate errors at runtime if there are semantic errors (e.g., wrong attribute names, wrong data types).

It is possible to specify which attributes are used to split the input data in groups by using the `groupBy(col1, ..., coln)` method, and then, apply the aggregate functions to compute by final result (the result is a `DataFrame`).

Some of the provided aggregate functions/methods are

- `avg(column)`
- `count(column)`
- `sum(column)`
- `abs(column)`
- `...`

Otherwise, the `agg()` method can be used to apply multiple aggregate functions at the same time over each group.

See the static methods of the `pyspark.sql.GroupedData` class for a complete list.



Example 1

1. Create a DataFrame from the “people.csv” file that contains the profiles of a set of people
 - The first line contains the header;
 - The others lines contain the users’ profiles: each line contains name and age of a person.

2. Create a DataFrame containing the for each name the average value of age.

Input file example

```
name,age
Andy,30
Michael,15
Justin,19
Andy,40
```

Expected output example

```
name,avg(age)
Andy,35
Michael,15
Justin,19
```

```
1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 grouped = df.groupBy("name").avg("age")
```

Example 2

1. Create a DataFrame from the “people.csv” file that contains the profiles of a set of people
 - The first line contains the header
 - The others lines contain the users’ profiles: each line contains name and age of a person
2. Create a DataFrame containing the for each name the average value of age and the number of person with that name

Input file example

```
name,age
Andy,30
Michael,15
Justin,19
Andy,40
```

Expected output example

```

name,avg(age),count(name)
Andy,35,2
Michael,15,1
Justin,19,1

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 grouped = df.groupBy("name") \
13     .agg({"age": "avg", "name": "count"})

```

Sort method

The `sort(col1, ..., coln, ascending=True)` method of the `DataFrame` class returns a new `DataFrame` that contains the same data of the input one, but whose content is sorted by `col1, ..., coln`. `ascending` determines if the sort should be ascending (`True`) or descending (`False`).

18.4 DataFrames and the SQL language

Sparks allows querying the content of a `DataFrame` also by using the SQL language, but in order to do this a table name must be assigned to a `DataFrame`. The `createOrReplaceTempView(tableName)` method of the `DataFrame` class can be used to assign a `tableName` as table name to the `DataFrame` which it is invoked on.

Once the `DataFrame` has been mapped to table names, SQL-like queries can be executed (the executed queries return `DataFrame` objects). The `sql(query)` method of the `SparkSession` class can be used to execute a SQL-like query, where `query` is a SQL-like query. Currently some SQL features are not supported (e.g., nested subqueries in the “WHERE” clause are not allowed).

Example 1

1. Create a `DataFrame` from a JSON file containing the profiles of a set of people: each line of the file contains a JSON object containing name, age, and gender of a person;
2. Create a new `DataFrame` containing only the people with age between 20 and 31 and print

them on the standard output (use the SQL language to perform this operation).

```
1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.json",
7     format="json"
8 )
9
10 ## Assign the "table name" people to the df DataFrame
11 df.createOrReplaceTempView("people")
12
13 ## Select the people with age between 20 and 31
14 ## by querying the people table
15 selectedPeople = spark.sql(
16     "SELECT * FROM people WHERE age>=20 and age<=31"
17 )
18
19 ## Print the result on the standard output
20 selectedPeople.show()
```

i Example 2

1. Create two DataFrames

- One based on the “people_id.csv” file that contains the profiles of a set of people, the schema is: uid, name, age;
- One based on the “liked_sports.csv” file that contains the liked sports for each person, the schema is: uid, sportname.

2. Join the content of the two DataFrames and show it on the standard output.

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Read people_id.csv and store it in a DataFrame
5 dfPeople = spark.read.load(
6     "people_id.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 ## Assign the "table name" people to the dfPerson
13 dfPeople.createOrReplaceTempView("people")
14
15 ## Read liked_sports.csv and store it in a DataFrame
16 dfUidSports = spark.read.load(
17     "liked_sports.csv",
18     format="csv",
19     header=True,
20     inferSchema=True
21 )
22
23 ## Assign the "table name" liked to dfUidSports
24 dfUidSports.createOrReplaceTempView("liked")
25
26 ## Join the two input tables by using the
27 #SQL-like syntax
28 dfPersonLikes = spark.sql(
29     "SELECT * from people, liked where people.uid=liked.uid"
30 )
31
32 ## Print the result on the standard output
33 dfPersonLikes.show()

```

i Example 3

1. Create a DataFrame from the “people.csv” file that contains the profiles of a set of people
 - The first line contains the header;
 - The others lines contain the users’ profiles: each line contains name and age of a person.
2. Create a DataFrame containing for each name the average value of age and the number of person with that name. Print its content on the standard output.

Input file example

name,age

```
Andy,30
Michael,15
Justin,19
Andy,40
```

Expected output example

```
name,avg(age),count(name)
Andy,35,2
Michael,15,1
Justin,19,1
```

```
1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.json",
7     format="json"
8 )
9
10 ## Assign the "table name" people to the df DataFrame
11 df.createOrReplaceTempView("people")
12
13 ## Define groups based on the value of name and
14 ## compute average and number of records for each group
15 nameAvgAgeCount = spark.sql(
16     "SELECT name, avg(age), count(name) FROM people GROUP BY name"
17 )
18
19 ## Print the result on the standard output
20 nameAvgAgeCount.show()
```

18.5 Save DataFrames

The content of DataFrames can be stored on disk by using two approaches

- Convert DataFrames to traditional RDDs by using the `rdd` method of the DataFrame, and then use `saveAsTextFile(outputFolder)`;
- Use the `write()` method of DataFrames, that returns a `DataFrameWriter` class instance.

i Example 1

1. Create a DataFrame from the “people.csv” file that contains the profiles of a set of people
 - The first line contains the header;
 - The others lines contain the users’ profiles: each line contains name, age, and gender of a person.
2. Store the DataFrame in the output folder by using the `saveAsTextFile()` method.

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 ## Save it
13 df.rdd.saveAsTextFile(outputPath)

```

i Example 2

1. Create a DataFrame from the “people.csv” file that contains the profiles of a set of people
 - The first line contains the header;
 - The others lines contain the users’ profiles: each line contains name, age, and gender of a person.
2. Store the DataFrame in the output folder by using the `write()` method, with the CSV format.

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Create a DataFrame from people.csv
5 df = spark.read.load(
6     "people.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 ## Save it
13 df.write.csv(outputPath, header=True)

```

18.6 UDFs: User Defined Functions

Spark SQL provides a set of system predefined functions, which can be used in some transformations (e.g., `selectExpr()`, `sort()`) but also in the SQL queries. Some examples are

- `hour(Timestamp)`
- `abs(Integer)`
- ...

However, users can also define custom functions, which are called **User Defined Functions (UDFs)**.

UDFs are defined/registered by invoking the `udf().register(name, function, datatype)` on the `SparkSession`, where

- `name` is the name of the defined UDF
- `function` is a lambda function used to specify how the parameters of the function are used to generate the returned value
 - One or more input parameters are accepted
 - One single returned value is accepted
- `datatype` is the SQL data type of the returned value

i Example

Define a UDFs that, given a string, returns the length of the string.

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Define the UDF
5 ## name: length
6 ## output: integer value
7 spark.udf.register("length", lambda x: len(x))

```

Use of the defined UDF in a `selectExpr` transformation.

```

1 result = inputDF.selectExpr("length(name) as size")

```

Use of the defined UDF in a SQL query.

```

1 result = spark.sql("SELECT length(name) FROM profiles")

```

18.7 Other notes

Data warehouse methods: cube and rollup

The method `cube(col1, ..., coln)` of the `DataFrame` class can be used to create a multi-dimensional cube for the input DataFrame, on top of which aggregate functions can be computed for each group.

The method `rollup(col1, ..., coln)` of the `DataFrame` class can be used to create a multi-dimensional rollup for the input DataFrame, on top of which aggregate functions can be computed for each group.

Specify which attributes are used to split the input data in groups by using `cube(col1, ..., coln)` or `rollup(col1, ..., coln)`, respectively, then, apply the aggregate functions to compute for each group of the cube/rollup. The result is a DataFrame. The same aggregate functions/methods already discussed for `groupBy` can be used also for cube and rollup.

Example

1. Create a DataFrame from the “purchases.csv” file
 - The first line contains the header;
 - The others lines contain the quantities of purchased products by users: each line contains userid, productid, quantity.
2. Create a first DataFrame containing the result of the cube method. Define one group for each pair userid, productid and compute the sum of quantity in each group; 3.Create a second DataFrame containing the result of the rollup method. Define one group for each pair userid, productid and compute the sum of quantity in each group.

Input file

```
userid,productid,quantity
u1,p1,10
u1,p1,20
u1,p2,20
u1,p3,10
u2,p1,20
u2,p3,40
u2,p3,30
```

Expected output - cube

```
userid,productid,sum(quantity)
null    null    150
null    p1      50
null    p2      20
null    p3      80
u1     null    60
u1     p1      30
u1     p2      20
u1     p3      10
```

```

u2      null    90
u2      p1     20
u2      p3     70

```

Expected output - rollup

```

userid,productid,sum(quantity)
null      null    150
u1        null    60
u1        p1     30
u1        p2     20
u1        p3     10
u2        null    90
u2        p1     20
u2        p3     70

```

```

1 ## Create a Spark Session object
2 spark = SparkSession.builder.getOrCreate()
3
4 ## Read purchases.csv and store it in a DataFrame
5 dfPurchases = spark.read.load(
6     "purchases.csv",
7     format="csv",
8     header=True,
9     inferSchema=True
10 )
11
12 dfCube=dfPurchases \
13     .cube("userid","productid") \
14     .agg({"quantity": "sum"})
15
16 dfRollup=dfPurchases \
17     .rollup("userid","productid")\
18     .agg({"quantity": "sum"})

```

Set methods

Similarly to RDDs also DataFrames can be combined by using set transformations

- `df1.union(df2)`
- `df1.intersect(df2)`
- `df1.subtract(df2)`

Broadcast join

Spark SQL automatically implements a broadcast version of the join operation if one of the two input DataFrames is small enough to be stored in the main memory of each executor.

It is possible to suggest/force it by creating a broadcast version of a DataFrame.

Example

```
1 dfPersonLikesBroadcast = dfUidSports\  
2     .join(  
3         broadcast(dfPersons),  
4         dfPersons.uid == dfUidSports.uid  
5     )
```

broadcast(dfPersons)

In this case we specify that `dfPersons` must be broadcasted and hence Spark will execute the join operation by using a broadcast join.

Execution plan

The method `explain()` can be invoked on a DataFrame to print on the standard output the execution plan of the part of the code that is used to compute the content of the DataFrame on which `explain()` is invoked.

19 Spark MLlib

Spark MLlib is the Spark component providing the machine learning/data mining algorithms

- Pre-processing techniques
- Classification (supervised learning)
- Clustering (unsupervised learning)
- Itemset mining

MLlib APIs are divided into two packages:

- `pyspark.mllib`: It contains the original APIs built on top of RDDs. This version of the APIs is in maintenance mode and will be probably deprecated in the next releases of Spark.
- `pyspark.ml`: It provides higher-level API built on top of DataFrames (i.e, `Dataset<Row>`) for constructing ML pipelines. It is recommended because the DataFrame-based API is more versatile and flexible, also providing the pipeline concept. This is the one explained in this course.

19.1 Data types

Spark MLlib is based on a set of basic local and distributed data types:

- Local vector
- Local matrix
- Distributed matrix
- ...

DataFrames for ML-based applications contain objects based on these basic data types.

Local vectors

Local `pyspark.ml.linalg.Vector` objects in MLlib are used to store vectors (in dense and sparse representations) of double values. The MLlib algorithms work on vectors of doubles, used to represent the input records/data (one vector for each input record). Non double attributes/values must be mapped to double values before applying MLlib algorithms.

Example

Consider the vector of doubles [1.0, 0.0, 3.0]. It can be represented

- In dense format as [1.0, 0.0, 3.0]
- In sparse format as (3, [0, 2], [1.0, 3.0]), where

- 3 is the size of the vector
- The array [0, 2] contains the indexes of the non-zero cells
- The array [1.0, 3.0] contains the values of the non-zero cells

The following code shows how dense and sparse vectors can be created in Spark

```

1  from pyspark.ml.linalg import Vectors
2
3  ## Create a dense vector [1.0, 0.0, 3.0]
4  dv = Vectors.dense([1.0, 0.0, 3.0])
5
6  ## Create a sparse vector [1.0, 0.0, 3.0] by specifying
7  ## its indices and values corresponding to non-zero entries
8  ## by means of a dictionary
9  sv = Vectors.sparse(3, { 0:1.0, 2:3.0 })

```

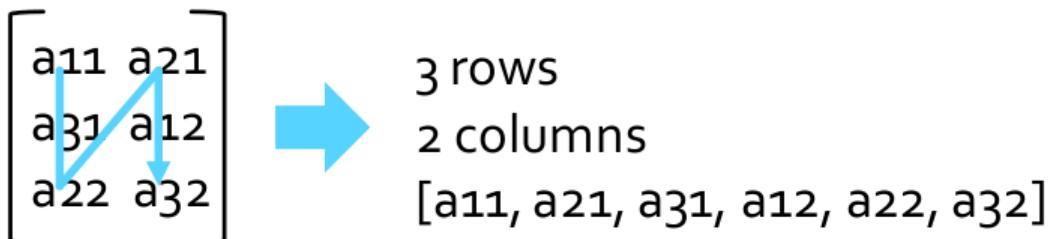
In the sparse vector

3	Size of the vector
2:3.0	Index and value of a non-empty cell
{ 0:1.0, 2:3.0 }	Dictionary of <i>index : value</i> pairs

Local matrices

Local `pyspark.ml.linalg.Matrix` objects in MLlib are used to store matrices (in dense and sparse representations) of double values. The column-major order is used to store the content of the matrix in a linear way.

Figure 19.1: Local matrices



i Example

The following code shows how dense and sparse matrices can be created in Spark.

```

1 from pyspark.ml.linalg import Matrices
2
3 ## Create a dense matrix with two rows and three columns
4 ## 3.0 0.0 0.0
5 ## 1.0 1.5 2.0
6 dm =Matrices.dense(2,3,[3.0, 1.0, 0.0, 1.5, 0.0, 2.0])
7
8 ## Create a sparse version of the same matrix
9 sm = Matrices.sparse(2,3, [0, 2, 3, 4], [0, 1, 1, 1] , [3, 1, 1.5, 2])

```

In the dense matrix vector

2	Number of rows
3	Number of columns
[3.0, 1.0, 0.0, 1.5, 0.0, 2.0]	Values in column/major order

In the sparse matrix vector

2	Number of rows
3	Number of columns
[0, 2, 3, 4]	One element per column that encodes the offset in the array of non-zero values where the values of the given column start. The last element is the number of non-zero values.
[0, 1, 1, 1]	Row index of each non-zero value
[3, 1, 1.5, 2]	Array of non-zero values of the represented matrix

19.2 Main concepts

Spark MLlib uses DataFrames as input data: the input of the MLlib algorithms are structured data (i.e., tables), and all input data must be represented by means of tables before applying the MLlib algorithms; also document collections must be transformed in a tabular format before applying the MLlib algorithms.

The DataFrames used and created by the MLlib algorithms are characterized by several columns, and each column is associated with a different role/meaning

- **label:** the target of a classification/regression analysis;
- **features:** the vector containing the values of the attributes/features of the input record/data points;
- **text:** the original text of a document before being transformed in a tabular format;
- **prediction:** the predicted value of a classification/regression analysis.

Transformer

A Transformer is an ML algorithm/procedure that transforms one DataFrame into another DataFrame by means of the method `transform(inputDataFrame)`.

i Example 1

A feature transformer might take a DataFrame, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new DataFrame with the mapped column appended.

i Example 2

A classification model is a Transformer that can be applied on a DataFrame with features and transforms it into a DataFrame with also the prediction column.

Estimator

An Estimator is an ML algorithm/procedure that is fit on an input (training) DataFrame to produce a Transformer: each Estimator implements a `fit()` method, which accepts a DataFrame and produces a Model of type **Transformer**.

An Estimator abstracts the concept of a learning algorithm or any algorithm that fits/trains on an input dataset and returns a model

i Example

The Logistic Regression classification algorithm is an Estimator: calling `fit(input training DataFrame)` on it a Logistic Regression Model is built, which is a Model/a Transformer.

Pipeline

A Pipeline chains multiple Transformers and Estimators together to specify a Machine learning/Data Mining workflow. In a pipeline, the output of a transformer/estimator is the input of the next one.

i Example

A simple text document processing workflow aiming at building a classification model includes several steps

1. Split each document into a set of words;
2. Convert each set of words into a numerical feature vector;
3. Learn a prediction model using the feature vectors and the associated class labels.

Parameters

Transformers and Estimators share common APIs for specifying the values of their parameters.

In the new APIs of Spark MLlib the use of the pipeline approach is preferred/recommended. This approach is based on the following steps

1. The set of Transformers and Estimators that are needed are instantiated;
2. A pipeline object is created and the sequence of transformers and estimators associated with the pipeline are specified;
3. The pipeline is executed and a model is trained;
4. (optional) The model is applied on new data.

19.3 Data preprocessing

Input data must be preprocessed before applying machine learning and data mining algorithms

- To organize data in a format consistent with the one expected by the applied algorithms;
- To define good (predictive) features;
- To remove bias (e.g., normalization);
- To remove noise and missing values.

Extracting, transformings, and selecting features

MLlib provides a set of transformers than can be used to extract, transform and select features from DataFrames

- Feature Extractors (e.g., TF-IDF, Word2Vec)
- Feature Transformers (e.g., Tokenizer, StopWordsRemover, StringIndexer, IndexToString, OneHotEncoderEstimator, Normalizer)
- Feature Selectors (e.g., VectorSlicer)

See the up-to-date list [here](#).

19.4 Feature transformations

Several algorithms are provided by MLlib to transform features. They are used to create new columns/features by combining or transforming other features It is possible to perform feature transformations and feature creations by using the standard methods for DataFrames and RDDs.

VectorAssembler

`VectorAssembler` (`pyspark.ml.feature.VectorAssembler`) is a transformer that combines a given list of columns into a single vector column. It is useful for combining features into a single feature vector before applying ML algorithms.

Given `VectorAssembler(inputCols, outputCol)`

- `inputCols`: the list of original columns to include in the new column of type `Vector`. The following input column types are accepted
 - all numeric types, boolean type, and vector type
 - Boolean values are mapped to 1 (True) and 0 (False)
- `outputCol`: the name of the new output column

When the `transform` method of `VectorAssembler` is invoked on a `DataFrame` the returned `DataFrame` has a new column (`outputCol`): for each record, the value of the new column is the concatenation of the values of the input columns. It has also all the columns of the input `DataFrame`.

i Example

Consider an input `DataFrame` with three columns: create a new `DataFrame` with a new column containing the concatenation of “`colB`” and “`colC`” in a new vector column. Set the name of the new column to “`features`”.

Original DataFrame

	colA	colB	colC
1	4.5	True	
2	0.6	True	
3	1.5	False	
4	12.1	True	
5	0.0	True	

Transformed DataFrame

	colA	colB	colC	features
1	4.5	True	[4.5, 1.0]	
2	0.6	True	[0.6, 1.0]	
3	1.5	False	[1.5, 0.0]	
4	12.1	True	[12.1, 1.0]	
5	0.0	True	[0.0, 1.0]	

Notice that columns of DataFrames can also be vectors.

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3
4  ## input and output folders
5  inputPath = "data/exampleDataAssembler.csv"
6
7  ## Create a DataFrame from the input data
8  inputDF = spark.read.load(
9      inputPath,
10     format="csv",
11     header=True,
12     inferSchema=True
13 )
14
15 ## Create a VectorAssembler that combines columns colB and colC
16 ## The new vector column is called features
17 myVectorAssembler = VectorAssembler(
18     inputCols = ['colB', 'colC'],
19     outputCol = 'features'
20 )
21
22 ## Apply myVectorAssembler on the input DataFrame
23 transformedDF = myVectorAssembler.transform(inputDF)

```

Data Normalization

MLlib provides a set of normalization algorithms (called scalers)

- StandardScaler
- MinMaxScaler
- Normalizer
- MaxAbsScaler

StandardScaler

`StandardScaler` (`pyspark.ml.feature.StandardScaler`) is an Estimator that returns a Transformer (`pyspark.ml.feature.StandardScalerModel`). `StandardScalerModel` transforms a vector column of an input DataFrame normalizing each feature of the input vector column to have unit standard deviation and/or zero mean.

Given `StandardScaler(inputCol, outputCol)`

- `inputCol`: the name of the input vector column (of doubles) to normalize
- `outputCol`: the name of the new output normalized vector column

Invoke the `fit` method of `StandardScaler` on the input DataFrame to infer a `StandardScalerModel`. The returned model is a Transformer.

Invoke the `transform` method of `StandardScalerModel` on the input DataFrame to create a new DataFrame that has a new column (`outputCol`): for each record, the value of the new column is the normalized version of the input vector column. It has also all the columns of the input DataFrame.

i Example

Consider an input DataFrame with four columns: create a new DataFrame with a new column containing the normalized version of the vector column features. Set the name of the new column to “scaledFeatures”.

Original DataFrame

	colA	colB	colC	features
1	4.5	True	[4.5, 1.0]	
2	0.6	True	[0.6, 1.0]	
3	1.5	False	[1.5, 0.0]	
4	12.1	True	[12.1, 1.0]	
5	0.0	True	[0.0, 1.0]	

Transformed DataFrame

colA	colB	colC	features	scaledFeatures
1	4.5	True	[4.5, 1.0]	[0.903, 2.236]
2	0.6	True	[0.6, 1.0]	[0.120, 2.236]
3	1.5	False	[1.5, 0.0]	[0.301, 0.0]
4	12.1	True	[12.1, 1.0]	[2.428, 2.236]
5	0.0	True	[0.0, 1.0]	[0.0, 2.236]

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.feature import StandardScaler
4  ## input and output folders
5  inputPath = "data/exampleDataAssembler.csv"
6
7  ## Create a DataFrame from the input data
8  inputDF = spark.read.load(
9      outputPath,
10     format="csv",
11     header=True,
12     inferSchema=True
13 )
14
15 ## Create a VectorAssembler that combines columns colB and colC
16 ## The new vector column is called features
17 myVectorAssembler = VectorAssembler(
18     inputCols = ['colB', 'colC'],
19     outputCol = 'features'
20 )
21
22 ## Apply myVectorAssembler on the input DataFrame
23 transformedDF = myVectorAssembler.transform(inputDF)
24
25 ## Create a Standard Scaler to scale the content of features
26 myScaler = StandardScaler(
27     inputCol="features",
28     outputCol="scaledFeatures"
29 )
30
31 ## Compute summary statistics by fitting the StandardScaler
32 ## Before normalizing the content of the data we need to compute mean and
33 ## standard deviation of the analyzed data
34 scalerModel = myScaler.fit(transformedDF)
35
36 ## Apply myScaler on the input column features
37 scaledDF = scalerModel.transform(transformedDF)

```

Categorical columns

Frequently the input data are characterized by categorical attributes (i.e., string columns), and the class label of the classification problem is a categorical attribute. The Spark MLlib classification and regression algorithms work only with numerical values, so categorical columns must be mapped to double values.

StringIndexer

A `StringIndexer` (`pyspark.ml.feature.StringIndexer`) is an Estimator that returns a Transformer of type `pyspark.ml.feature.StringIndexerModel`. `StringIndexerModel` encodes a string column of “labels” to a column of “label indices”: each distinct value of the input string column is mapped to an integer value in [0, **number of distinct values**).

`StringIndexer(inputCol, outputCol)`

- `inputCol`: the name of the input string column to map to a set of integers
- `outputCol`: the name of the new output column

Invoke the `fit` method of `StringIndexer` on the input DataFrame to infer a `StringIndexerModel`. The returned model is a Transformer.

Invoke the `transform` method of `StringIndexerModel` on the input DataFrame to create a new DataFrame that has a new column (`outputCol`): for each record, the value of the new column is the integer (casted to a double) associated with the value of the input string column. It has also all the columns of the input DataFrame.

i Example

Consider an input DataFrame with two columns: create a new DataFrame with a new column containing the integer version of the string column category. Set the name of the new column to “`categoryIndex`”.

Original DataFrame

	id	category
	1	a
	2	b
	3	c
	4	c
	5	a

Transformed DataFrame

	id	category	categoryIndex
1	1	a	0.0
2	2	b	2.0
3	3	c	1.0
4	4	c	1.0
5	5	a	0.0

```

1 from pyspark.mllib.linalg import Vectors
2 from pyspark.ml.feature import StringIndexer
3
4 ## input DataFrame
5 df = spark.createDataFrame(
6     [(1,"a"),(2,"b"),(3,"c"),(4,"c"),(5,"a")],
7     ["id","category"]
8 )
9
10 ## Create a StringIndexer to map the content of category
11 ## to a set of "integers"
12 indexer = StringIndexer(
13     inputCol="category",
14     outputCol="categoryIndex"
15 )
16
17 ## Analyze the input data to define the mapping string -> integer
18 indexerModel = indexer.fit(df)
19
20 ## Apply indexerModel on the input column category
21 indexedDF = indexerModel.transform(df)

```

IndexToString

`IndexToString` (`pyspark.ml.feature.IndexToString`), which is symmetrical to `StringIndexer`, is a Transformer that maps a column of “label indices” back to a column containing the original “labels” as strings. Classification models return the integer version of the predicted label values. To obtain human readable results, remap those values to the original ones.

Given `IndexToString(inputCol, outputCol, labels)`

- `inputCol`: the name of the input numerical column to map to the original a set of string “labels”;
- `outputCol`: the name of the new output column;
- `labels`: the list of original “labels”/strings; the mapping with integer values is given by the positions of the strings inside labels.

Invoke the transform method of `IndexToString` on the input DataFrame to create a new DataFrame that has a new column (`outputCol`): for each record, the value of the new column is the original string associated with the value of the input numerical column. It has also all the columns of the input DataFrame.

Example

Consider an input DataFrame with two columns: create a new DataFrame with a new column containing the integer version of the string column category and then map it back to the string version in a new column.

Original DataFrame

	id	category
	1	a
	2	b
	3	c
	4	c
	5	a

Transformed DataFrame

	id	category	categoryIndex	originalCategory
1	1	a	0.0	a
2	2	b	2.0	b
3	3	c	1.0	c
4	4	c	1.0	c
5	5	a	0.0	a

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import StringIndexer
3  from pyspark.ml.feature import IndexToString
4
5  ## input DataFrame
6  df = spark.createDataFrame(
7      [(1,"a"),(2,"b"),(3,"c"),(4,"c"),(5,"a")],
8      ["id","category"]
9  )
10
11 ## Create a StringIndexer to map the content of category
12 ## to a set of integers
13 indexer = StringIndexer(
14     inputCol="category",
15     outputCol="categoryIndex"
16 )
17
18 ## Analyze the input data to define the mapping
19 ## string -> integer
20 indexerModel = indexer.fit(df)
21
22 ## Apply indexerModel on the input column category
23 indexedDF = indexerModel.transform(df)
24
25 ## Create an IndexToString to map the content of numerical
26 ## attribute categoryIndex to the original string value
27 converter = IndexToString(
28     inputCol="categoryIndex",
29     outputCol="originalCategory",
30     labels=indexerModel.labels
31 )
32
33 ## Apply converter on the input column categoryIndex
34 reconvertedDF = converter.transform(indexedDF)

```

SQLTransformer

`SQLTransformer` (`pyspark.ml.feature.SQLTransformer`) is a transformer that implements the transformations which are defined by SQL queries. Currently, the syntax of the supported (simplified) SQL queries is

```
1 SELECT attributes, function(attributes) FROM __THIS__
```

Where `__THIS__` represents the DataFrame on which the `SQLTransformer` is invoked.

`SQLTransformer` executes an SQL query on the input DataFrame and returns a new DataFrame associated with the result of the query.

Given `SQLTransformer(statement)`

- `statement`: the SQL query to execute.

When the `transform` method of `SQLTransformer` is invoked on a DataFrame the returned DataFrame is the result of the executed statement query.

i Example

Consider an input DataFrame with two columns: “text” and “id”: create a new DataFrame with a new column, called “numWords”, containing the number of words occurring in column “text”.

Original DataFrame

id	text
1	This is Spark
2	Spark
3	Another sample sentence of words
4	Paolo Rossi
5	Giovanni

Transformed DataFrame

id	text	numWords
1	This is Spark	3
2	Spark	1
3	Another sample sentence of words	5
4	Paolo Rossi	2
5	Giovanni	1

```
1  from pyspark.sql.types import *
2  from pyspark.ml.feature import SQLTransformer
3
4  #Local Input data
5  inputList = [
6      (1,"ThisisSpark"),
7      (2,"Spark"),
8      (3,"Anothersamplesentenceofwords"),
9      (4,"PaoloRossi"),
10     (5,"Giovanni")
11 ]
12
13 ## Create the initial DataFrame
14 dfInput = spark.createDataFrame(
15     inputList,
16     ["id","text"]
17 )
18
19 ## Define a UDF function that counts the number of words
20 ## in an input string
21 spark.udf.register(
22     "countWords",
23     lambda text: len(text.split(" ")),
24     IntegerType()
25 )
26
27 ## Define an SQLTransformer to create the columns we are
28 ## interested in
29 sqlTrans = SQLTransformer(
30     statement="""
31     SELECT *, countWords(text) AS numLines
32     FROM __THIS__
33     """
34 )
35
36 ## Create the new DataFrame by invoking the transform method of
37 ## the defined SQLTransformer
38 newDF = sqlTrans.transform(dfInput)
```

20 Classification algorithms

Spark MLlib provides a (limited) set of classification algorithms

- Logistic regression
 - Binomial logistic regression
 - Multinomial logistic regression
- Decision tree classifier
- Random forest classifier
- Gradient-boosted tree classifier
- Multilayer perceptron classifier
- Linear Support Vector Machine

All the available classification algorithms are based on two phases:

1. Model generation based on a set of training data
2. Prediction of the class label of new unlabeled data

All the classification algorithms available in Spark work only on numerical attributes: categorical values must be mapped to integer values (one distinct value per class) before applying the MLlib classification algorithms.

All the Spark classification algorithms are trained on top of an input DataFrame containing (at least) two columns

- label: the class label, (i.e., the attribute to be predicted by the classification model); it is an integer value (casted to a double)
- features: a vector of doubles containing the values of the predictive attributes of the input records/data points; the data type of this column is `pyspark.ml.linalg.Vector`, and both dense and sparse vectors can be used

i Example

Consider the following classification problem: the goal is to predict if new customers are good customers or not based on their monthly income and number of children.

The predictive attributes are

- Monthly income
- Number of children

The class label (target attribute) is “Customer type”:

- “Good customer”, mapped to 1

- “Bad customer”, mapped to 0

Example of input training data

The training data is the set of customers for which the value of the class label is known: they are used by the classification algorithm to infer/train a classification model.

CustomerType	MonthlyIncome	NumChildren
Good customer	1400.0	2
Bad customer	11105.5	0
Good customer	2150.0	2

Example of input training DataFrame

The input training DataFrame that must be provided as input to train an MLlib classification algorithm must have the following structure

label	features
1.0	[1400.0, 2.0]
0.0	[11105.5, 0.0]
1.0	[2150.0, 2.0]

Notice that

- The categorical values of “CustomerType” (the class label column) must be mapped to integer data values (then casted to doubles).
- The values of the predictive attributes are stored in vectors of doubles. One single vector for each input record.
- In the generated DataFrame the names of the predictive attributes are not preserved.

20.1 Structured data classification

Example of logistic regression and structured data

The following paragraphs show how to

- Create a classification model based on the logistic regression algorithm on structured data: the model is inferred by analyzing the training data, (i.e., the example records/data points for which the value of the class label is known).
- Apply the model to new unlabeled data: the inferred model is applied to predict the value of the class label of new unlabeled records/data points.

Training data

The input training data is stored in a text file that contains one record/data point per line. The records/data points are structured data with a fixed number of attributes (four)

- One attribute is the class label: it assumed that the first column of each record contains the class label;
- The other three attributes are the predictive attributes that are used to predict the value of the class label;

The values are already doubles (no need to convert them), and the input file has the header line.

Consider the following example input training data file

```
label,attr1,attr2,attr3
1.0,0.0,1.1,0.1
0.0,2.0,1.0,-1.0
0.0,2.0,1.3,1.0
1.0,0.0,1.2,-0.5
```

It contains four records/data points. This is a binary classification problem because the class label assumes only two values: 0 and 1.

The first operation consists in transforming the content of the input training file into a DataFrame containing two columns

- label: the double value that is used to specify the label of each training record;
- features: it is a vector of doubles associated with the values of the predictive features.

label	features
1.0	[0.0, 1.1, 0.1]
0.0	[2.0, 1.0, -1.0]
0.0	[2.0, 1.3, 1.0]
1.0	[0.0, 1.2, -0.5]

- Data type of “label” is double
- Data type of “features” is `pyspark.ml.linalg.Vector`

Unlabeled data

The file containing the unlabeled data has the same format of the training data file, however the first column is empty because the class label is unknown. The goal is to predict the class label value of each unlabeled data by applying the classification model that has been trained on the training data: the predicted class label value of the unlabeled data is stored in a new column, called “prediction”, of the returned DataFrame.

Consider the following input unlabeled data file

```
label,attr1,attr2,attr3
,-1.0,1.5,1.3
,3.0,2.0,-0.1
,0.0,2.2,-1.5
```

It contains three unlabeled records/data points. Notice that the first column is empty (the content before the first comma is the empty string).

Also the unlabeled data must be stored into a DataFrame containing two columns: “label” and “features”. So, “label” column is required also for unlabeled data, but its value is set to null for all records.

label	features
null	[−1.0, 1.5, 1.3]
null	[3.0, 2.0, −0.1]
null	[0.0, 2.2, −1.5]

Prediction column

After the application of the classification model on the unlabeled data, Spark returns a new DataFrame containing

- The same columns of the input DataFrame
- A new column called prediction, that, for each input unlabeled record, contains the predicted class label value
- Two columns, associated with the probabilities of the predictions (these columns are not considered in the example)

label	features	prediction	rawPrediction	probability
null	[−1.0, 1.5, 1.3]	1.0
null	[3.0, 2.0, −0.1]	0.0
null	[0.0, 2.2, −1.5]	1.0

The “prediction” column contains the predicted class label values.

Example code

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.classification import LogisticRegression
4
5  ## input and output folders
6  trainingData = "ex_data/trainingData.csv"
7  unlabeledData = "ex_data/unlabeledData.csv"
8  outputPath = "predictionsLR/"
9
10 ## ****
11 ## Training step
12 ## ****
13

```

```
14 ## Create a DataFrame from trainingData.csv
15 ## Training data in raw format
16 trainingData = spark.read.load(
17     trainingData,
18     format="csv",
19     header=True,
20     inferSchema=True
21 )
22
23 ## Define an assembler to create a column (features) of type Vector
24 ## containing the double values associated with columns attr1, attr2, attr3
25 assembler = VectorAssembler(
26     inputCols=["attr1","attr2","attr3"],
27     outputCol="features"
28 )
29
30 ## Apply the assembler to create column features for the training data
31 trainingDataDF = assembler.transform(trainingData)
32
33 ## Create a LogisticRegression object.
34 ## LogisticRegression is an Estimator that is used to
35 ## create a classification model based on logistic regression.
36 lr = LogisticRegression()
37
38 ## It is possible to set the values of the parameters of the
39 ## Logistic Regression algorithm using the setter methods.
40 ## There is one set method for each parameter
41 ## For example, the number of maximum iterations is set to 10
42 ## and the regularization parameter is set to 0.01
43 lr.setMaxIter(10)
44 lr.setRegParam(0.01)
45
46 ## Train a logistic regression model on the training data
47 classificationModel = lr.fit(trainingDataDF)
48
49 #### ****
50 ## Prediction step
51 #### ****
52
53 ## Create a DataFrame from unlabeledData.csv
54 ## Unlabeled data in raw format
55 unlabeledData = spark.read.load(
56     unlabeledData,
57     format="csv",
58     header=True,
59     inferSchema=True
60 )
```

```
61
62 ## Apply the same assembler we created before also on the unlabeled data
63 ## to create the features column
64 unlabeledDataDF = assembler.transform(unlabeledData)
65
66 ## Make predictions on the unlabeled data using the transform() method of the
67 ## trained classification model transform uses only the content of 'features'
68 ## to perform the predictions
69 predictionsDF = classificationModel.transform(unlabeledDataDF)
70
71 ## The returned DataFrame has the following schema (attributes)
72 ## - attr1
73 ## - attr2
74 ## - attr3
75 ## - features: vector (values of the attributes)
76 ## - label: double (value of the class label)
77 ## - rawPrediction: vector (nullable = true)
78 ## - probability: vector (The i-th cell contains the probability that
79 ## the current record belongs to the i-th class
80 ## - prediction: double (the predicted class label)
81
82 ## Select only the original features (i.e., the value of the original attributes
83 ## attr1, attr2, attr3) and the predicted class for each record
84 predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")
85
86 ## Save the result in an HDFS output folder
87 predictions.write.csv(outputPath, header="true")
```

Pipelines

In the previous solution the same preprocessing steps were applied on both training and unlabeled data (the same assembler on both input data). It is possible to use a pipeline to specify the common phases we apply on both input data sets.

 Example

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.classification import LogisticRegression
4  from pyspark.ml import Pipeline
5  from pyspark.ml import PipelineModel
6
7  ## input and output folders
8  trainingData = "ex_data/trainingData.csv"
9  unlabeledData = "ex_data/unlabeledData.csv"
10 outputPath = "predictionsLR/"
11
12 #### ****
13 ## Training step
14 #### ****
15 ## Create a DataFrame from trainingData.csv
16 ## Training data in raw format
17 trainingData = spark.read.load(
18     trainingData,
19     format="csv",
20     header=True,
21     inferSchema=True
22 )
23
24 ## Define an assembler to create a column (features) of type Vector
25 ## containing the double values associated with columns attr1, attr2, attr3
26 assembler = VectorAssembler(
27     inputCols=["attr1","attr2","attr3"],
28     outputCol="features"
29 )
30
31 ## Create a LogisticRegression object
32 ## LogisticRegression is an Estimator that is used to
33 ## create a classification model based on logistic regression.
34 lr = LogisticRegression()
35
36 ## Set the values of the parameters of the
37 ## Logistic Regression algorithm using the setter methods.
38 ## There is one set method for each parameter
39 ## For example, we are setting the number of maximum iterations to 10
40 ## and the regularization parameter to 0.01
41 lr.setMaxIter(10)
42 lr.setRegParam(0.01)
43
44 ## Define a pipeline that is used to create the logistic regression
45 ## model on the training data. The pipeline includes also
46 ## the preprocessing step
47 pipeline = Pipeline().setStages([assembler, lr])(1)
48
49 ## Execute the pipeline on the training data to build the
50 ## classification model
51 classificationModel = pipeline.fit(trainingData)
52

```

① **assembler:** the sequence of transformers and estimators to apply on the input data

Decision trees and structured data

The following paragraphs show how to

- Create a classification model based on the decision tree algorithm on structured data: the model is inferred by analyzing the training data, i.e., the example records/data points for which the value of the class label is known;
- Apply the model to new unlabeled data: the inferred model is applied to predict the value of the class label of new unlabeled records/data points.

The same example structured data already used in the running example related to the logistic regression algorithm are used also in this example related to the decision tree algorithm. The main steps are the same of the previous example, the only difference is the definition and configuration of the used classification algorithm.

 Example

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.classification import DecisionTreeClassifier
4  from pyspark.ml import Pipeline
5  from pyspark.ml import PipelineModel
6
7  ## input and output folders
8  trainingData = "ex_data/trainingData.csv"
9  unlabeledData = "ex_data/unlabeledData.csv"
10 outputPath = "predictionsLR/"
11
12 #### ****
13 ## Training step
14 #### ****
15 ## Create a DataFrame from trainingData.csv
16 ## Training data in raw format
17 trainingData = spark.read.load(
18     trainingData,
19     format="csv",
20     header=True,
21     inferSchema=True
22 )
23
24 ## Define an assembler to create a column (features) of type Vector
25 ## containing the double values associated with columns attr1, attr2, attr3
26 assembler = VectorAssembler(
27     inputCols=["attr1","attr2","attr3"],
28     outputCol="features"
29 )
30
31 ## Create a DecisionTreeClassifier object.
32 ## DecisionTreeClassifier is an Estimator that is used to
33 ## create a classification model based on decision trees.
34 dt = DecisionTreeClassifier()
35
36 ## We can set the values of the parameters of the Decision Tree
37 ## For example we can set the measure that is used to decide if a
38 ## node must be split. In this case we set gini index
39 dt.setImpurity("gini")
40
41 ## Define a pipeline that is used to create the decision tree
42 ## model on the training data. The pipeline includes also
43 ## the preprocessing step
44 pipeline = Pipeline().setStages([assembler, dt])
45
46 ## Execute the pipeline on the training data to build the
47 ## classification model
48 classificationModel = pipeline.fit(trainingData)
49
50 ## Now, the classification model can be used to predict the class label
51 ## of new unlabeled data
52

```

- ① **assembler**: the sequence of transformers and estimators to apply on the input data. A decision tree algorithm is used in this case.

20.2 Categorical class labels

Usually the class label is a categorical value (i.e., a string). However, as reported before, Spark MLlib works only with numerical values and hence categorical class label values must be mapped to integer (and then double) values: processing and postprocessing steps are used to manage this transformation.

Consider the following input training data

categoricalLabel	Attr1	Attr2	Attr3
Positive	0.0	1.1	0.1
Negative	2.0	1.0	-1.0
Negative	2.0	1.3	1.0

A modified input DataFrame must be generated as input for the MLlib classification algorithms

label	features
1.0	[0.0, 1.1, 0.1]
1.0	[2.0, 1.0, -1.0]
0.0	[2.0, 1.3, 1.0]

Notice that the categorical values of “categoricalLabel” (the class label column) must be mapped to integer data values (finally casted to doubles).

StringIndexer and IndexToString

The Estimator **StringIndexer** and the Transformer **IndexToString** support the transformation of categorical class label into numerical one and vice versa:

- **StringIndexer** maps each categorical value of the class label to an integer (then casted to a double);
- **IndexToString** is used to perform the opposite operation.

All in all, these are the main steps

1. Use **StringIndexer** to extend the input DataFrame with a new column, called “label”, containing the numerical representation of the class label column;
2. Create a column, called “features”, of type vector containing the predictive features;
3. Infer a classification model by using a classification algorithm (e.g., Decision Tree, Logistic regression);
4. Apply the model on a set of unlabeled data to predict their numerical class label;
5. Use **IndexToString** to convert the predicted numerical class label values to the original categorical values.

Notice that the model is built by considering only the values of features and label. All the other columns are not considered by the classification algorithm during the generation of the prediction model.

Training data

Given the following input training file

```
categoricalLabel,attr1,attr2,attr3  
Positive,0.0,1.1,0.1  
Negative,2.0,1.0,-1.0  
Negative,2.0,1.3,1.0
```

The initial training DataFrame will be

categoricalLabel	features
Positive	[0.0, 1.1, 0.1]
Negative	[2.0, 1.0, -1.0]
Negative	[2.0, 1.3, 1.0]

- The type of “categoricalLabel” is String
- The type of “features” is Vector

After applying `StringIndexer`, the training DataFrame will be

categoricalLabel	features	label!
Positive	[0.0, 1.1, 0.1]	1.0
Negative	[2.0, 1.0, -1.0]	0.0
Negative	[2.0, 1.3, 1.0]	0.0

“label” contains the mapping generated by `StringIndexer`:

- “Positive”: 1.0
- “Negative”: 0.0

Unlabeled data

Given the input unlabeled data file

```
categoricalLabel,attr1,attr2,attr3  
, -1.0, 1.5, 1.3  
, 3.0, 2.0, -0.1  
, 0.0, 2.2, -1.5
```

The initial unlabeled DataFrame will be

categoricalLabel	features
null	[−1.0, 1.5, 1.3]
null	[3.0, 2.0, −0.1]
null	[0.0, 2.2, −1.5]

After performing the prediction, and applying `IndexToString`, the output DataFrame will be

categoricalLabel	features	label	prediction	predictedLabel	...
...	[−1.0, 1.5, 1.3]	...	1.0	Positive	
...	[3.0, 2.0, −0.1]	...	0.0	Negative	
...	[0.0, 2.2, −1.5]	...	1.0	Negative	

- “prediction” contains the predicted label, expressed as a number
- “predictedLabel” contains the predicted label, expressed as a category (original name)

i Example

In this example, the input training data is stored in a text file that contains one record/data point per line, and the records/data points are structured data with a fixed number of attributes (four)

- One attribute is the class label (“categoricalLabel”): this is a categorical attribute that can assume two values, “Positive” or “Negative”;
- The other three attributes (“attr1”, “attr2”, “attr3”) are the predictive attributes that are used to predict the value of the class label.

The input file has the header line.

The file containing the unlabeled data has the same format of the training data file, however the first column is empty because the class label is unknown.

The goal is to predict the class label value of each unlabeled data by applying the classification model that has been inferred on the training data.

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.feature import StringIndexer
4  from pyspark.ml.feature import IndexToString
5  from pyspark.ml.classification import DecisionTreeClassifier
6  from pyspark.ml import Pipeline
7  from pyspark.ml import PipelineModel
8
9  ## input and output folders
10 trainingData = "ex_dataCategorical/trainingData.csv"
11 unlabeledData = "ex_dataCategorical/unlabeledData.csv"
12 outputPath = "predictionsDTCTegoricalPipeline/"
13
14 ##### ****
15 ## Training step
16 ##### ****
17
18 ## Create a DataFrame from trainingData.csv
19 ## Training data in raw format
20 trainingData = spark.read.load(trainingData,\n21 format="csv", header=True, inferSchema=True)
22
23 ## Define an assembler to create a column (features) of type Vector
24 ## containing the double values associated with columns attr1, attr2, attr3
25 assembler = VectorAssembler(
26     inputCols=["attr1","attr2","attr3"],
27     outputCol="features"
28 )
29
30 ## The StringIndexer Estimator is used to map each class label
31 ## value to an integer value (casted to a double).
32 ## A new attribute called label is generated by applying
33 ## transforming the content of the categoricalLabel attribute.
34 labelIndexer = StringIndexer(
35     inputCol="categoricalLabel" ①
36     outputCol="label",
37     handleInvalid="keep"
38 ).fit(trainingData)
39
40 ## Create a DecisionTreeClassifier object.
41 ## DecisionTreeClassifier is an Estimator that is used to
42 ## create a classification model based on decision trees.
43 dt = DecisionTreeClassifier()
44
45 ## Set the values of the parameters of the Decision Tree
46 ## For example set the measure that is used to decide if a
47 ## node must be split.
48 ## In this case we set gini index
49 dt.setImpurity("gini")
50
51 ## At the end of the pipeline we must convert indexed labels back
52 ## to original labels (from numerical to string).

```

- ① This `StringIndexer` estimator is used to infer a transformer that maps the categorical values of column “categoricalLabel” to a set of integer values stored in the new column called “label”. The list of valid label values are extracted from the training data.
- ② This `IndexToString` component is used to remap the numerical predictions available in the “prediction” column to the original categorical values that are stored in the new column called “predictedLabel”. The mapping of integer to original string value is the one of “labelIndexer”.
- ③ This `Pipeline` is composed of four steps.
- ④ The “predictedLabel” field is the column containing the predicted categorical class label for the unlabeled data.

20.3 Textual data management and classification

The following paragraphs show how to

- Create a classification model based on the logistic regression algorithm for textual documents: a set of specific preprocessing estimators and transformers are used to preprocess textual data.
- Apply the model to new textual documents

The input training dataset represents a textual document collection, where each line contains one document and its class

- The class label
- A list of words (the text of the document)

i Example

Given the following example training file

```
Label,Text
1,The Spark system is based on scala
1,Spark is a new distributed system
0,Turin is a beautiful city
0,Turin is in the north of Italy
```

It contains four textual documents, and each line contains two attributes, that are the class label (first attribute) and the text of the document (second attribute).

The input data before preprocessing, represented as a DataFrame, is

Label	Text
1	The Spark system is based on scala
1	Spark is a new distributed system
0	Turin is a beautiful city
0	Turin is in the north of Italy

A set of preprocessing steps must be applied on the textual attribute before generating a classification model.

1. Since Spark ML algorithms work only on “Tables” and double values, the textual part of the input data must be translated in a set of attributes to represent the data as a table: usually a table with an attribute for each word is generated.
2. Many words are useless (e.g., conjunctions): stopwords are usually removed. In general,
 - the words appearing in almost all documents are not characterizing the data, and so they are not very important for the classification problem;
 - the words appearing in few documents allow to distinguish the content of those documents (and hence the class label) with respect to the others, and so they are very important for the classification problem.
3. Traditionally a weight, based on the TF-IDF measure, is used to assign a different importance to the words based on their frequency in the collection.

Example

Input data after the preprocessing transformations (tokenization, stopword removal, TF-IDF computation)

	Label	Spark	system	scala	...
1	0.5	0.3	0.75	...	
1	0.5	0.3	0	...	
0	0	0	0	...	
0	0	0	0	...	

The DataFrame associated with the input data after the preprocessing transformations must contain, as usual, the columns

- label: class label value
- features: the preprocessed version of the input text

There are also some other intermediate columns, related to applied transformations, but they are not considered by the classification algorithm.

Example

The DataFrame associated with the input data after the preprocessing transformations

label	features	text
1	[0.5, 0.3, 0.75, ...]	The Spark system is based on scala
1	[0.5, 0.3, 0, ...]	Spark is a new distributed system
0	[0, 0, 0, ...]	Turin is a beautiful city
0	[0, 0, 0, ...]	Turin is in the north of Italy

Only “label” and “features” are considered by the classification algorithm.

In the following solution we will use a set of new Transformers to prepare input data

- **Tokenizer**: to split the input text in words;
- **StopWordsRemover**: to remove stopwords;
- **HashingTF**: to compute the (approximate) term frequency of each input term;
- **IDF**: to compute the inverse document frequency of each input word.

The input data (training and unlabeled data) are stored in input csv files. Each line contains two attributes:

- The class label (label)
- The text of the document (text)

We infer a linear regression model on the training data and apply the model on the unlabeled data.

 Example

```
1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.feature import Tokenizer
4  from pyspark.ml.feature import StopWordsRemover
5  from pyspark.ml.feature import HashingTF
6  from pyspark.ml.feature import IDF
7  from pyspark.ml.classification import LogisticRegression
8  from pyspark.ml import Pipeline
9  from pyspark.ml import PipelineModel
10
11 ## input and output folders          261
12 trainingData = "ex_dataText/trainingData.csv"
13 unlabeledData = "ex_dataText/unlabeledData.csv"
```

20.4 Performance evaluation

In order to test the goodness of algorithms there are some evaluators. The Evaluator can be

- a `BinaryClassificationEvaluator` for binary data
- a `MulticlassClassificationEvaluator` for multiclass problems

Provided metrics are:

- Accuracy
- Precision
- Recall
- F-measure

Use the `MulticlassClassificationEvaluator` estimator from `pyspark.ml.evaluator` on a DataFrame. The instantiated estimator has the method `.evaluate()` that is applied on a DataFrame: it compares the predictions with the true label values, and the output is the double value of the computed performance metric.

The parameters of `MulticlassClassificationEvaluator` are

- `metricName`: type of metric to compute. It can assume the following values
 - "accuracy"
 - "f1"
 - "weightedPrecision"
 - "weightedRecall"
- `labelCol`: input column with the true label/class value
- `predictionCol`: input column with the predicted class/label value

Example

In this example, the set of labeled data is read from a text file that contains one record/data point per line, and the records/data points are structured data with a fixed number of attributes (four)

- One attribute is the class label ("label");
- The other three attributes ("attr1", "attr2", "attr3") are the predictive attributes that are used to predict the value of the class label.

All attributes are already double attributes, and the input file has the header line.

Consider the following example input labeled data file

```
label,attr1,attr2,attr3
1,0.0,1.1,0.1
0,2.0,1.0,-1.0
0,2.0,1.3,1.0
1,0.0,1.2,-0.5
```

Follow these steps

1. Split the labeled data set in two subsets

- Training set: 75% of the labeled data
 - Test set: 25% of the labeled data
2. Infer/train a logistic regression model on the training set
 3. Evaluate the prediction quality of the inferred model on both the test set and the training set

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.classification import LogisticRegression
4  from pyspark.ml.evaluation import MulticlassClassificationEvaluator
5  from pyspark.ml import Pipeline
6  from pyspark.ml import PipelineModel
7
8  ## input and output folders
9  labeledData = "ex_dataValidation/labeledData.csv"
10 outputPath = "predictionsLRPipelineValidation/"
11
12 ## Create a DataFrame from labeledData.csv
13 ## Training data in raw format
14 labeledDataDF = spark.read.load(
15     labeledData,
16     format="csv",
17     header=True,
18     inferSchema=True
19 )
20
21 ## Split labeled data in training and test set
22 ## training data : 75%
23 ## test data: 25%
24 trainDF, testDF = labeledDataDF.randomSplit([0.75, 0.25], seed=10) ①
25
26 #### ****
27 ## Training step
28 #### ****
29 ## Define an assembler to create a column (features) of type Vector
30 ## containing the double values associated with columns attr1, attr2, attr3
31 assembler = VectorAssembler(
32     inputCols=["attr1", "attr2", "attr3"],
33     outputCol="features"
34 )
35
36 ## Create a LogisticRegression object.
37 ## LogisticRegression is an Estimator that is used to
38 ## create a classification model based on logistic regression.
39 lr = LogisticRegression()
40
41 ## Set the values of the parameters of the
42 ## Logistic Regression algorithm using the setter methods.
43 ## There is one set method for each parameter
44 ## For example, we are setting the number of maximum iterations to 10
45 ## and the regularization parameter to 0.01
46 lr.setMaxIter(10)
47 lr.setRegParam(0.01)
48
49 ## Define a pipeline that is used to create the logistic regression
50 ## model on the training data. The pipeline includes also
51 ## the preprocessing step
52 pipeline = Pipeline().setStages([assembler, lr])

```

- ① `randomSplit` can be used to split the content of an input DataFrame in subsets

20.5 Hyperparameter tuning

The setting of the parameters of an algorithm is always a difficult task. A brute force approach can be used to find the setting optimizing a quality index, by splitting the training data in two subsets:

- The first set is used to build a model
- The second one is used to evaluate the quality of the model

The setting that maximizes a quality index (e.g., the prediction accuracy) is used to build the final model on the whole training dataset.

Using one single split of the training set usually leads to biased results, so the cross-validation approach is normally used

- Create k splits and k models
- The parameter setting that achieves, on the average, the best result on the k models is selected as final setting of the algorithm parameters

Spark supports a brute-force grid-based approach to evaluate a set of possible parameter settings on a pipeline

- Input
 - An MLlib pipeline
 - A set of values to be evaluated for each input parameter of the pipeline: all the possible combinations of the specified parameter values are considered and the related models are automatically generated and evaluated by Spark
 - A quality evaluation metric to evaluate the result of the input pipeline
- Output: the model associated with the best parameter setting, in term of quality evaluation metric

Example

This example shows how a grid-based approach can be used to tune a logistic regression classifier on a structured dataset: the pipeline that is repeated multiple times is based on the cross validation component. The input data set is the same structured dataset used for the example of the evaluators. The following parameters of the logistic regression algorithm are considered in the brute-force search/parameter tuning

- Maximum iteration: [10, 100, 1000]
- Regulation parameter: [0.1, 0.01]

In total, 6 parameter configurations are evaluated ($3 * 2$).

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.classification import LogisticRegression
4  from pyspark.ml.evaluation import MulticlassClassificationEvaluator
5  from pyspark.ml.evaluation import BinaryClassificationEvaluator
6  from pyspark.ml.tuning import ParamGridBuilder
7  from pyspark.ml.tuning import CrossValidator
8  from pyspark.ml import Pipeline
9  from pyspark.ml import PipelineModel
10
11 ## input and output folders
12 labeledData = "ex_dataValidation/labeledData.csv"
13 unlabeledData = "ex_dataValidation/unlabeledData.csv"
14 outputPath = "predictionsLRPipelineTuning/"
15
16 ## Create a DataFrame from labeledData.csv
17 ## Training data in raw format
18 labeledDataDF = spark.read.load(
19     labeledData,
20     format="csv",
21     header=True,
22     inferSchema=True
23 )
24
25 #### ****
26 ## Training step
27 ## ****
28 ## Define an assembler to create a column (features) of type Vector
29 ## containing the double values associated with columns attr1, attr2, attr3
30 assembler = VectorAssembler(
31     inputCols=["attr1", "attr2", "attr3"],
32     outputCol="features"
33 )
34
35 ## Create a LogisticRegression object.
36 ## LogisticRegression is an Estimator that is used to
37 ## create a classification model based on logistic regression.
38 lr = LogisticRegression()
39
40 ## Define a pipeline that is used to create the logistic regression
41 ## model on the training data. The pipeline includes also the preprocessing step
42 pipeline = Pipeline().setStages([assembler, lr])
43
44 ## We use a ParamGridBuilder to construct a grid of parameter values to
45 ## search over.
46 ## We set 3 values for lr.setMaxIter and 2 values for lr.regParam.
47 ## This grid will evaluate 3 x 2 = 6 parameter settings for
48 ## the input pipeline.
49 paramGrid = ParamGridBuilder() \
50     .addGrid(lr.maxIter, [10,100,1000]) \
51     .addGrid(lr.regParam, [0.1,0.01]) \
52     .build()

```

- ① There is one call to the addGrid method for each parameter that we want to set: each call to the addGrid method is characterized by the parameter we want to consider, and the list of values to test/to consider.
- ② Here the characteristics of the cross validation are set: the pipeline to be evaluated, the set of parameters to be considered, the evaluator (i.e., the object that is used to evaluate the quality of the model), and the number of folds to consider (i.e., the number of repetitions).
- ③ The returned model is the one associated with the best parameter setting, based on the result of the cross-validation test

20.6 Sparse labeled data

Frequently the training data are sparse (e.g., textual data are sparse: each document contains only a subset of the possible words), so sparse vectors are frequently used. MLlib supports reading training examples stored in the LIBSVM format: this is a commonly used textual format that is used to represent sparse documents/data points.

The LIBSVM format is a textual format in which each line represents an input record/data point by using a sparse feature vector: each line has the format

```
label index1:value1 index2:value2 ...
```

where

- **label** is an integer associated with the class label. It is the first value of each line.
- **index#** are integer values representing the features
- **value#** are the (double) values of the features

Consider the following two records/data points characterized by 4 predictive features and a class label

Features = [5.8, 1.7, 0, 0]	Label = 1
Features = [4.1, 0, 2.5, 1.2]	Label = 0

Their LIBSVM format-based representation is the following

```
1 1:5.8 2:1.7
0 1:4.1 3:2.5 4:1.2
```

LIBSVM files can be loaded into DataFrames by combining the following methods

- `read()`
- `format("libsvm")`
- `load(inputpath)`

The returned DataFrame has two columns:

- **label**: the double value associated with the label
- **features**: the sparse vector associated with the predictive features

 Example

```
1 spark.read.format("libsvm").load("sample_libsvm_data.txt")
```

21 Clustering algorithms

Spark MLlib provides a (limited) set of clustering algorithms

- K-means
- Bisecting k-means
- Gaussian Mixture Model (GMM)

Each clustering algorithm has its own parameters, however all the provided algorithms identify a set of groups of objects/clusters and assign each input object to one single cluster. All the clustering algorithms available in Spark work only with numerical data: categorical values must be mapped to integer values (i.e., numerical values).

The input of the MLlib clustering algorithms is a DataFrame containing a column called features of type Vector. The clustering algorithm clusters the input records by considering only the content of features (the other columns, if any, are not considered).

:::{.callout-note collapse="true"} ### Example The goal is to group customers in groups based on their characteristics.

Consider the following input data: a set of customer profiles.

MonthlyIncome	NumChildren
1400.0	2
11105.5	0
2150.0	2

The following input DataFrame that must be generated as input for the MLlib clustering algorithms

features
1400.0, 2.0
11105.5, 0.0
2150.0, 2.0

The values of all input attributes are stored in a vector of doubles (one vector for each input record). The generated DataFrame contains a column called features containing the vectors associated with the input records.

21.1 Main steps

The steps for clustering with Mllib are the following

1. Create a DataFrame with the features column.
2. Define the clustering pipeline and run the `.fit()` method on the input data to infer the clustering model (e.g., the centroids of the k-means algorithm). This step returns a clustering model.
3. Invoke the `.transform()` method of the inferred clustering model on the input data to assign each input record to a cluster. This step returns a new DataFrame with the new column “prediction” in which the cluster identifier is stored for each input record.

21.2 K-means clustering algorithm

K-means is one of the most popular clustering algorithms, characterized by one important parameter: the number of clusters K (the choice of K is a complex operation). Notice that this method is able to identify only spherical shaped clusters.

Example

The following paragraphs show how to apply the K-means algorithm provided by MLlib. The input dataset is a structured dataset with a fixed number of attributes, and all the attributes are numerical attributes.

Example of input file

```
attr1,attr2,attr3
0.5,0.9,1.0
0.6,0.6,0.7
```

In this example code it is assumed that the input data is already normalized (i.e., all values are already in the range $[0, 1]$). Scalers/Normalizers can be used to normalize data if it is needed.

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.clustering import KMeans
4  from pyspark.ml import Pipeline
5  from pyspark.ml import PipelineModel
6
7  ## input and output folders
8  inputData = "ex_datakmeans/dataClusteering.csv"
9  outputPath = "clusterskmeans/"
10
11 ## Create a DataFrame from dataClusteering.csv
12 ## Training data in raw format
13 inputDataDF = spark.read.load(
14     inputData,format="csv",
15     header=True,
16     inferSchema=True
17 )
18
19 ## Define an assembler to create a column (features) of type Vector
20 ## containing the double values associated with columns attr1, attr2, attr3
21 assembler = VectorAssembler(
22     inputCols=["attr1", "attr2", "attr3"],
23     outputCol="features"
24 )
25
26 ## Create a k-means object.
27 ## k-means is an Estimator that is used to
28 ## create a k-means algorithm
29 km = KMeans()
30
31 ## Set the value of k ( = number of clusters)
32 km.setK(2)
33
34 ## Define the pipeline that is used to cluster
35 ## the input data
36 pipeline = Pipeline().setStages([assembler, km])
37
38 ## Execute the pipeline on the data to build the
39 ## clustering model
40 kmeansModel = pipeline.fit(inputDataDF)
41
42 ## Now the clustering model can be applied on the input data
43 ## to assign them to a cluster (i.e., assign a cluster id)
44 ## The returned DataFrame has the following schema (attributes)
45 ## - features: vector (values of the attributes)
46 ## - prediction: double (the predicted cluster id)
47 ## - original attributes attr1, attr2, attr3
48 clusteredDataDF = kmeansModel.transform(inputDataDF)
49
50 ## Select only the original columns and the clusterID (prediction) one
51 ## I rename prediction to clusterID
52 clusteredData = clusteredDataDF \

```

- ① The returned DataFrame has a new column called “prediction” in which the predicted cluster identifier (an integer) is stored for each input record.

22 Regression algorithms

Spark MLlib provides a set of regression algorithms

- Linear regression
- Decision tree regression
- Random forest regression
- Survival regression
- Isotonic regression

A regression algorithm is used to predict the value of a continuous attribute (the target attribute) by applying a model on the predictive attributes. The model is trained on a set of training data (i.e., a set of data for which the value of the target attribute is known), and it is applied on new data to predict the target attribute.

The regression algorithms available in Spark work only on numerical data. They work similarly to classification algorithms, but they predict continuous numerical values (the target attribute is a continuous numerical attribute).

The input data must be transformed in a DataFrame having the following attributes:

- label: the continuous numerical double value to be predicted
- features: the double vector with the predictive features.

The main steps used to infer a regression model with MLlib are the same we use to infer a classification model, and the difference is only given by the type of the target attribute to predict.

22.1 Linear regression

Linear regression (LR) is a popular, effective and efficient regression algorithm. The following paragraphs show how to instantiate a linear regression algorithm in Spark and apply it on new data.

LR with structured data

The input dataset is a structured dataset with a fixed number of attributes

- One attribute is the target attribute (the label): it is assumed that the first column contains the target attribute;
- The others are predictive attributes that are used to predict the value of the target attribute.

 Example

Consider the following example file

```
label,attr1,attr2,attr3  
2.0,0.0,1.1,0.1  
5.0,2.0,1.0,-1.0  
5.0,2.0,1.3,1.0  
2.0,0.0,1.2,-0.5
```

Each record has three predictive attributes and the target attribute

- The first attribute (“label”) is the target attribute;
- The other attributes (“attr1”, “attr2”, “attr3”) are the predictive attributes.

```

1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.regression import LinearRegression
4  from pyspark.ml import Pipeline
5  from pyspark.ml import PipelineModel
6
7  ## input and output folders
8  trainingData = "ex_dataregression/trainingData.csv"
9  unlabeledData = "ex_dataregression/unlabeledData.csv"
10 outputPath = "predictionsLinearRegressionPipeline/"
11
12 #### ****
13 ## Training step
14 #### ****
15 ## Create a DataFrame from trainingData.csv
16 ## Training data in raw format
17 trainingData = spark.read.load(
18     trainingData,
19     format="csv",
20     header=True,
21     inferSchema=True
22 )
23
24 ## Define an assembler to create a column (features) of type Vector
25 ## containing the double values associated with columns attr1, attr2, attr3
26 assembler = VectorAssembler(
27     inputCols=["attr1", "attr2", "attr3"],
28     outputCol="features"
29 )
30
31 ## Create a LinearRegression object.
32 ## LinearRegression is an Estimator that is used to
33 ## create a regression model based on linear regression
34 lr = LinearRegression()
35
36 ## Set the values of the parameters of the
37 ## Linear Regression algorithm using the setter methods.
38 ## There is one set method for each parameter
39 ## For example, we are setting the number of maximum iterations to 10
40 ## and the regularization parameter to 0.01
41 lr.setMaxIter(10)
42 lr.setRegParam(0.01)
43
44 ## Define a pipeline that is used to create the linear regression
45 ## model on the training data. The pipeline includes also
46 ## the preprocessing step
47 pipeline = Pipeline().setStages([assembler, lr])
48
49 ## Execute the pipeline on the training data to build the
50 ## regression model
51 regressionModel = pipeline.fit(trainingData)
52 ## Now, the regression model can be used to predict the target attribute value

```

LR with textual data

The linear regression algorithms can be used also when the input dataset is a collection of documents/texts. Also in this case the text must be mapped to a set of continuous attributes.

Parameter setting

The tuning approach that we used for the classification problem can also be used to optimize the regression problem, the only difference is given by the used evaluator: in this case the difference between the actual value and the predicted one must be computed.

23 Itemset and Association rule mining

Spark MLlib provides

- An itemset mining algorithm based on the FP-growth algorithm, that extracts all the sets of items (of any length) with a minimum frequency;
- A rule mining algorithm, that extracts the association rules with a minimum frequency and a minimum confidence; notice that only the rules with one single item in the consequent of the rules are extracted.

The input dataset in this case is a set of transactions, where each transaction is defined as a set of items

A transactional dataset example

ABCD
AB
BC
ADE

It contains 4 transactions, and the distinct items are A, B, C, D, E.

23.1 The FP-Growth algorithm and Association rule mining

FP-growth is one of the most popular and efficient itemset mining algorithms. It is characterized by one single parameter: the minimum support threshold (**minsup**), that is the minimum frequency of the itemset in the input transactional dataset; it can assume a real value in the range $(0, 1]$. The **minsup** threshold is used to limit the number of mined itemsets.

The input dataset is a transactional dataset.

Given a set of frequent itemsets, the frequent association rules can be mined. An association rule is mined if

- Its frequency is greater than the minimum support threshold **minsup** (i.e., a minimum frequency). The **minsup** value is specified during the itemset mining step and not during the association rule mining step.
- Its confidence is greater than the minimum confidence threshold **minconf** (i.e., a minimum correlation). It is a real value in the range $[0, 1]$.

The MLlib implementation of FP-growth is based on DataFrames, but differently from the other algorithms, the FP-growth algorithm is not invoked by using pipelines.

Steps for itemset and association rule mining in Spark

1. Instantiate an FP-Growth object
2. Invoke the `fit(input data)` method on the FP-Growth object
3. Retrieve the sets of frequent itemset and association rules by invoking the following methods of on the FP-Growth object
 - `freqItemsets()`
 - `associationRules()`

Input

The input of the MLlib itemset and rule mining algorithm is a DataFrame containing a column called `items`, whose data type is array of values. Each record of the input DataFrame contains one transaction (i.e., a set of items).

Example

Example of input data

```
transactions
ABCD
AB
BC
ADE
```

The column `items` must be created before invoking FP-growth

items
[A, B, C, D]
[A, B]
[B, C]
[A, D, E]

Each input line is stored in an array of strings. The generated DataFrame contains a column called `items`, which is an `ArrayType`, containing the lists of items associated with the input transactions.

Note

This example shows how to extract the set of frequent itemsets from a transactional dataset and the association rules from the extracted frequent itemsets.

The input dataset is a transactional dataset: each line of the input file contains a transaction (i.e., a set of items)

```
transactions
ABCD
AB
```

BC
ADE

```

1  from pyspark.ml.fpm import FPGrowth
2  from pyspark.ml import Pipeline
3  from pyspark.ml import PipelineModel
4  from pyspark.sql.functions import col, split
5
6  ## input and output folders
7  transactionsData = "ex_dataitemsets/transactions.csv"
8  outputPathItemsets = "Itemsets/"
9  outputPathRules = "Rules/"
10
11 ## Create a DataFrame from transactions.csv
12 transactionsDataDF = spark.read.load(
13     transactionsData,
14     format="csv",
15     header=True,
16     inferSchema=True
17 )
18
19 ## Transform Column transactions into an ArrayType
20 trsDataDF = transactionsDataDF \
21     .selectExpr('split(transactions, " ")') \
22     .withColumnRenamed("split(transactions, )", "items")      (1)
23
24 ## Transform Column transactions into an ArrayType
25 trsDataDF = transactionsDataDF \
26     .selectExpr('split(transactions, " ")') \
27     .withColumnRenamed("split(transactions, )", "items")
28
29 ## Create an FP-growth Estimator
30 fpGrowth = FPGrowth(
31     itemsCol="items",
32     minSupport=0.5,
33     minConfidence=0.6
34 )
35
36 ## Extract itemsets and rules
37 model = fpGrowth.fit(trsDataDF)
38
39 ## Retrieve the DataFrame associated with the frequent itemsets
40 dfItemsets = model.freqItemsets
41
42 ## Retrieve the DataFrame associated with the frequent rules
43 dfRules = model.associationRules
44
45 ## Save the result in an HDFS output folder
46 dfItemsets.write.json(outputPathItemsets)                      (2)
47
48 ## Save the result in an HDFS output folder
49 dfRules.write.json(outputPathRules)

```

- ① 'split(transactions, " ")' is the `pyspark.sql.functions.split()` function. It returns a `SQL ArrayType`.
- ② The result is stored in a JSON file because itemsets and rules are stored in columns associated with the data type `Array`. Hence, CSV files cannot be used to store the result.

24 Graph analytics in Spark

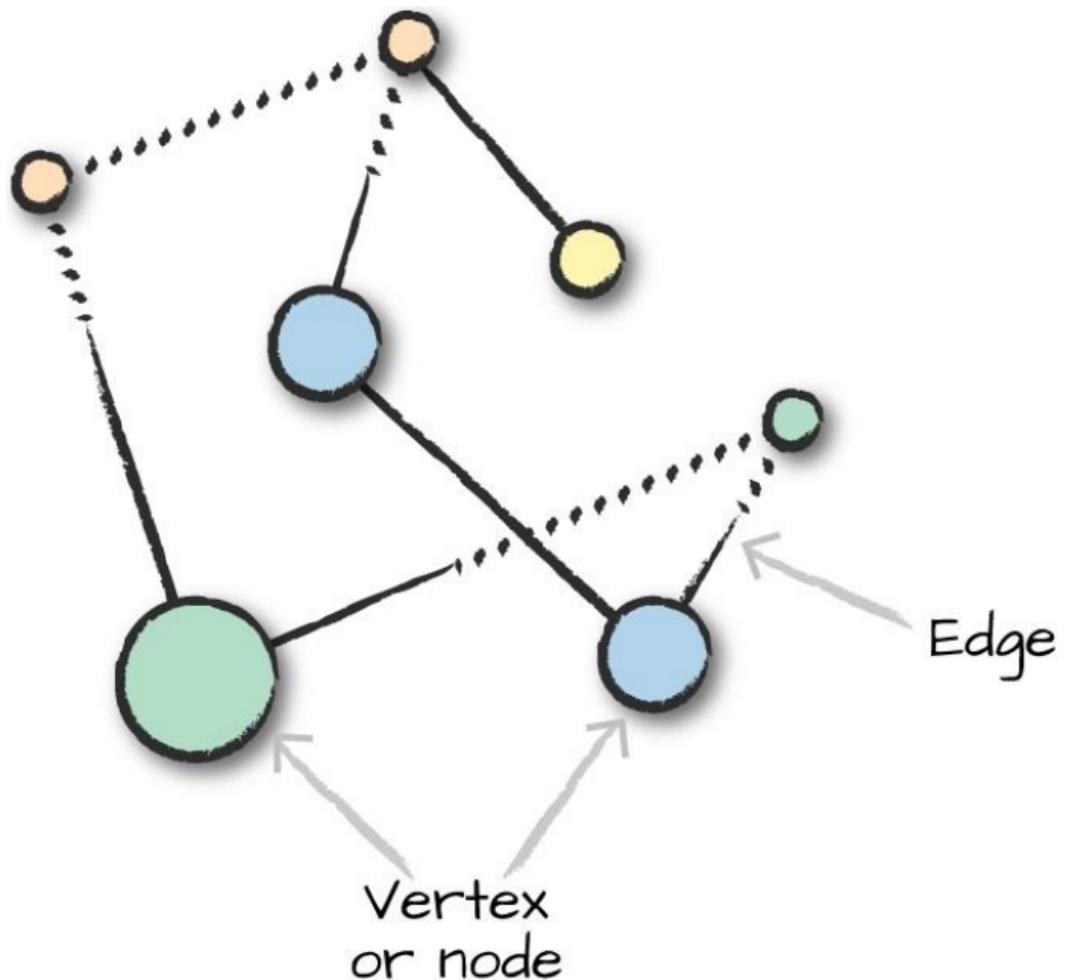
24.1 Introduction

Graphs are data structures composed of nodes and edges

- nodes/vertexes are denoted as $V = \{v_1, v_2, \dots, v_n\}$
- edges are denoted as $E = \{e_1, e_2, \dots, e_n\}$

Graph analytics is the process of analyzing relationships between vertexes and edges.

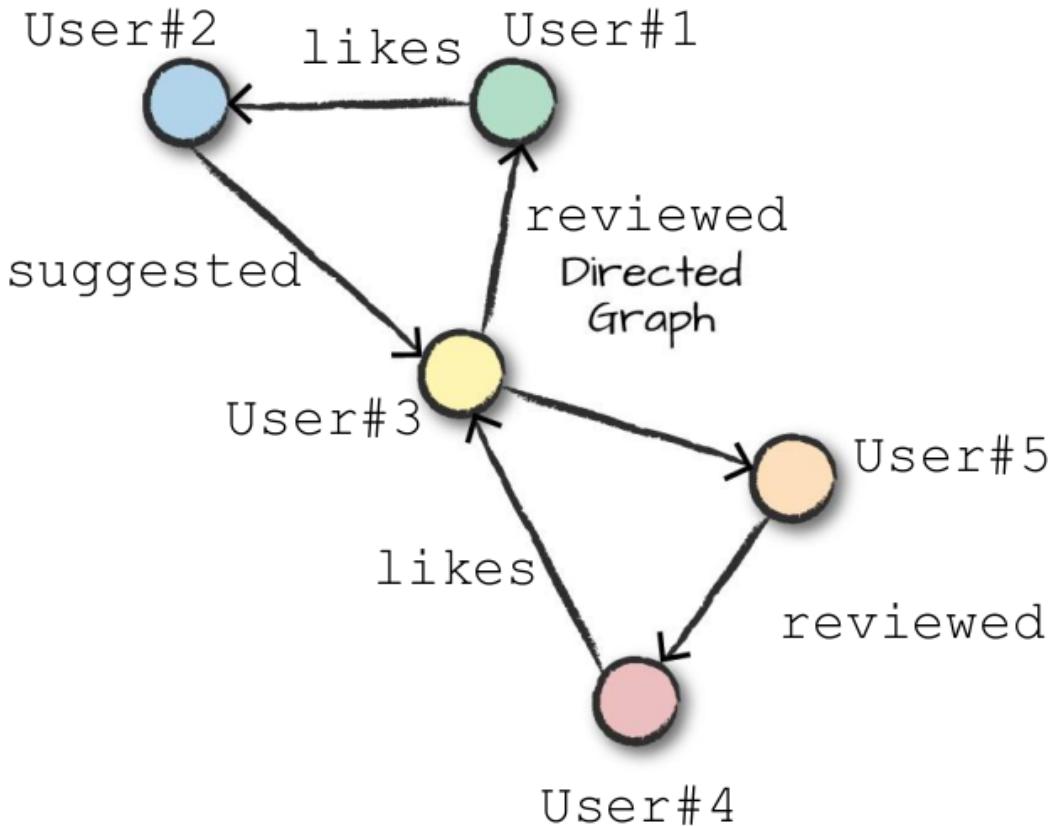
Figure 24.1: Example of graph



Graphs are called **undirected** if edges do not have a direction, otherwise they are called **directed** graphs. Vertexes and edges can have data associated with them

- weights are associated to edges (e.g., they may represent the strength of the relationship);
- labels are associated to vertexes (e.g., they may be the string associated with the name of the vertex).

Figure 24.2: Graph with labels and weights

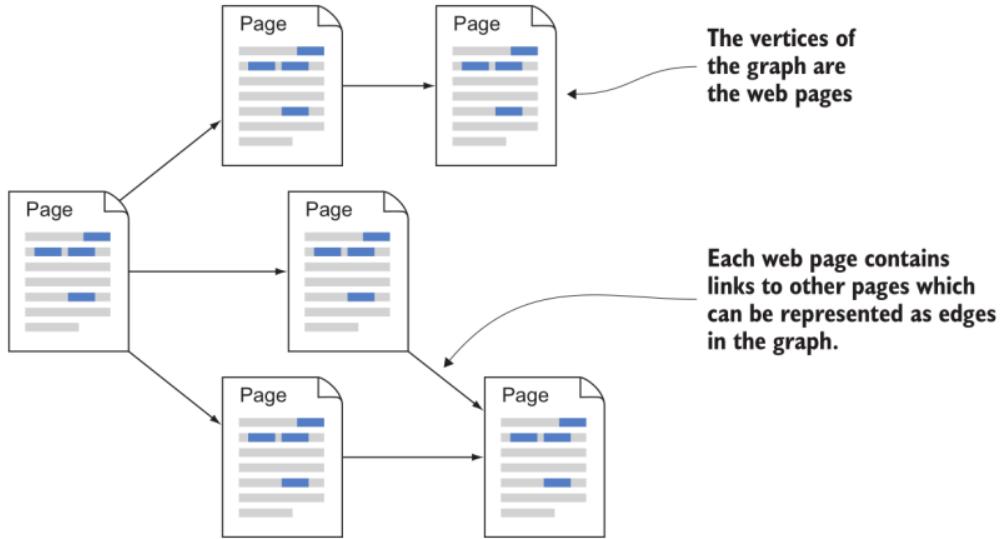


💡 Why graph analytics?

Graphs are natural way of describing relationships. Some practical example of analytics over graphs

- Ranking web pages (Google PageRank)

Figure 24.3: Pages in the web



- Detecting group of friends

Figure 24.4: Social networks

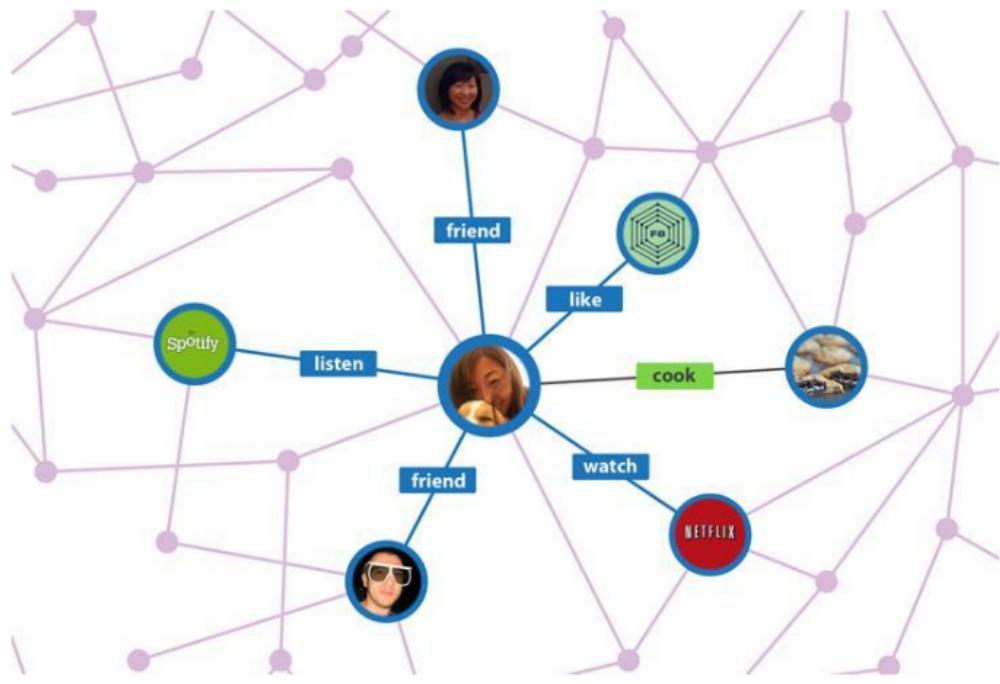
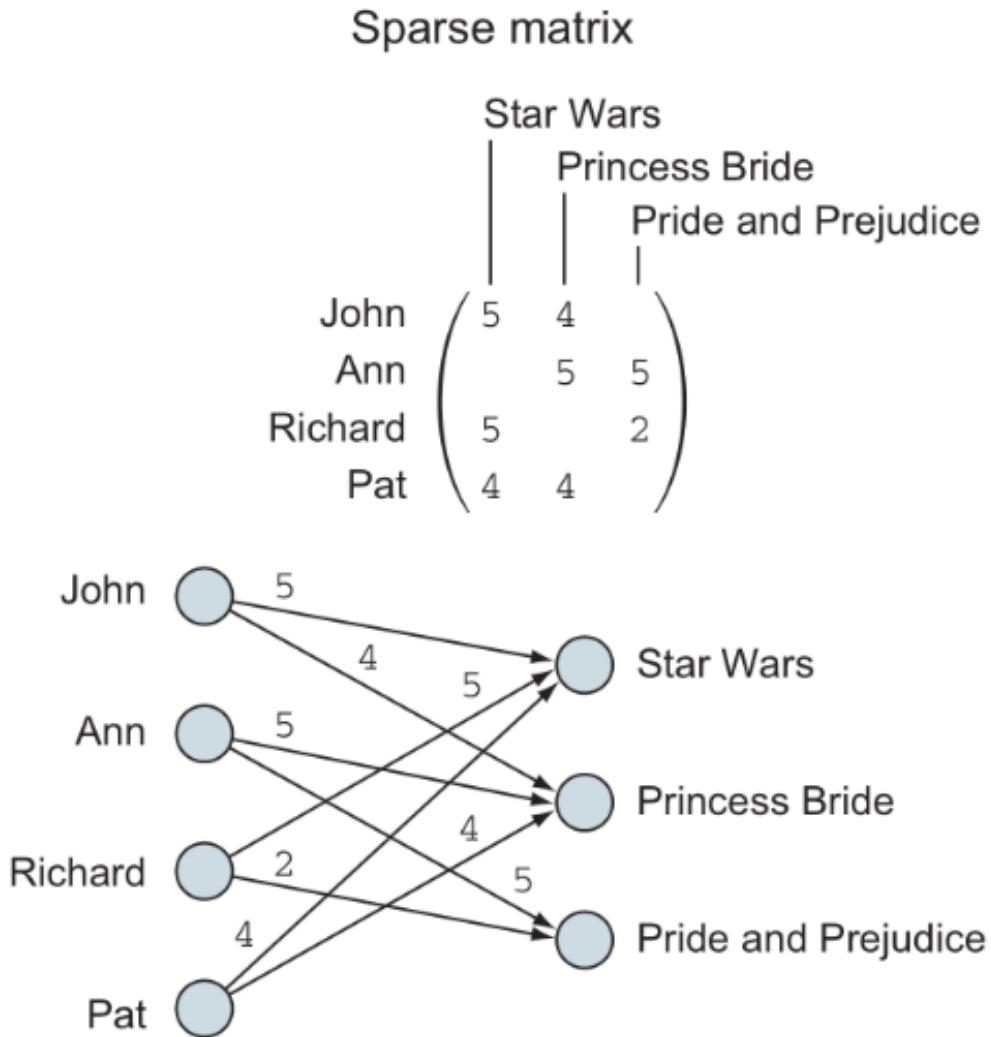


Figure 24.5: Movies watched by users

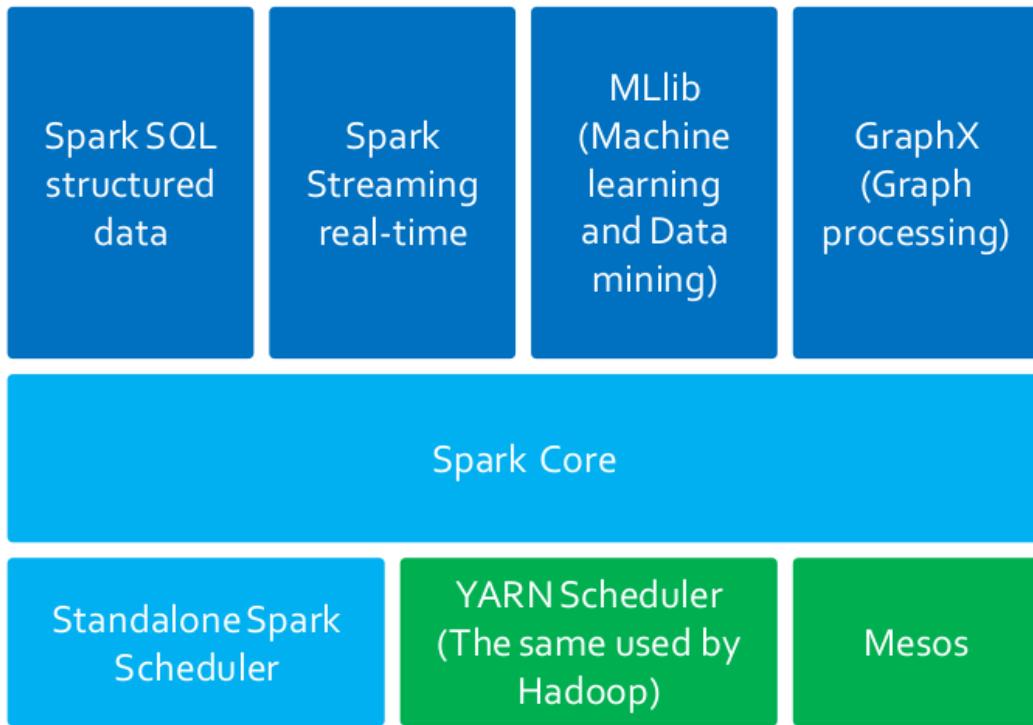


- Determine importance of infrastructure in electrical networks
- ...

24.2 Spark GraphX and GraphFrames

GraphX is the Spark RDD-based library for performing graph processing. It is a core part of Spark.

Figure 24.6: Spark core libraries

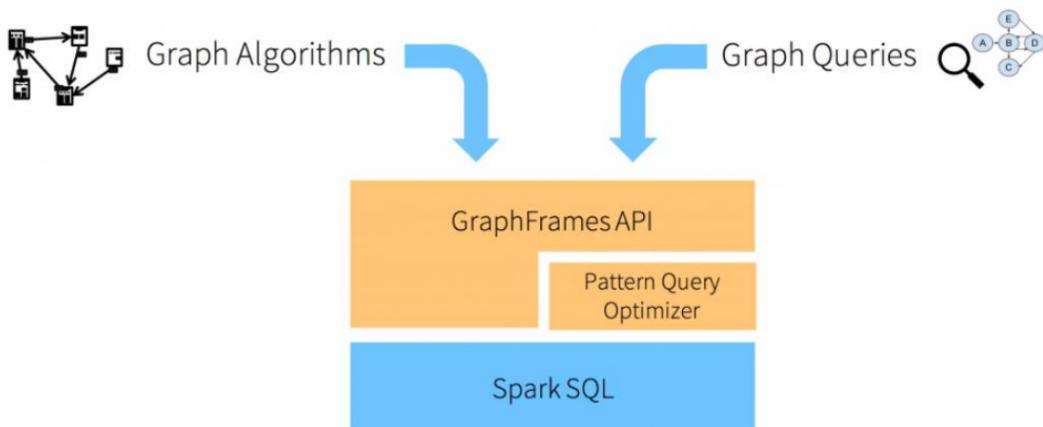


GraphX

- is low level interface with RDD
- is very powerful: many application and libraries built on top of it
- is not easy to use or optimize
- has no Python version of the APIs

[GraphFrames](#) is a library DataFrame-based for performing graph processing. It is a Spark external package built on top of GraphX.

Figure 24.7: GraphFrame structure



24.3 Building and querying graphs with GraphFrames

Building a Graph

Define vertexes and edges of the graph: vertexes and edges are represented by means of records inside DataFrames with specifically named columns

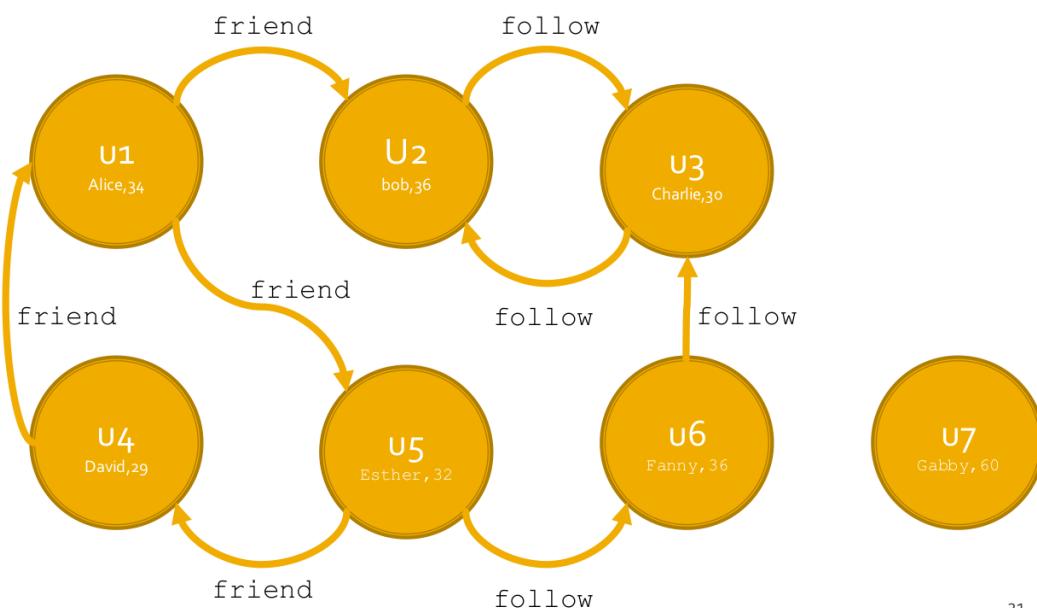
- One DataFrame for the definition of the vertexes of the graph. The DataFrames that are used to represent nodes/vertexes
 - Contain one record per vertex
 - Must contain a column named “id” that stores unique vertex IDs
 - Can contain other columns that are used to characterize vertexes
- One DataFrame for the definition of the edges of the graph. The DataFrames that are used to represent edges
 - Contain one record per edge
 - Must contain two columns “src” and “dst” storing source vertex IDs and destination vertex IDs of edges
 - Can contain other columns that are used to characterize edges

Create a graph of type `graphframes.graphframe.GraphFrame` by invoking the constructor `GraphFrame(v,e)`

- `v`: the DataFrame containing the definition of the vertexes
- `e`: the DataFrame containing the definition of the edges

Graphs in graphframes are directed graphs.

Figure 24.8: Building a graph example



 Example

Given this Vertex DataFrame

id	name	age
u1	Alice	34
u2	Bob	36
u3	Charlie	30
u4	David	29
u5	Esther	32
u6	Fanny	36
u7	Gabby	60

And this Edge DataFrame

src	dst	relationship
u1	u2	friend
u2	u3	follow
u3	u2	follow
u6	u3	follow
u5	u6	follow
u5	u4	friend
u4	u1	friend
u1	u5	friend

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)

```

Directed vs undirected edges

In undirected graphs the edges indicate a two-way relationship (each edge can be traversed in both directions). In GraphX it is possible to use `to_undirected()` to create an undirected copy of the Graph. Unfortunately GraphFrames does not support it, but it is possible to convert a graph by applying a `flatMap` function over the edges of the directed graph that creates symmetric edges and then create a new GraphFrame.

Cache graphs

As with RDD and DataFrame, it is possible to cache graphs in GraphFrame: it is convenient if the same (complex) graph result of (multiple) transformations is used multiple times in the same application. To do it, simply invoke `cache()` on the GraphFrame to cache, so that it persists the DataFrame-based representation of vertexes and edges of the graph.

Querying the graph

Some specific methods are provided to execute queries on graphs

- `filterVertices(condition)`
- `filterEdges(condition)`
- `dropIsolatedVertices()`

The returned result is the filtered version of the input graph.

`filterVertices(condition)`

`filterVertices(condition)` selects only the vertexes for which the specified condition is satisfied and returns a new graph with only the subset of selected vertexes.

`condition` contains an SQL-like condition on the values of the attributes of the vertexes (e.g., “age>35”).

`filterEdges(condition)`

`filterEdges(condition)` selects only the edges for which the specified condition is satisfied and returns a new graph with only the subset of selected edges.

`condition` contains an SQL-like condition on the values of the attributes of the edges (e.g., “relationship=‘friend’ ”).

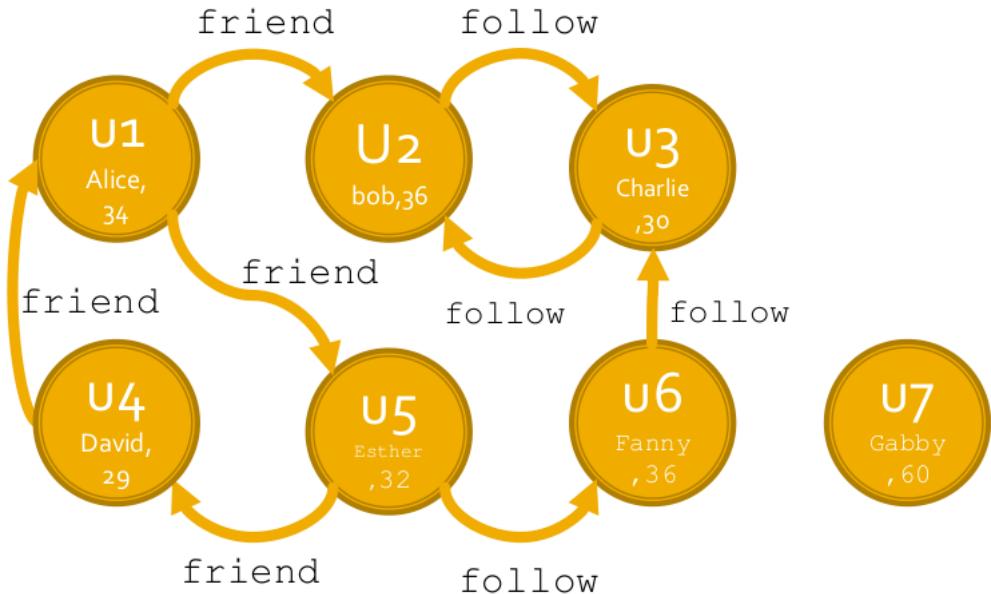
`dropIsolatedVertices()`

`dropIsolatedVertices()` drops the vertexes that are not connected with any other node and returns a new graph without the dropped nodes.

i Example

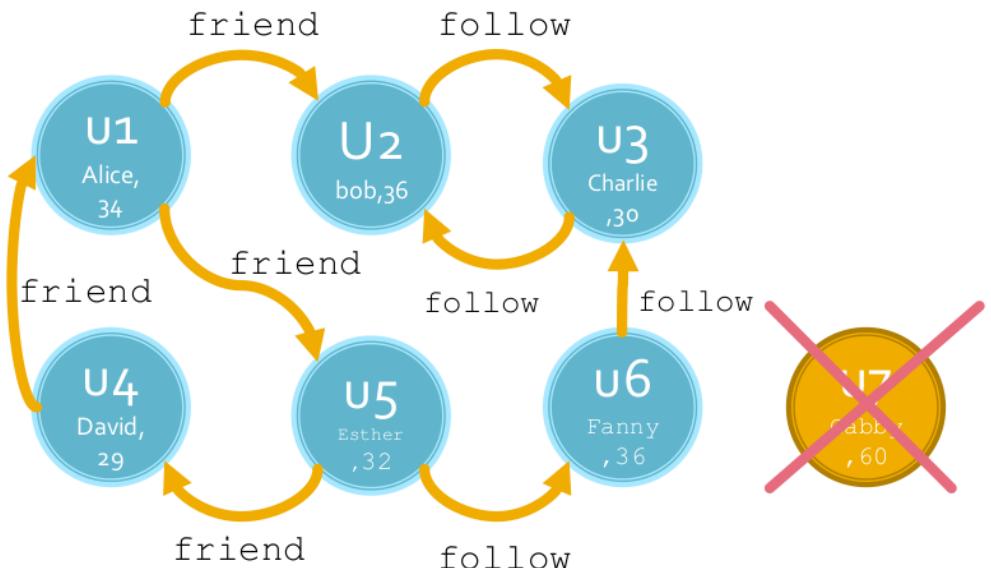
Given the input graph, create a new subgraph

Figure 24.9: Input graph



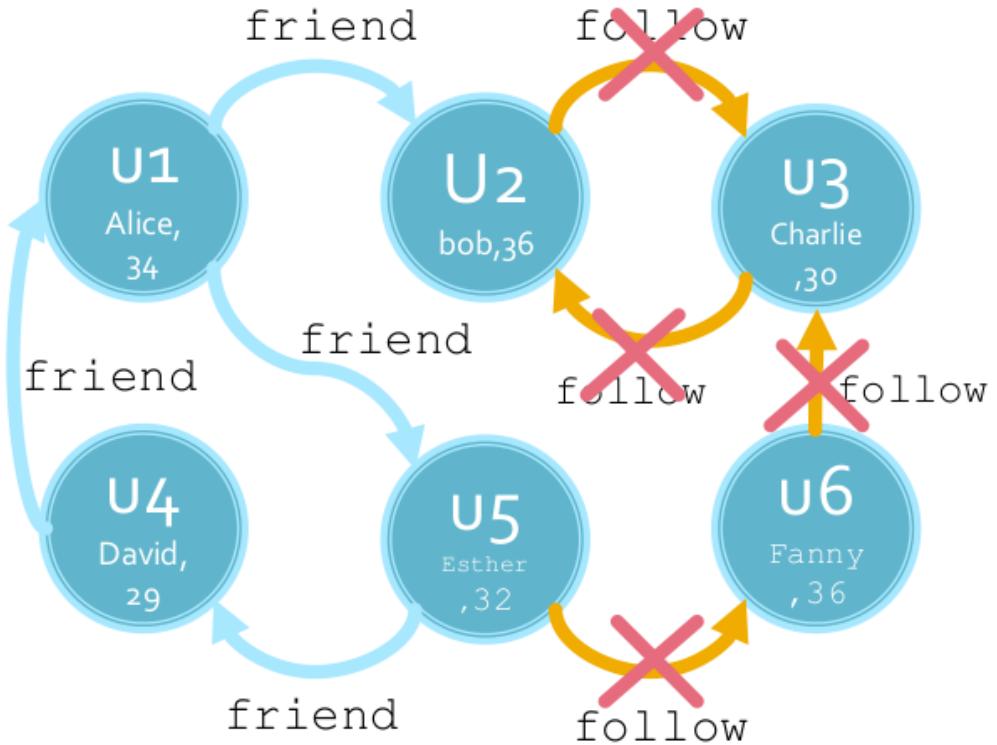
1. Include only the vertexes associated with users characterized by age between 29 and 50

Figure 24.10: Filter vertexes



2. Include only the edges representing the friend relationship

Figure 24.11: Filter edges



3. Drop isolated vertexes

Figure 24.12: Drop isolated vertices

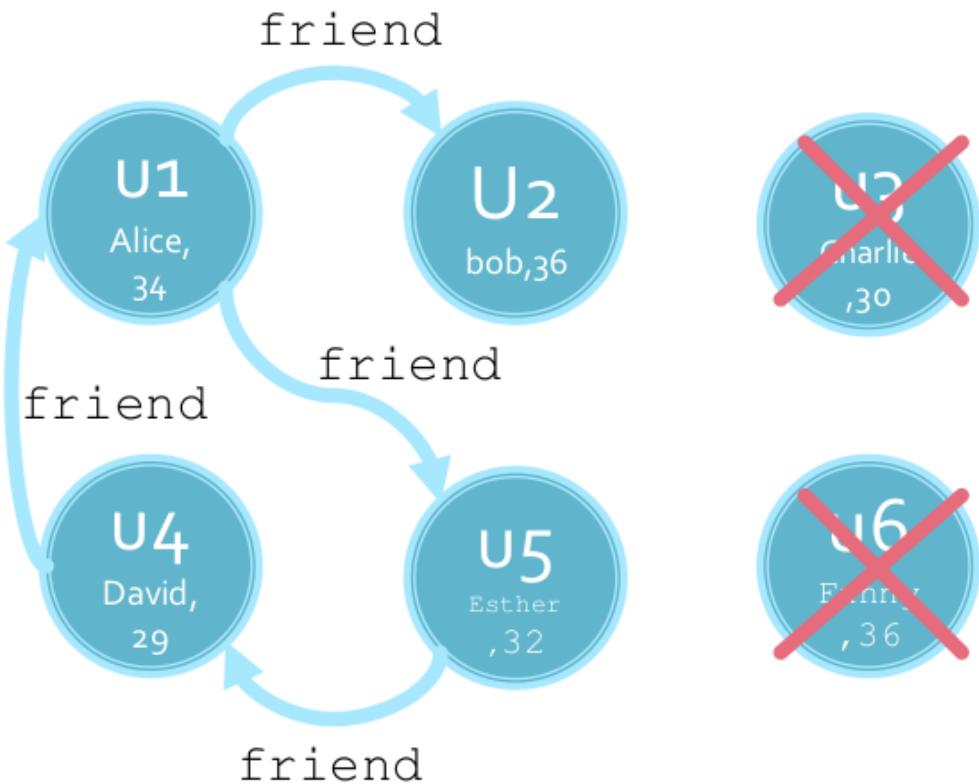
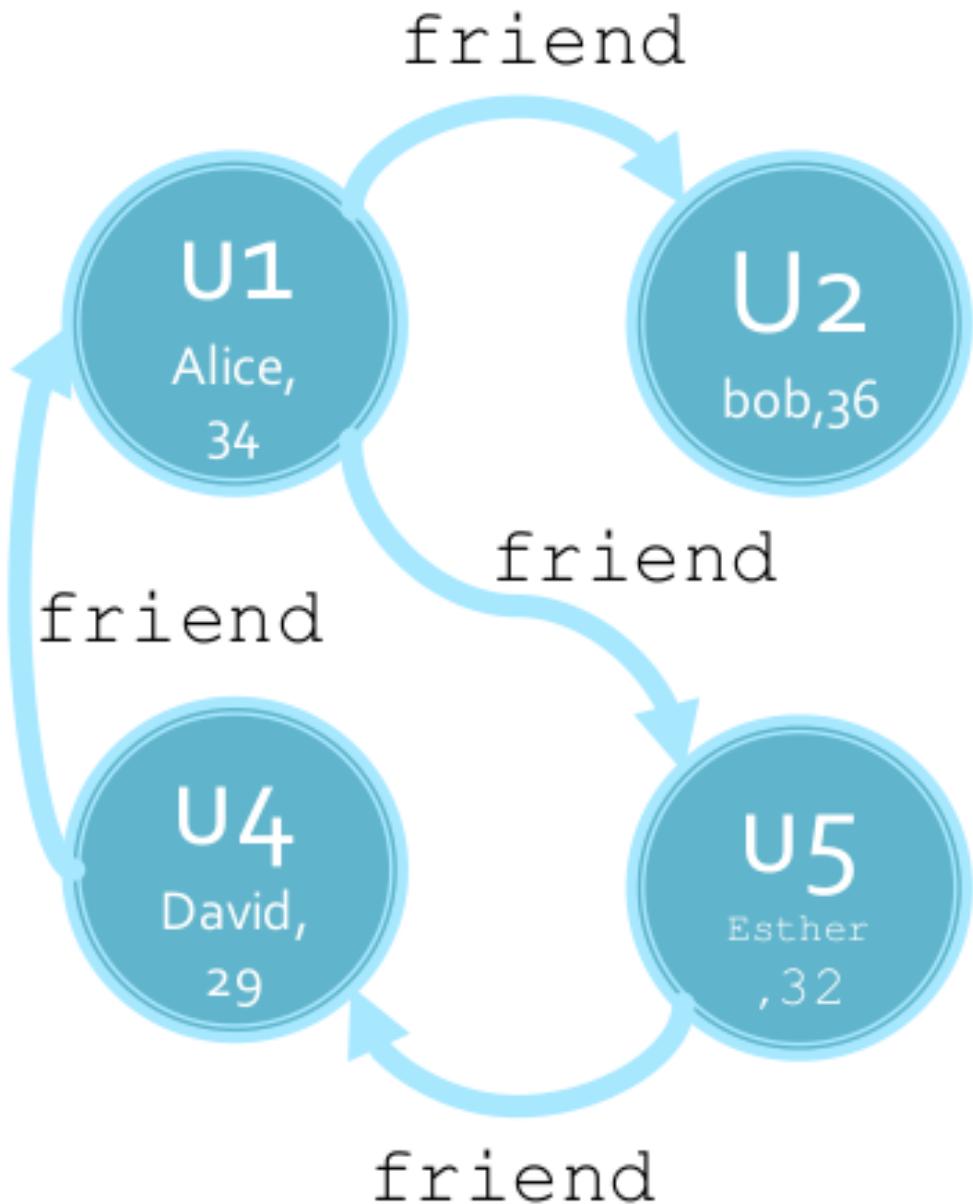


Figure 24.13: Output graph



```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 selectedUsersandFriendRelGraph = g \
36     .filterVertices("age>=29 AND age<=50") \
37     .filterEdges("relationship='friend'") \
38     .dropIsolatedVertices()

```

Given a GraphFrame, it is possible to access its vertexes and edges

- `g.vertices` returns the DataFrame associated with the vertexes of the input graph
- `g.edges` returns the DataFrame associated with the edges of the input graph

All the standard DataFrame transformations/actions are available also for the DataFrames that are used to store vertexes and edges. For example, the number of vertexes and the number of edges can be computed by invoking the `count()` action on the DataFrames `vertices` and `edges`, respectively.

 Example

Given the input graph

1. Count how many vertexes and edges has the graph
2. Find the smallest value of age (i.e., the age of the youngest user in the graph)
3. Count the number of edges of type “follow” in the graph

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Count how many vertexes and edges has the graph
36 print("Number of vertexes: ",g.vertices.count())
37 print("Number of edges: ",g.edges.count())
38
39 ## Print on the standard output the smallest value of age
40 ## (i.e., the age of the youngest user in the graph)
41 g.vertices.agg({"age":"min"}).show()
42
43 ## Print on the standard output
44 ## the number of "follow" edges in the graph.
45 numFollows = g.edges.filter("relationship = 'follow' ").count()
46
47 print(numFollows)

```

Motif finding

Motif finding refers to searching for structural patterns in graphs. A simple Domain-Specific Language (DSL) is used to specify the structure of the interesting patterns: the paths/subgraphs in the graph matching the specified structural pattern are selected.

DSL for Motif finding

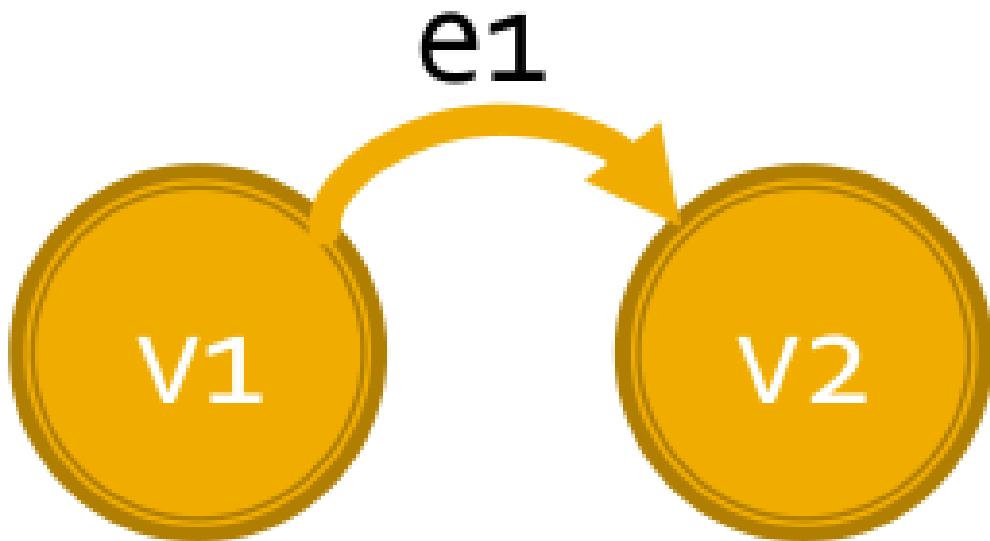
The basic unit of a pattern is a connection between vertexes

$$(v1)-[e1] \rightarrow (v2)$$

means: an arbitrary edge [e1] from an arbitrary vertex (v1) to another arbitrary vertex (v2)

- Edges are denoted by square brackets: [e1]
- Vertices are expressed by round brackets: (v1), (v2)

Figure 24.14: Basic unit

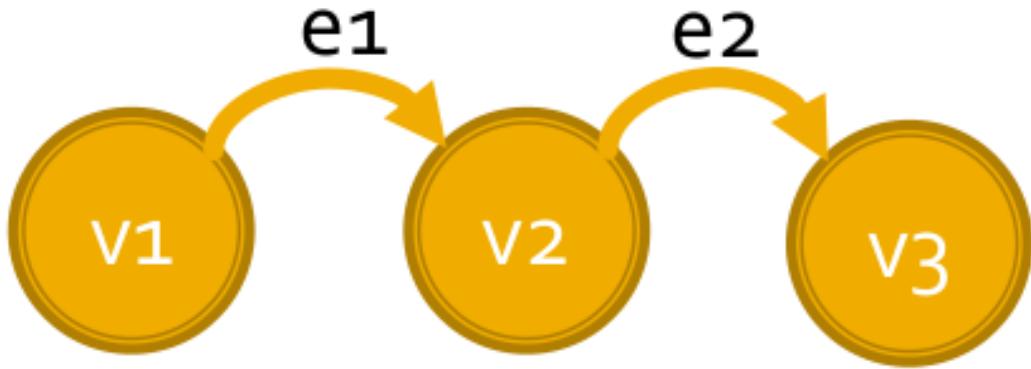


Patterns are chains of basic units

$$(v1)-[e1] \rightarrow (v2); \quad (v2)-[e2] \rightarrow (v3)$$

means: an arbitrary edge from an arbitrary vertex v_1 to another arbitrary vertex v_2 and another arbitrary edge from v_2 to another arbitrary vertex v_3 . Notice that v_3 and v_1 can be the same vertex.

Figure 24.15: Basic unit chaining

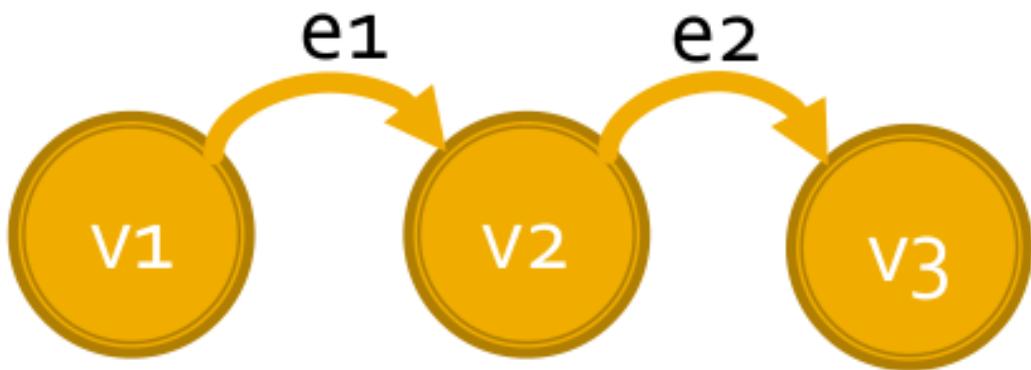


The same vertex name is used in a pattern to have a reference to the same vertex

$$(v1)-[e1] \rightarrow (v2); \quad (v2)-[e2] \rightarrow (v1)$$

means: an arbitrary edge from an arbitrary vertex $v1$ to another arbitrary vertex $v2$ and vice-versa.

Figure 24.16: Basic unit self-chaining

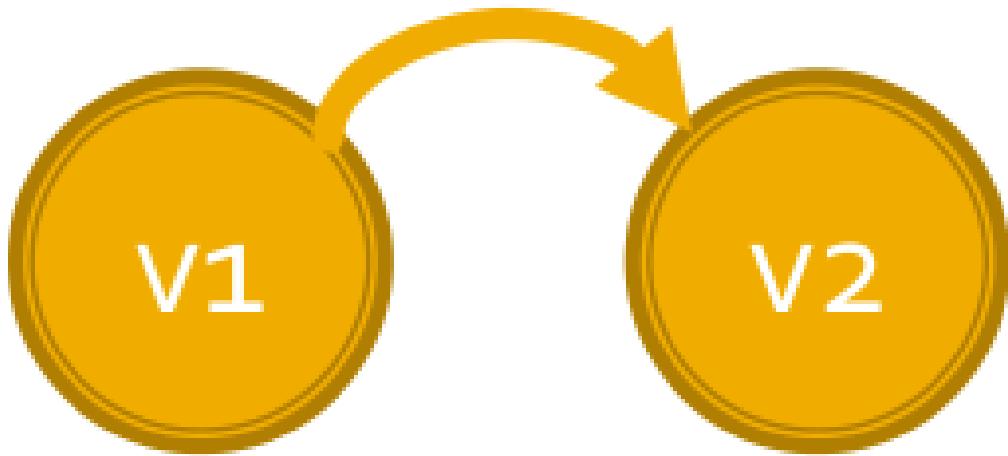


It is acceptable to omit names for vertices or edges in patterns when not needed

$$(v1) - [] \rightarrow (v2)$$

means: an arbitrary edge between two arbitrary vertexes $v1, v2$, but does not assign a name to the edge. These are called **anonymous** vertexes and edges.

Figure 24.17: Anonymous vertexes and edges

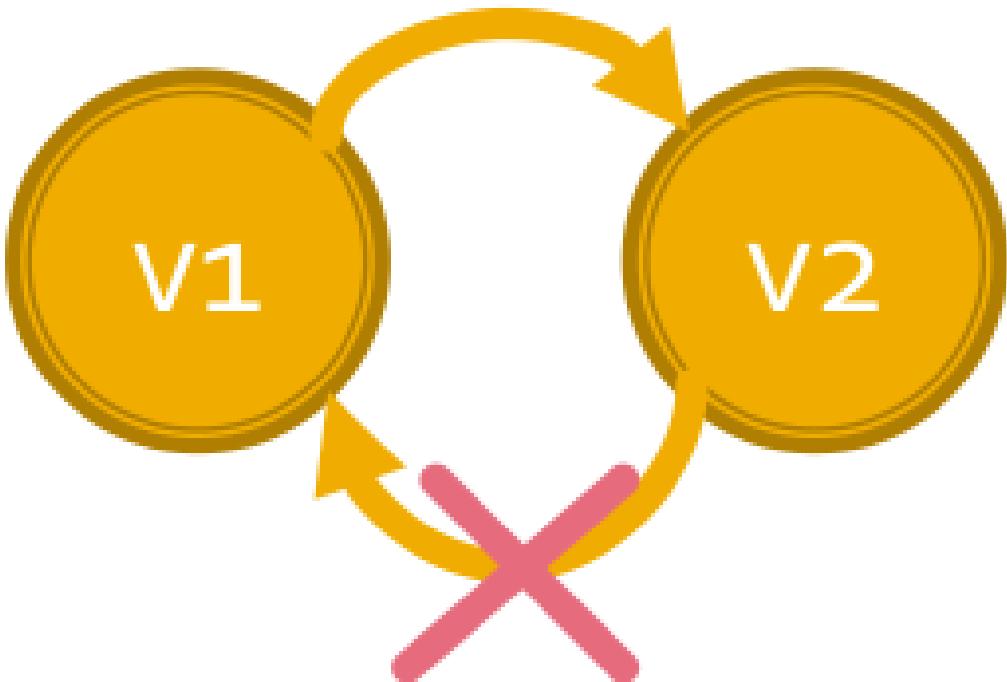


A basic unit (an edge between two vertexes) can be negated to indicate that the edge should not be present in the graph

$$(v1) - [] \rightarrow (v2); \quad !(v2) - [] \rightarrow (v1)$$

means: edges from $v1$ to $v2$ but no edges from $v2$ to $v1$.

Figure 24.18: Negating edges



The `find(motif)` method of GraphFrame is used to select motifs

- **motif** is a DSL representation of the structural pattern

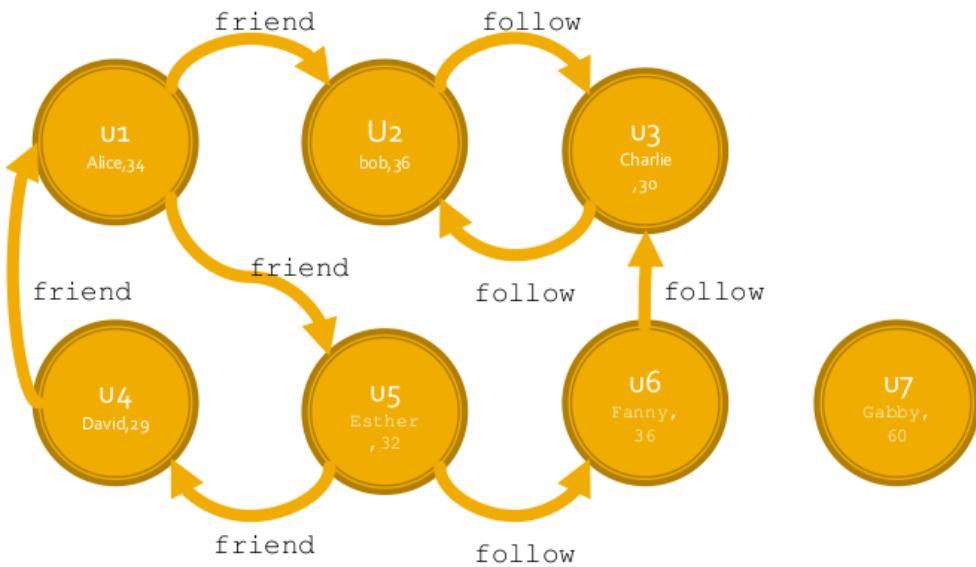
`find()` returns a DataFrame of all the paths matching the structural motif/pattern, one path per record. The returned DataFrame will have a column for each of the named elements (vertexes and edges) in the structural pattern/motif: Each column is a struct, and the fields of each struct are the labels/features of the associated vertex or edge. It can return duplicate rows/records, if there are many paths connecting the same nodes.

More complex queries on the structure and content of the patterns can be expressed by applying filters to the result DataFrame (i.e., more complex queries can be applied by combining `find()` and `filter()`).

i Example 1

Given the following graph

Figure 24.19: Example graph

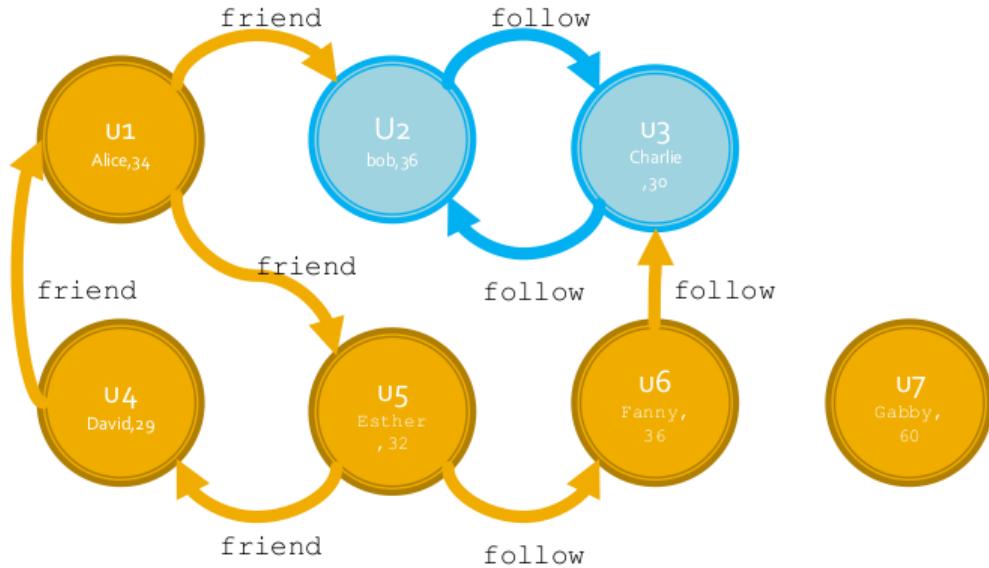


Find the paths/subgraphs matching the pattern

$$(v1)-[e1] \rightarrow (v2); \quad (v2)-[e2] \rightarrow (v1)$$

Store the result in a DataFrame

Figure 24.20: Result



Pay attention that two paths are returned:

- $u2 \rightarrow \text{follow} \rightarrow u3 \rightarrow \text{follow} \rightarrow u2$
- $u3 \rightarrow \text{follow} \rightarrow u2 \rightarrow \text{follow} \rightarrow u3$

The content of the returned Dataframe is the following

$v1$	$e1$	$v2$	$e2$
$[u2, \text{Bob}, 36]$	$[u2, u3, \text{follow}]$	$[u3, [\text{Charlie}], 30]$	$[u3, u2, \text{follow}]$
$[u3, [\text{Charlie}], 30]$	$[u3, u2, \text{follow}]$	$[u2, \text{Bob}, 36]$	$[u2, u3, \text{follow}]$

- There is one column for each (distinct) named vertex and edge of the structural pattern;
- The records are associated with the vertexes and edges of the selected paths;
- All columns are associated with the data type “struct”. Each struct has the same “schema/features” of the associated vertex or edge.

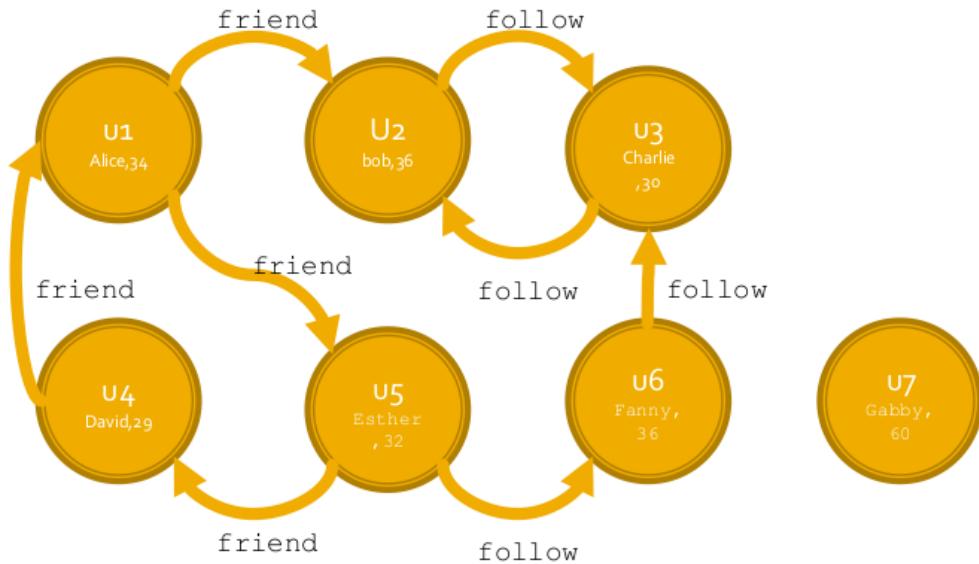
```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Retrieve the motifs associated with the pattern
36 ## vertex -> edge -> vertex -> edge ->vertex
37 motifs = g.find("(v1)-[e1]->(v2); (v2)-[e2]->(v1)")
```

i Example 2

Given the following graph

Figure 24.21: Example graph



Find the paths/subgraphs matching the pattern

$$(v1) - [\text{friend}] \rightarrow (v2); \quad (v2) - [\text{follow}] \rightarrow (v3)$$

Store the result in a DataFrame

Figure 24.22: First selected path

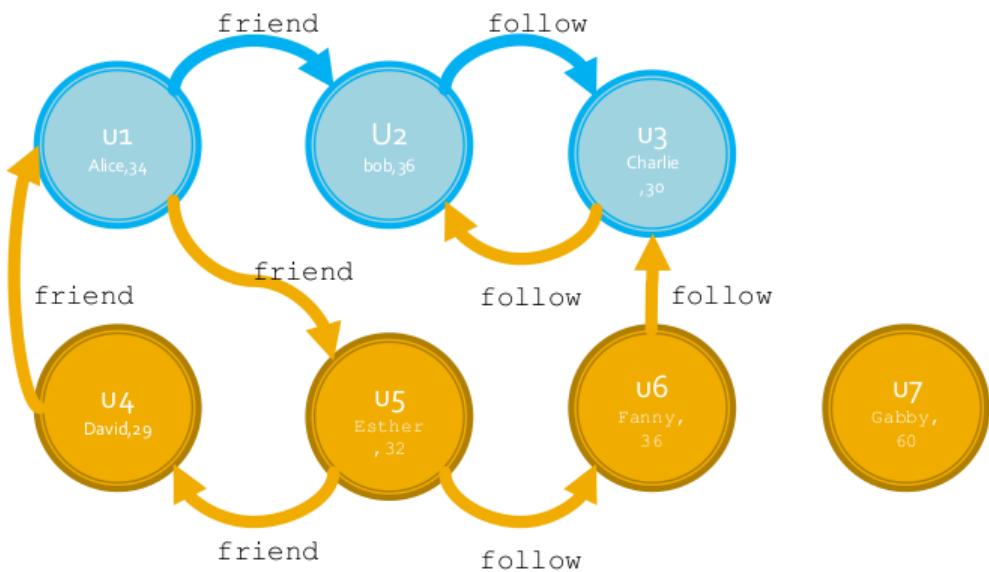
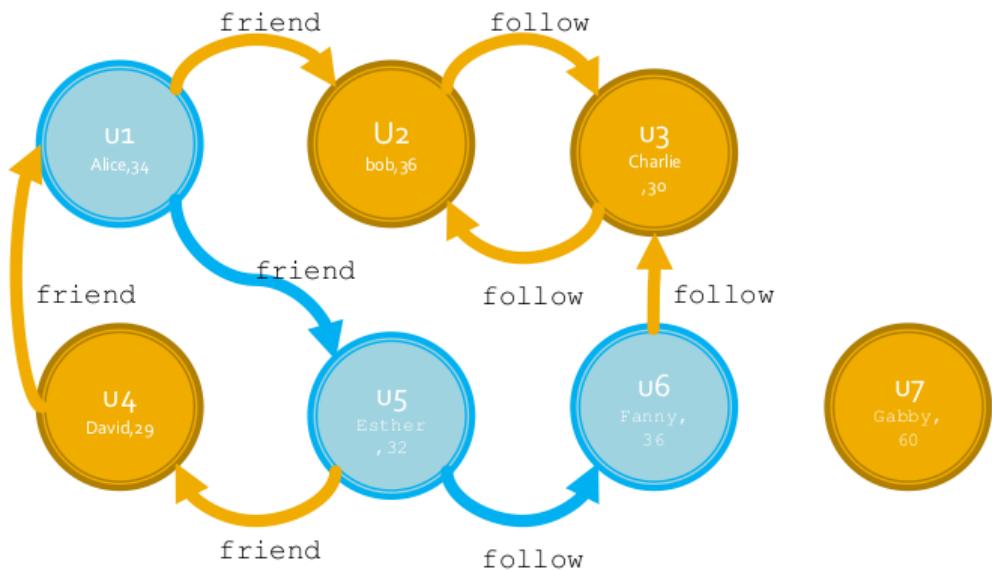


Figure 24.23: Second selected path



```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Retrieve the motifs associated with the pattern
36 ## vertex -> edge -> vertex -> edge -> vertex
37 motifs = g.find("(v1)-[friend]->(v2); (v2)-[follow]->(v3)")
38
39 ## Filter the motifs (the content of the motifs DataFrame)
40 ## Select only the ones matching the pattern
41 ## vertex -> friend-> vertex -> follow ->vertex
42 motifsFriendFollow = motifs \
43     .filter("friend.relationship='friend' AND follow.relationship='follow' ") ①

```

- ① Columns friend and follow are structs with three fields/attributes: “src”, “dst”, “relationship”. To access a field of a struct column use the syntax `columnName.field` (e.g.,

```
friend.relationship)
```

24.4 Basic statistics

Some specific properties are provided to compute basic statistics on the degrees of the vertexes

- `degrees`
- `inDegrees`
- `outDegrees`

The returned result of each of this property is a DataFrame with id and (in/out)Degree value.

`degrees`

`degrees` returns the degree of each vertex (i.e., the number of edges associated with each vertex). The result is stored in a DataFrame with Columns (vertex) “id” and “degree”, with one record per vertex. Only the vertexes with degree ≥ 1 are stored in the returned DataFrame.

`inDegrees`

`inDegrees` returns the in-degree of each vertex (i.e., the number of in-edges associated with each vertex). The result is stored in a DataFrame with Columns (vertex) “id” and “inDegree”, with one record per vertex. Only the vertexes with in-degree ≥ 1 are stored in the returned DataFrame.

`outDegrees`

`outDegrees` returns the out-degree of each vertex (i.e., the number of out-edges associated with each vertex). The result is stored in a DataFrame with Columns (vertex) “id” and “outDegree”, with one record per vertex. Only the vertexes with out-degree ≥ 1 are stored in the returned DataFrame.

i Example 1

Given the input graph, compute

- Degree of each vertex
- inDegree of each vertex
- outDegree of each vertex

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Retrieve the DataFrame with the information about the degree of
36 ## each vertex
37 vertexesDegreesDF = g.degrees
38
39 ## Retrieve the DataFrame with the information about the in-degree of
40 ## each vertex
41 vertexesInDegreesDF = g.inDegrees
42
43 ## Retrieve the DataFrame with the information about the out-degree of
44 ## each vertex
45 vertexesOutDegreesDF = g.outDegrees

```

Note

Given the input graph, select only the ids of the vertexes with at least 2 in-edges.

Figure 24.24: Example graph

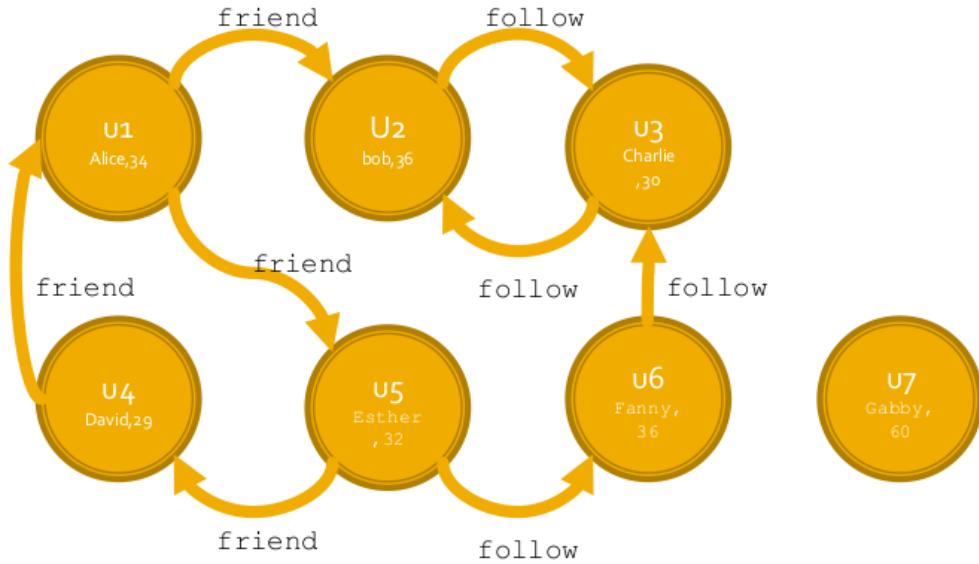


Figure 24.25: First selected vertex

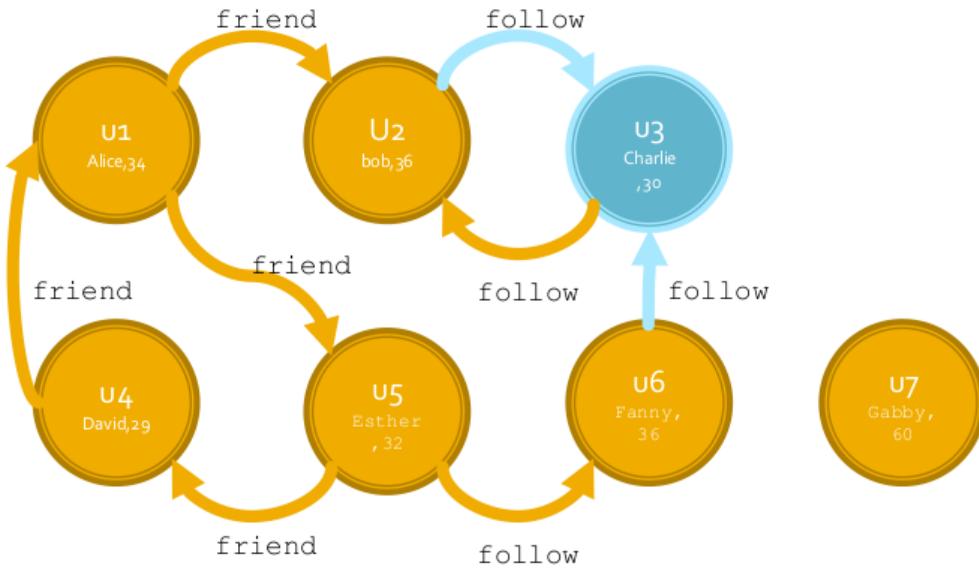


Figure 24.26: Second selected vertex

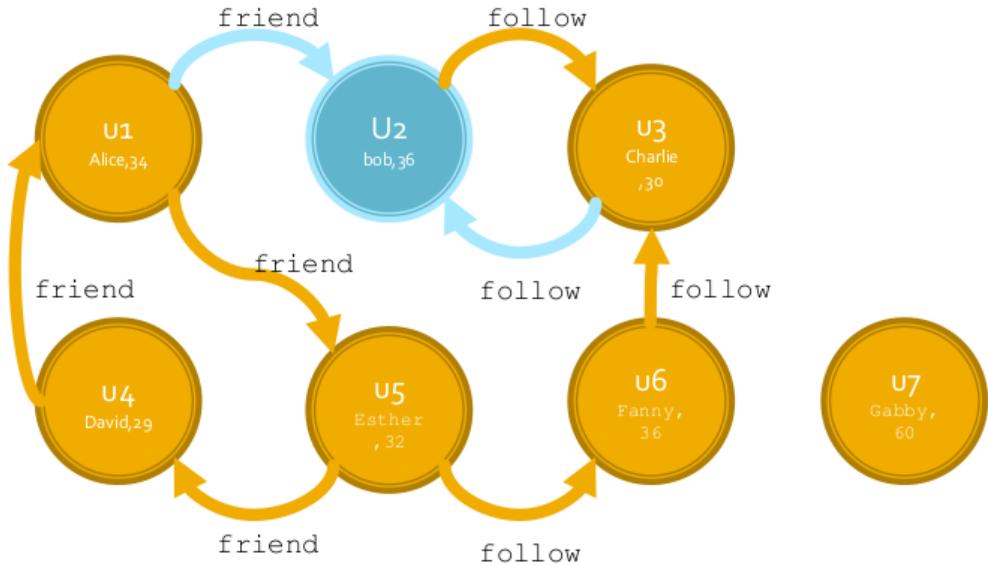
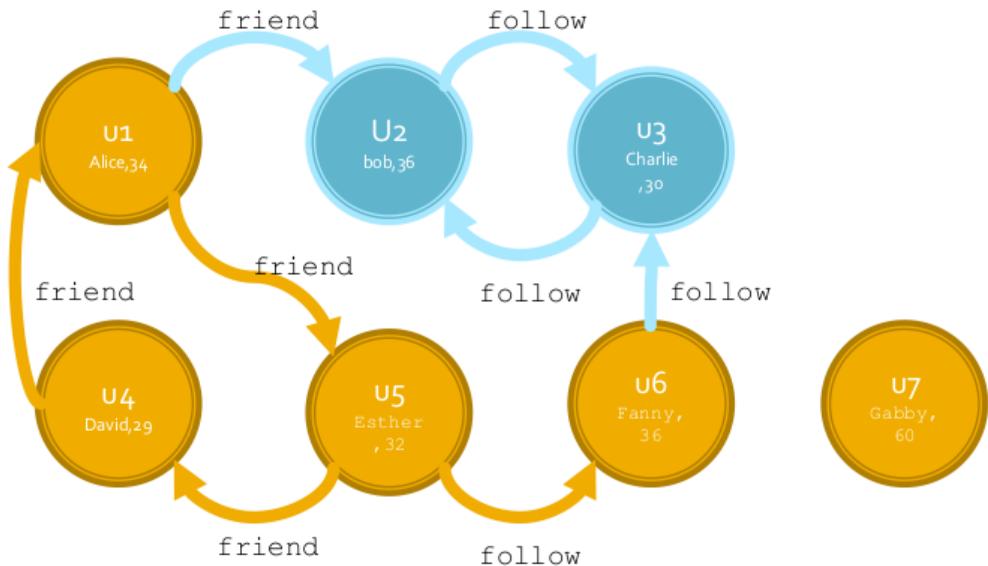


Figure 24.27: Resulting vertexes



The selected IDs are u_2 and u_3

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Retrieve the DataFrame with the information about the in-degree of
36 ## each vertex
37 vertexesInDegreesDF = g.inDegrees
38
39 ## Select only the vertexes with and in-degree value >=2
40 selectedVertexesDF = vertexesInDegreesDF.filter("inDegree>=2")
41
42 ## Select only the content of Column id
43 selectedVertexesIDsDF = selectedVertexesDF.select("id")

```

25 Graph Analytics in Spark

25.1 Algorithms over graphs

GraphFrame provides the parallel implementation of a set of state of the art algorithms for graph analytics

- Breadth first search
- Shortest paths
- Connected components
- Strongly connected component
- Label propagation
- PageRank
- ...

Also custom algorithms can be designed and implemented.

Checkpoint directory

To run some expensive algorithms, set a checkpoint directory that will store the state of the job at every iteration. This allows to continue where left off if the job crashes. Create such a folder to set the checkpoint directory with:

```
1 sc.setCheckpointDir(graphframes_ckpt_dir)
```

- `graphframes_ckpt_dir` is the new checkpoint folder directory
- `sc` is the `SparkContext` object (retrieve it from a `SparkSession` by using `spark.sparkContext`)

Breadth first search

Breadth-first search (BFS) is an algorithm for traversing/searching graph data structures: it finds the shortest path(s) from one vertex (or a set of vertexes) to another vertex (or a set of vertexes). It is used in many other algorithms

- Length of shortest paths
- Connected components
- ...

Implementation

```
1 bfs(fromExpr, toExpr, edgeFilter=None maxPathLength=10)
```

The `bfs()` method of the `GraphFrame` class returns the shortest path(s) from the vertexes matching expression `fromExpr` expression to vertexes matching expression `toExpr`. If there are many vertexes matching `fromExpr` and `toExpr`, only the couple(s) with the shortest length is returned.

- `fromExpr`: Spark SQL expression specifying valid starting vertexes for the execution of the BFS algorithm (e.g., to start from a specific vertex: “`id = [start vertex id]`”);
- `toExpr`: Spark SQL expression specifying valid target vertexes for the BFS algorithm;
- `maxPathLength`: Limit on the length of paths (default = 10);
- `edgeFilter`: Spark SQL expression specifying edges that may be used in the search (default None).

`bfs()` returns a DataFrame containing the selected shortest path(s). Notice that if multiple paths are valid and their length is equal to the shortest length, the returned DataFrame will contain one Row for each path. The number of columns of the returned DataFrame is equal to $(\text{length of the shortest path} * 2) + 1$.

Example 1

1. Find the shortest path from Esther to Charlie
2. Store the result in a DataFrame

Figure 25.1: Example graph

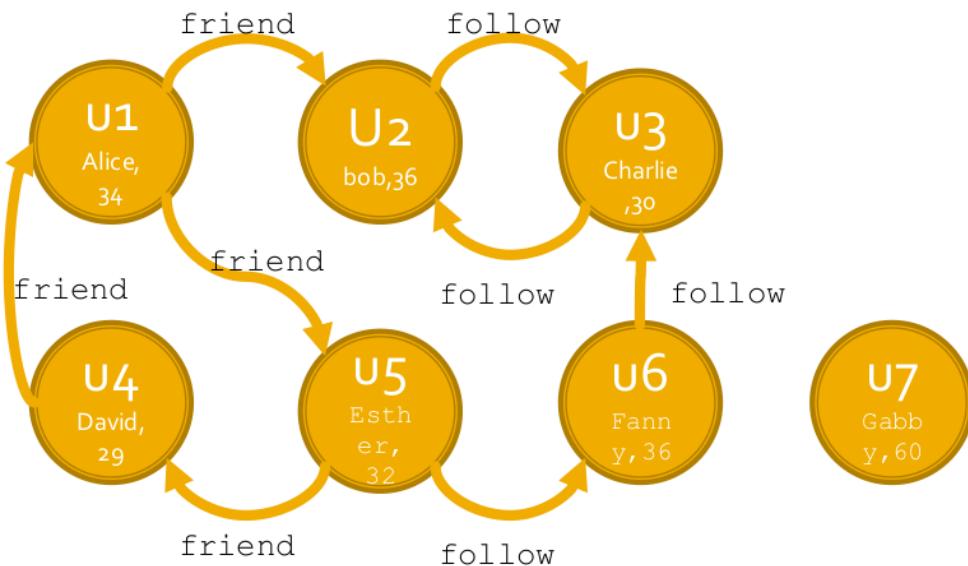
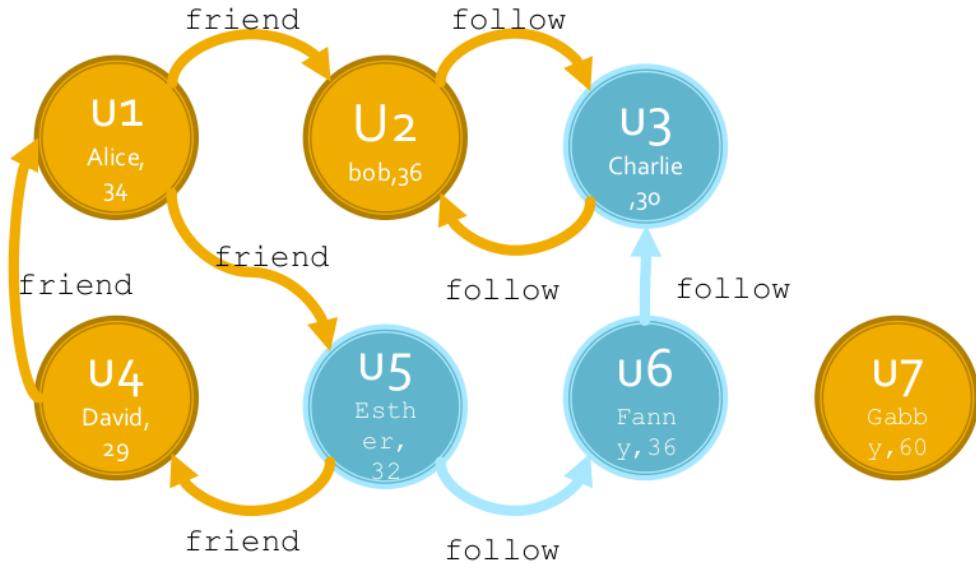


Figure 25.2: Resulting graph



The content of the returned DataFrame is the following

from	e0	v1	e1	to
[u5, Esther, 32]	[u5, u6, follow]	[u6, Fanny, 36]	[u6, u3, follow]	[u3, Charlie, 30]

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Search from vertex with name = "Esther" to vertex with name = "Charlie"
36 shortestPaths = g.bfs("name = 'Esther' ", "name = 'Charlie' ")

```

i Example 2

1. Find the shortest path from Alice to a user who is 30 years old
2. Store the result in a DataFrame

Figure 25.3: Example graph

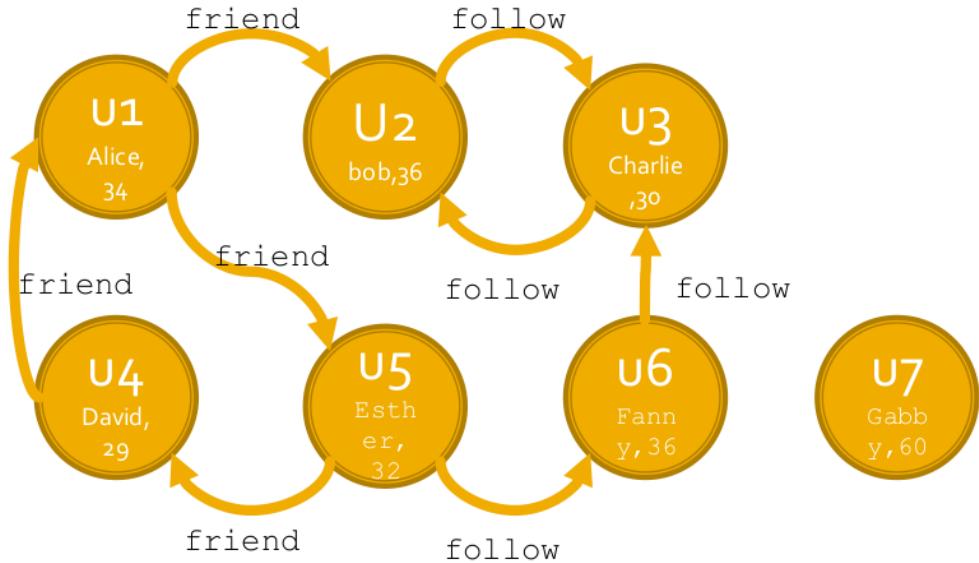
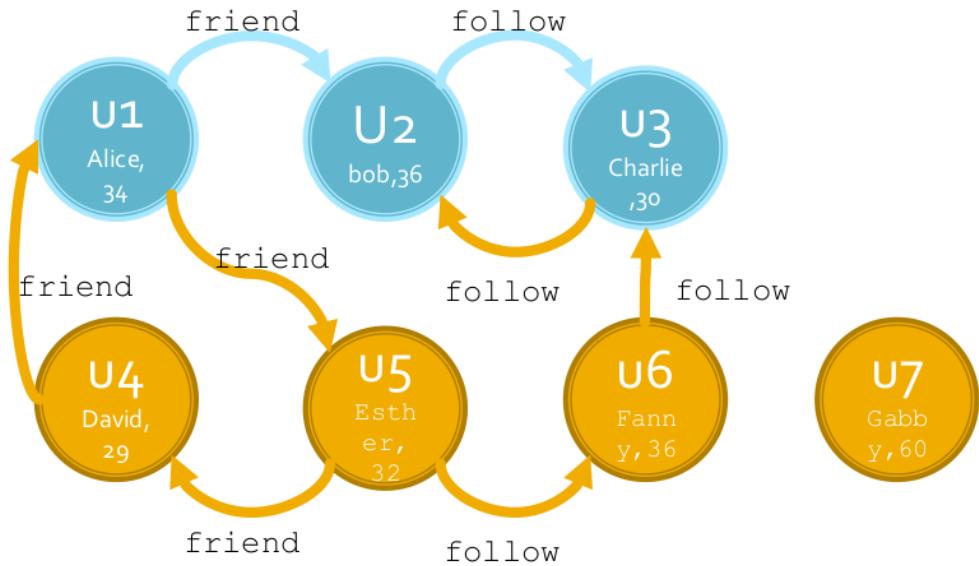


Figure 25.4: Resulting graph



```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Find the shortest path from Alice to a user who is 30 years old
36 shortestPaths = g.bfs("name = 'Alice' ", "age= 30")

```

i Example 3

1. Find the shortest path from any user who is less than 31 years old to any user who is more than 30 years old
2. Store the result in a DataFrame

Figure 25.5: Example graph

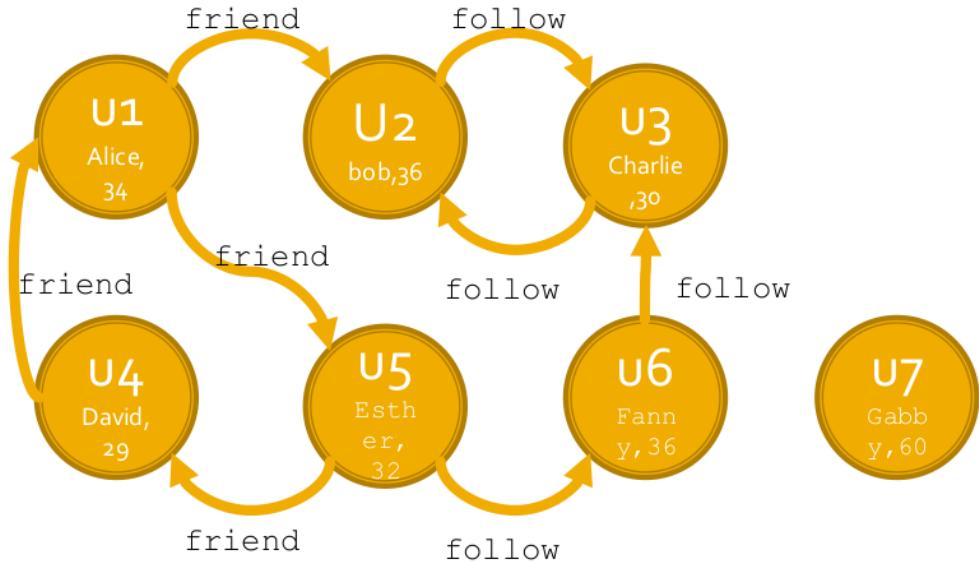
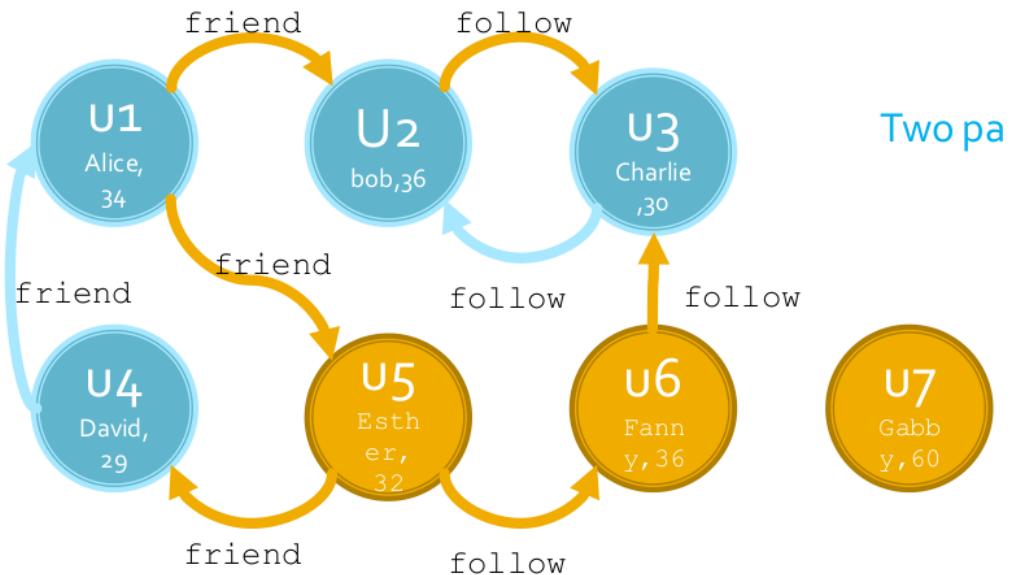


Figure 25.6: Resulting graph



Notice that two paths are selected in this case

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Find the shortest path from any user who is less than 31 years old
36 ## to any user who is more than 30 years old
37 shortestPaths = g.bfs("age<31", "age>30")

```

Example 4

1. Find the shortest path from Alice to any user who is less than 31 years old without using follow edges
2. Store the result in a DataFrame

Figure 25.7: Example graph

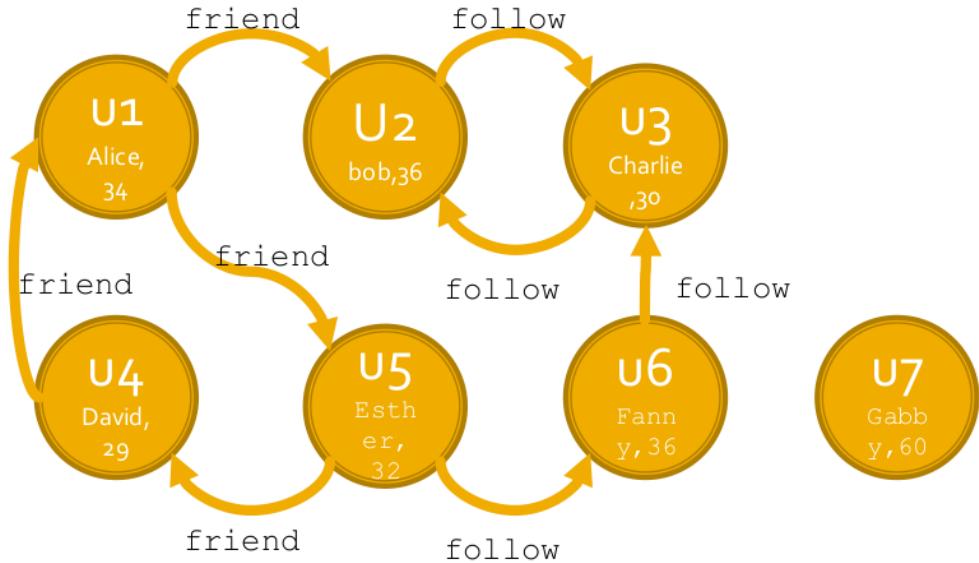
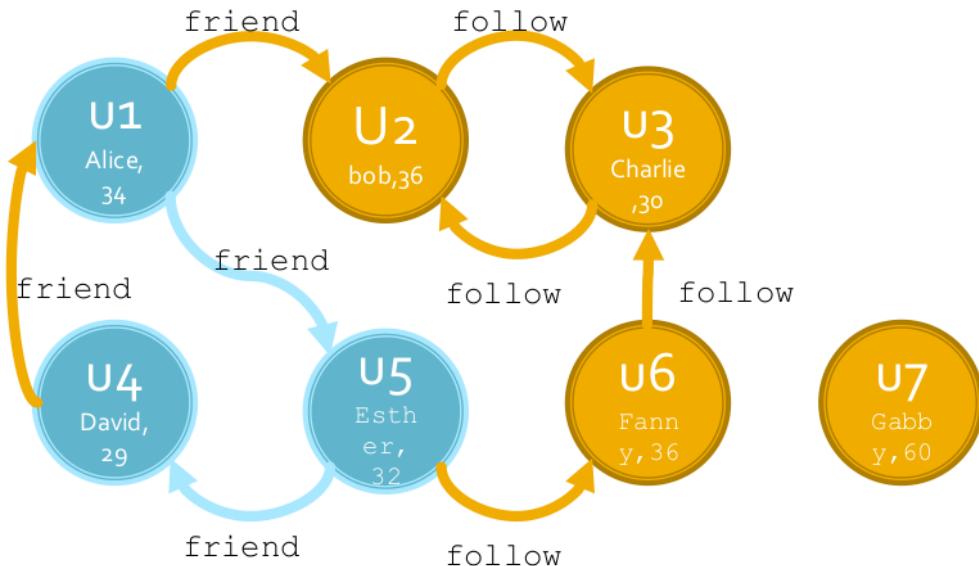


Figure 25.8: Resulting graph



Notice that two paths are selected in this case

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Find the shortest path from Alice to any user who is less
36 ## than 31 years old without using "follow" edges
37 shortestPaths = g.bfs("name = 'Alice' ", "age<31", "relationship<> 'follow' ")

```

Shortest path

The shortest path method selects the length of the shortest path(s) from each vertex to a given set of landmark vertexes. It uses the BFS algorithm for computing the shortest paths.

Implementation

```
1 shortestPaths(landmarks)
```

The `shortestPaths(landmarks)` method of the `GraphFrame` class returns the length of the shortest path(s) from each vertex to a given set of landmarks vertexes. For each vertex, one shortest path for each landmark vertex is computed and its length is returned.

- `landmarks`: list of IDs of landmark vertexes (e.g., $[u1, u4]$)

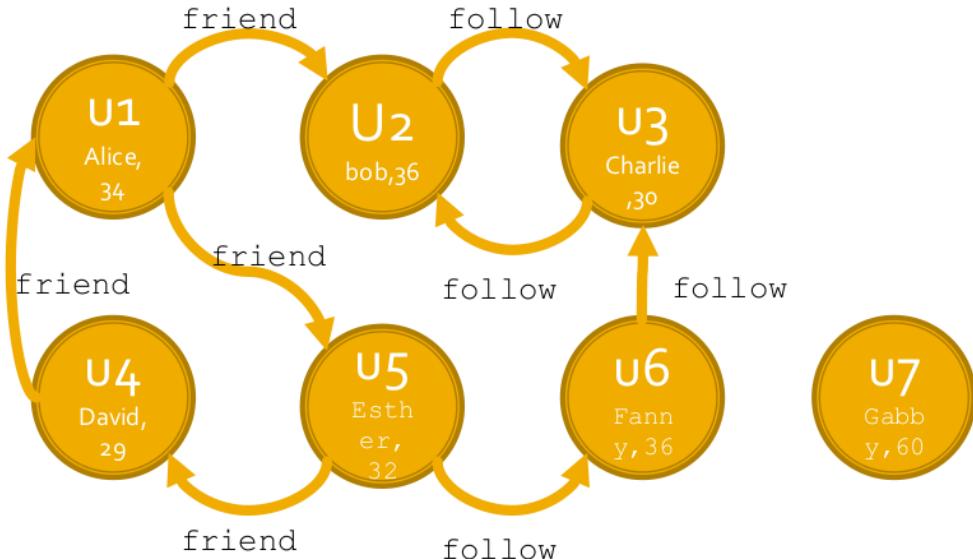
`shortestPaths()` returns a DataFrame that contains one record/row for each distinct vertex of the input graph (also for the non-connected ones). This method is characterized by the following columns

- one column for each attribute of the vertexes
- distances (type map): for each landmark lm it contains one pair (ID lm: length shortest path from the vertex of the current record to lm)

Example 1

1. Find for each user the length of the shortest path to user $u1$ (i.e., Alice)
2. Store the result in a DataFrame

Figure 25.9: Example graph



Vertex	Distance to $u1$
$u1$	0
$u2$	-
$u3$	-
$u4$	1

<i>u5</i>	2
<i>u6</i>	-
<i>u7</i>	-

The content of the returned DataFrame is the following

id	name	age	distances
<i>u1</i>	Alice	34	[<i>u1</i> → 0]
<i>u2</i>	Bob	36	[]
<i>u3</i>	Charlie	30	[]
<i>u4</i>	David	29	[<i>u1</i> → 1]
<i>u5</i>	Esther	32	[<i>u1</i> → 2]
<i>u6</i>	Fanny	36	[]
<i>u7</i>	Gabby	60	[]

- $[u1 \rightarrow 0]$: data type is map. It stores a set of pairs (Key: Value)

```

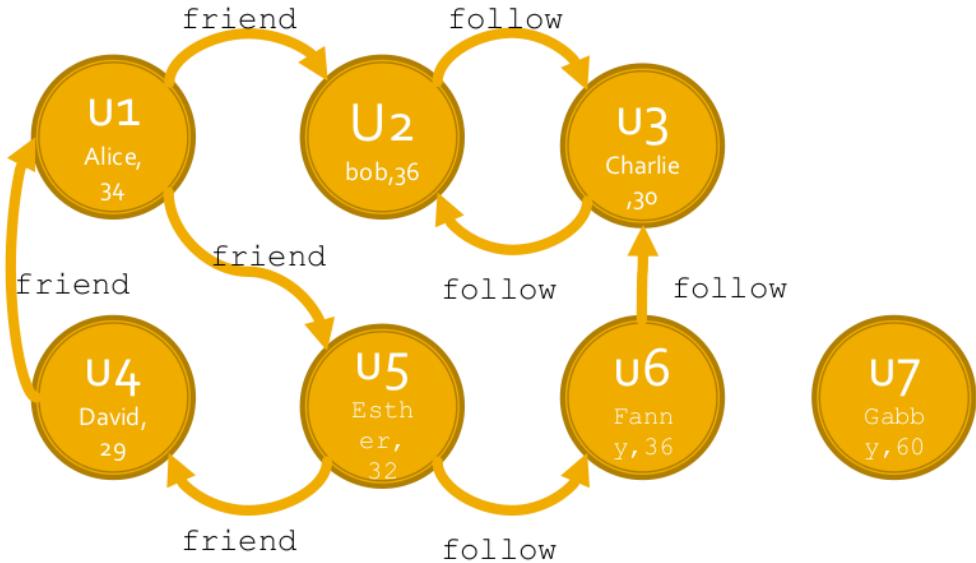
1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Find for each user the length of the shortest path to user u1
36 shortestPaths = g.shortestPaths(["u1"])

```

Example 2

1. Find for each user the length of the shortest path to users $u1$ (Alice) and $u4$ (David)
2. Store the result in a DataFrame

Figure 25.10: Example graph



Vertex	Distance to u_1	Distance to u_1
u_1	0	2
u_2	-	-
u_3	-	-
u_4	1	0
u_5	2	1
u_6	-	-
u_7	-	-

The content of the returned DataFrame is the following

	id	name	age	distances
	<i>u1</i>	Alice	34	[<i>u1</i> → 0, <i>u4</i> → 2]
	<i>u2</i>	Bob	36	[]
	<i>u3</i>	Charlie	30	[]
	<i>u4</i>	David	29	[<i>u1</i> → 1, <i>u4</i> → 0]
	<i>u5</i>	Esther	32	[<i>u1</i> → 2, <i>u4</i> → 1]
	<i>u6</i>	Fanny	36	[]
	<i>u7</i>	Gabby	60	[]

- $[u1 \rightarrow 0]$: data type is map. It stores a set of pairs (Key: Value)

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Find for each user the length of the shortest paths to users u1 and u4
36 shortestPaths = g.shortestPaths(["u1", "u4"])

```

Connected components

A connected component of a graph is a subgraph sg such that

- Any two vertexes in sg are connected to each other by at least one path
- The set of vertexes in sg is not connected to any additional vertexes in the original graph

The direction of edges is not considered.

Figure 25.11: Two connected components

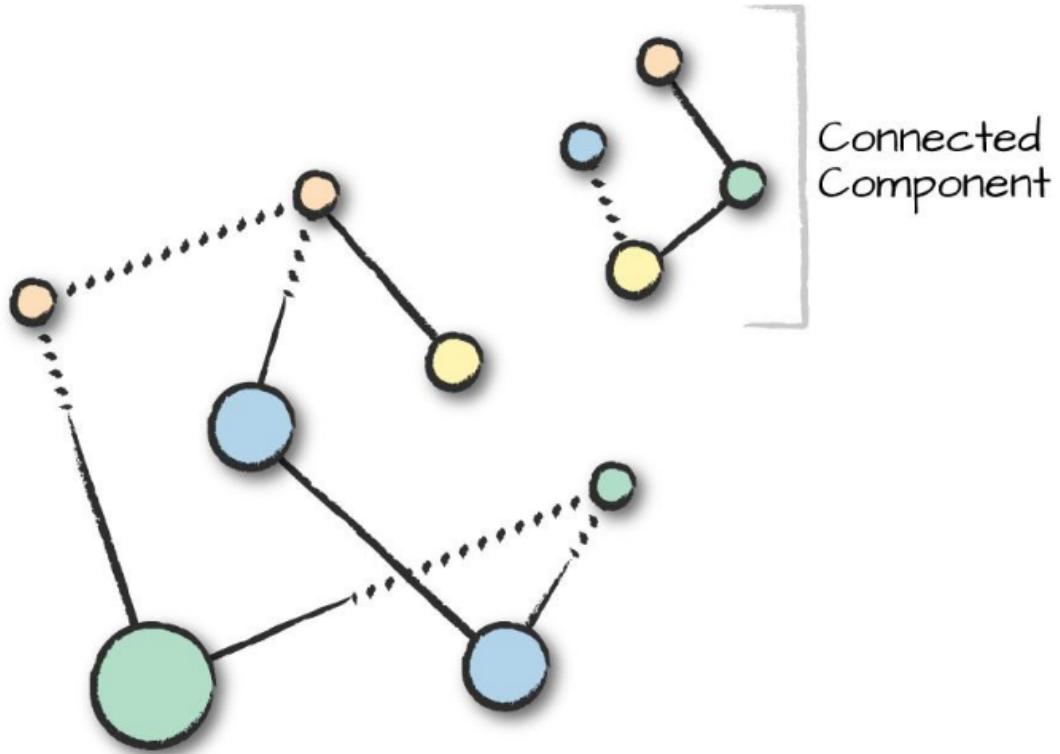
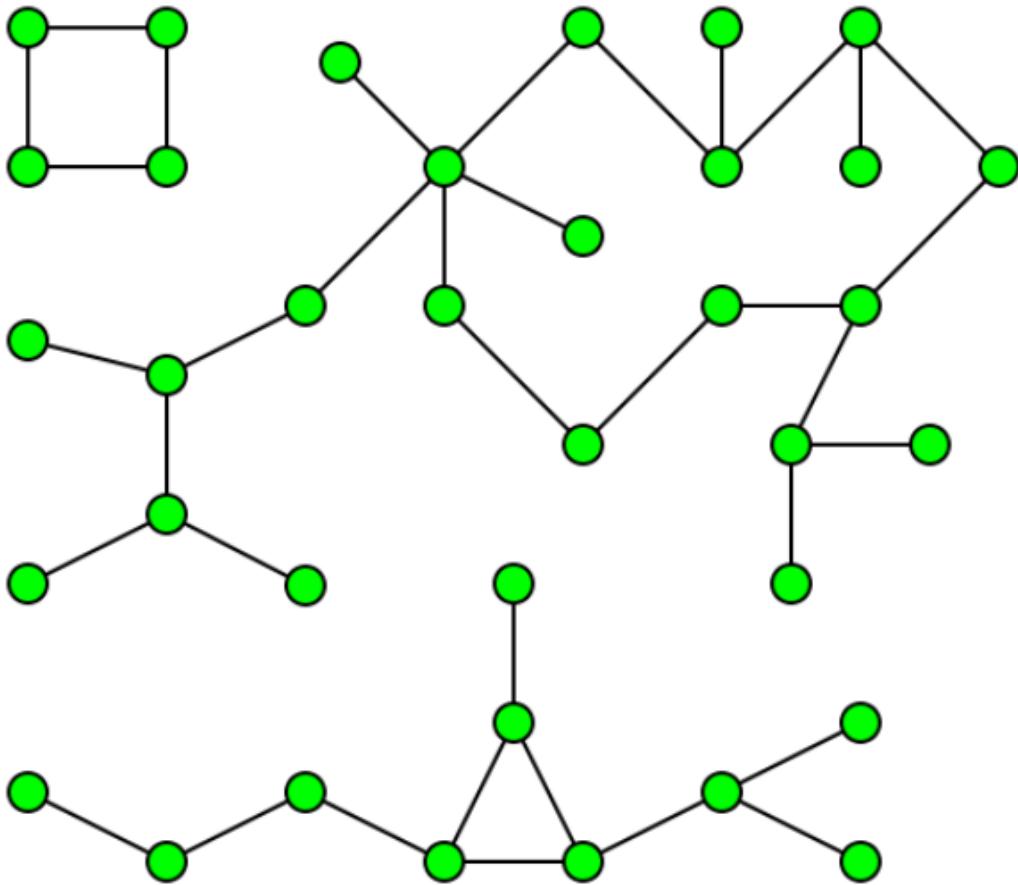


Figure 25.12: Three connected components



The `connectedComponents()` method of the `GraphFrame` class returns the connected components of the input graph. This is an expensive algorithm, and requires setting a Spark checkpoint directory.

Implementation

```
1 connectedComponents()
```

The `connectedComponents()` method returns a `DataFrame` that contains one record/row for each distinct vertex of the input graph. It is characterized by the following columns

- one column for each attribute of the vertexes
- component (type long). It is the identifier of the connected component to which the current vertex has been assigned.

i Example

Print on the stdout the number of connected components of the following graph

Figure 25.13: Example graph

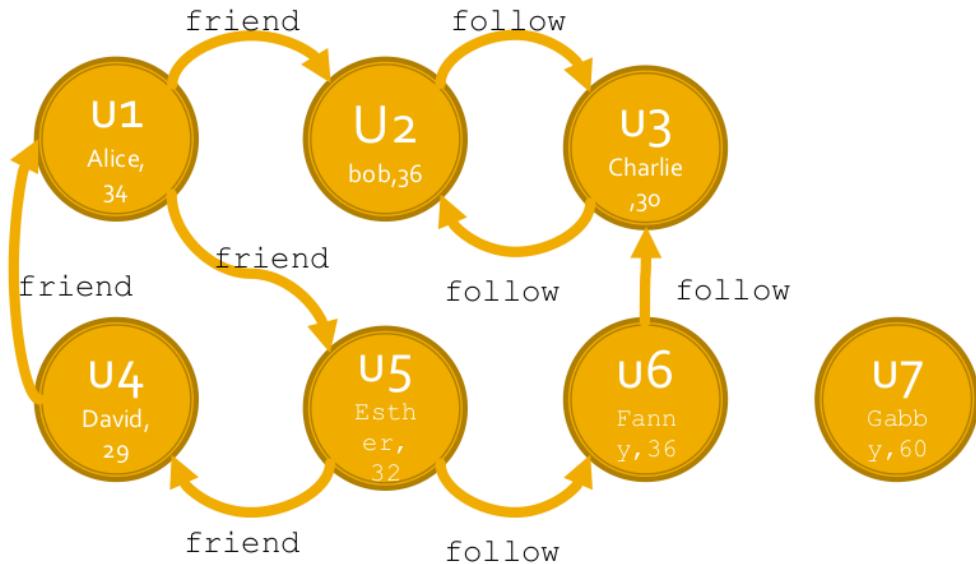
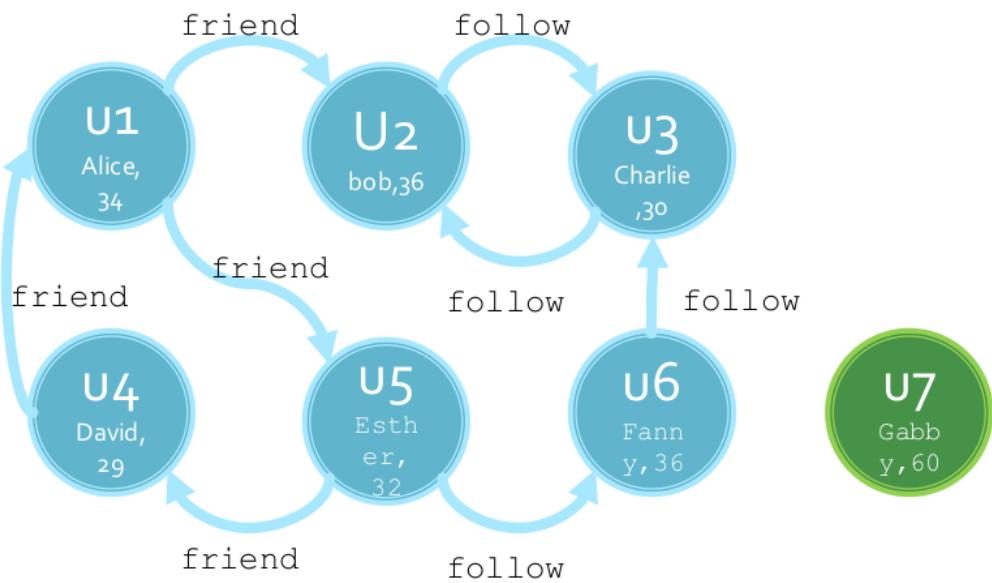


Figure 25.14: Result



Notice that there are two connected components on this graph.

This is the content of the DataFrame used to store the two identified connected components

	id	name	age	component
	<i>u6</i>	Fanny	36	146028888064
	<i>u1</i>	Alice	34	146028888064
	<i>u3</i>	Charlie	30	146028888064
	<i>u5</i>	Esther	32	146028888064
	<i>u2</i>	Bob	36	146028888064
	<i>u4</i>	David	29	146028888064
	<i>u7</i>	Gabby	60	1546188226560

Notice the “component” field

- “146028888064”: vertexes of the first component
- “1546188226560”: vertexes of the second component

```

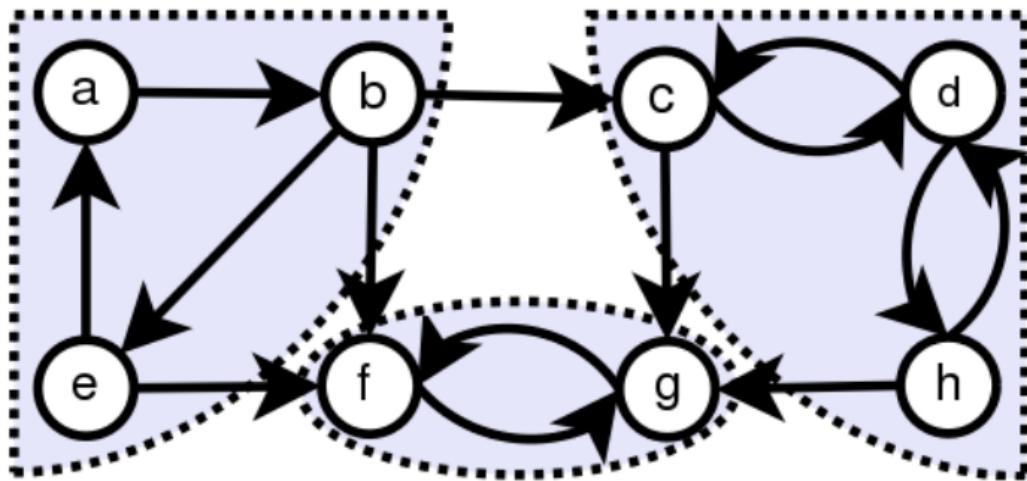
1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Set checkpoint folder
36 sc.setCheckpointDir("tmp_ckpts")
37
38 ## Run the algorithm
39 connComp=g.connectedComponents()
40
41 ## Count the number of components
42 nComp=connComp.select("component").distinct().count()
43
44 print("Number of connected components: ", nComp)

```

Strongly connected components

A directed subgraph sg is called strongly connected if every vertex in sg is reachable from every other vertex in sg . For undirected graph, connected and strongly connected components are the same.

Figure 25.15: Three strongly connected subgraphs/components



Implementation

```
1  stronglyConnectedComponents()
```

The `stronglyConnectedComponents()` method of the `GraphFrame` class returns the strongly connected components of the input graph. It is an expensive algorithm (better to run it on a cluster with yarn scheduler even with small graphs), and it requires setting a Spark checkpoint directory.

`stronglyConnectedComponents()` returns a `DataFrame` that contains one record/row for each distinct vertex of the input graph. It is characterized by the following columns

- one column for each attribute of the vertexes
- component (type long). It is the identifier of the strongly connected component to which the current vertex has been assigned.

i Example

Print on the stdout the number of strongly connected components of the input graph.

Figure 25.16: Example graph

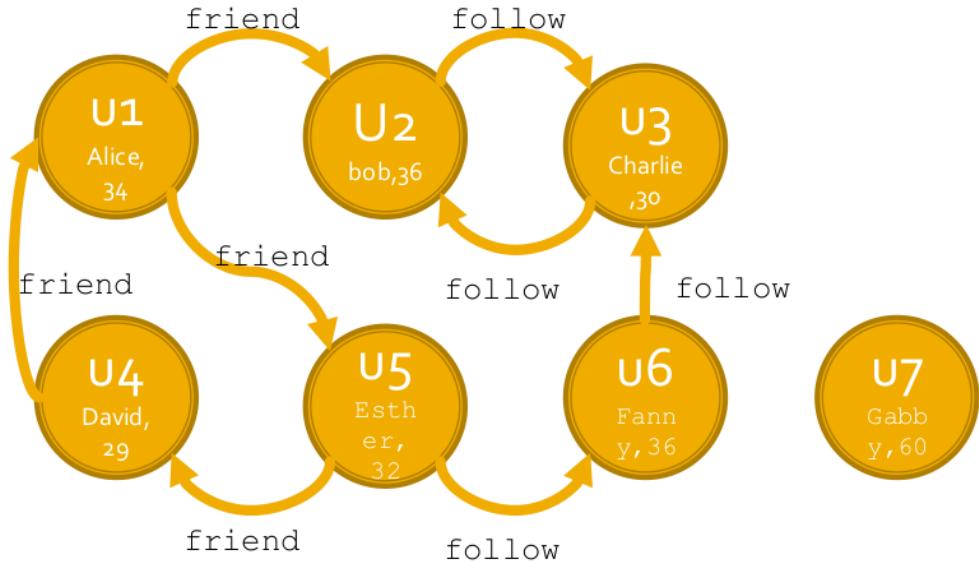
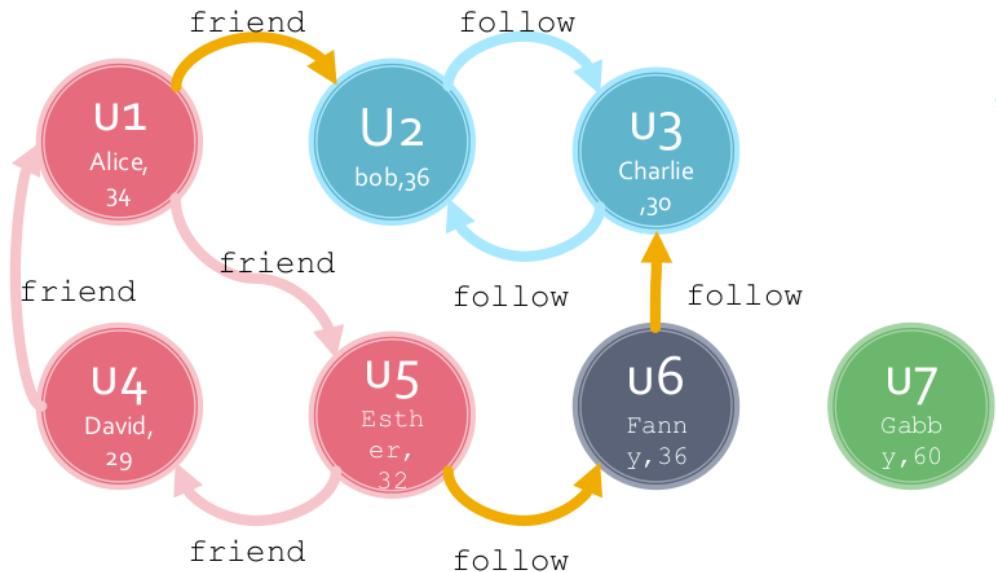


Figure 25.17: Resulting graph



Notice that there are four connected components on this graph.

This is the content of the DataFrame used to store the identified strongly connected components

id	name	age	component
u3	Charlie	30	146028888064
u2	Bob	36	146028888064
u1	Alice	34	498216206336

<i>u5</i>	Esther	32	498216206336
<i>u4</i>	David	29	498216206336
<i>u6</i>	Fanny	36	1090921693184
<i>u7</i>	Gabby	60	1546188226560

Notice the “component” field

- “146028888064”: vertexes of the first strongly connected component
- “498216206336”: vertexes of the second strongly connected component
- “1090921693184”: vertexes of the third strongly connected component
- “1546188226560”: vertexes of the fourth strongly connected component

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Set checkpoint folder
36 sc.setCheckpointDir("tmp_ckpts")
37
38 ## Run the algorithm
39 strongConnComp = g.stronglyConnectedComponents(maxIter=10)
40
41 ## Count the number of strongly connected components
42 nComp=strongConnComp.select("component").distinct().count()
43
44 print("Number of strongly connected components: ", nComp)

```

Label propagation

Label Propagation is an algorithm for detecting communities in graphs. It is similar to clustering, but exploits connectivity. Convergence is not guaranteed, and also it is possible to end up with trivial solutions.

The Label Propagation algorithm

Each vertex in the network is initially assigned to its own community: at every step, vertexes send their community affiliation to all neighbors and update their state to the mode community affiliation of incoming messages.

Implementation

```
1  labelPropagation(maxIter)
```

The `labelPropagation(maxIter)` method of the `GraphFrame` class runs and returns the result of the label propagation algorithm.

- `maxIter`: number of iterations to run

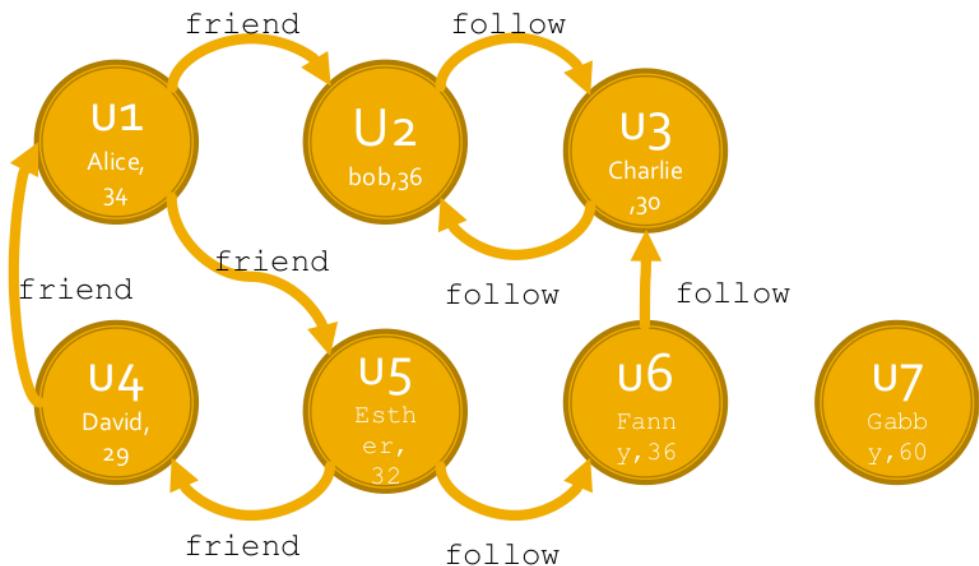
`labelPropagation()` returns a `DataFrame` that contains one record/Row for each distinct vertex of the input graph. It is characterized by the following columns

- one column for each attribute of the vertexes
- label (type long). It is the identifier of the community to which the current vertex has been assigned.

Example

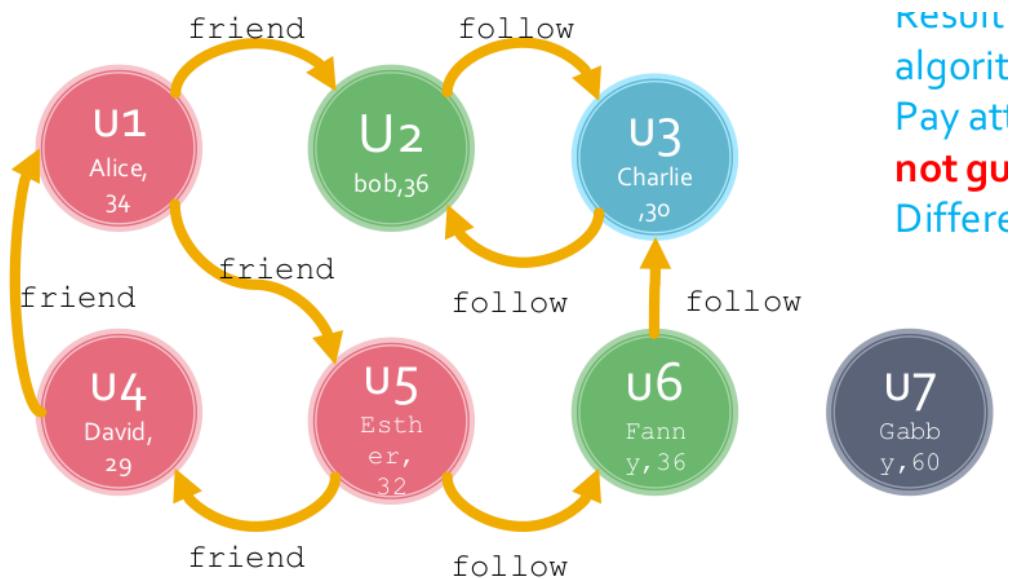
Split in groups the vertexes of the graph by using the label propagation algorithm.

Figure 25.18: Example graph



Notice that the result returned by one run of the algorithm. Pay attention that convergence is not guarantee, and different results may come out from different runs.

Figure 25.19: Results



This is the content of the DataFrame used to store the identified communities

	id	name	age	label
	u3	Charlie	30	146028888064
	u4	David	29	498216206336
	u1	Alice	34	498216206336

<i>u5</i>	Esther	32	498216206337
<i>u7</i>	Gabby	60	1546188226560
<i>u2</i>	Bob	36	1606317768704
<i>u6</i>	Fanny	36	1606317768704

- “146028888064”: vertexes of the first community
- “498216206336”: vertexes of the second community
- “1546188226560”: vertexes of the third community
- “1606317768704”: vertexes of the fourth community

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Run the label propagation algorithm
36 labelComm = g.labelPropagation(10)

```

PageRank

PageRank is the original famous algorithm used by the Google Search engine to rank vertexes (web pages) in a graph by order of importance. For the Google search engine vertexes are web pages in the World Wide Web, and edges are hyperlinks among web pages. This algorithm assigns a numerical weighting (importance) to each node.

It computes a likelihood that a person randomly clicking on links will arrive at any particular web page. For having a high PageRank, it is important to

- Have many in-links
- Be liked by relevant pages (pages characterized by a high PageRank)

The basic idea is that each link vote is proportional to the importance of its source page p : if page p with importance $\text{PageRank}(p)$ has n out-links, each out-link gets $\frac{\text{PageRank}(p)}{n}$ votes; the importance of page p is the sum of the votes on its in-links.

Simple recursive formulation

- Initialize each page rank to 1.0: for each p in pages set $\text{PageRank}(p)$ to 1.0
- Iterate for max iterations
 1. Page p sends a contribution $\frac{\text{PageRank}(p)}{\text{numOutLinks}(p)}$ to its neighbors (the pages it links);
 2. Update each page rank $\text{PageRank}(p)$ with the sum of the received contributions.

Random jumps formulation

The PageRank algorithm simulates the “random walk” of a user on the web. Indeed, at each step of the random walk, the random surfer has two options:

- with probability $1 - \alpha$, follow a link at random among the ones in the current page;
- with probability α , jump to a random page.
- Initialize each page rank to 1.0: for each p in pages set $\text{PageRank}(p)$ to 1.0
- Iterate for max iterations
 1. Page p sends a contribution $\frac{\text{PageRank}(p)}{\text{numOutLinks}(p)}$ to its neighbors (the pages it links);
 2. Update each page rank $\text{PageRank}(p)$ to $\alpha + (1 - \alpha)$ times the sum of the received contributions.

i Example

- $\alpha = 0.15$
- Initialization: $\forall p, \text{PageRank}(p) = 1.0$

Figure 25.20: Initialization

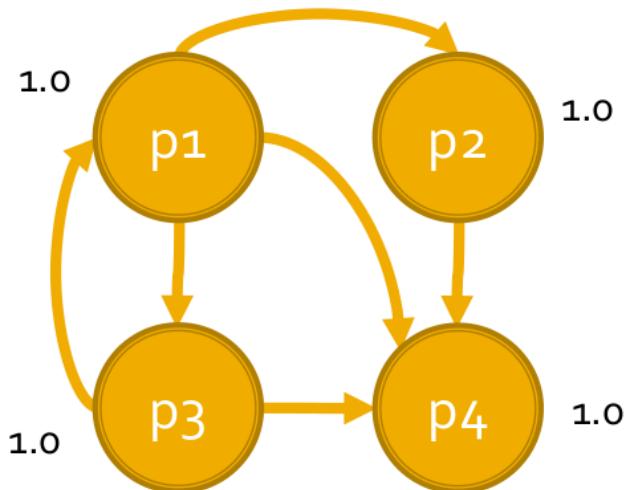
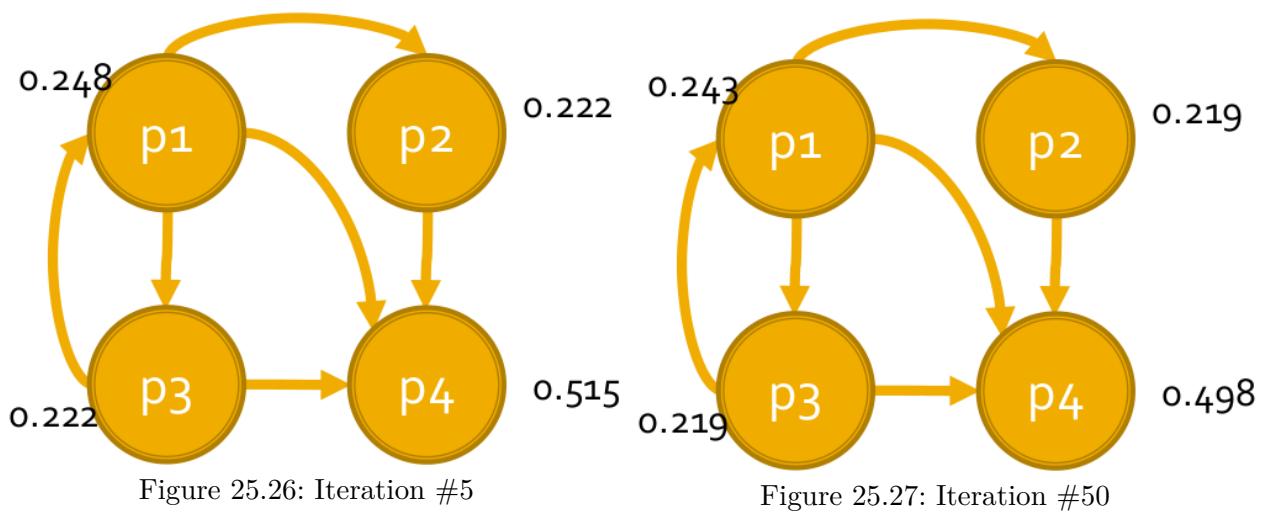
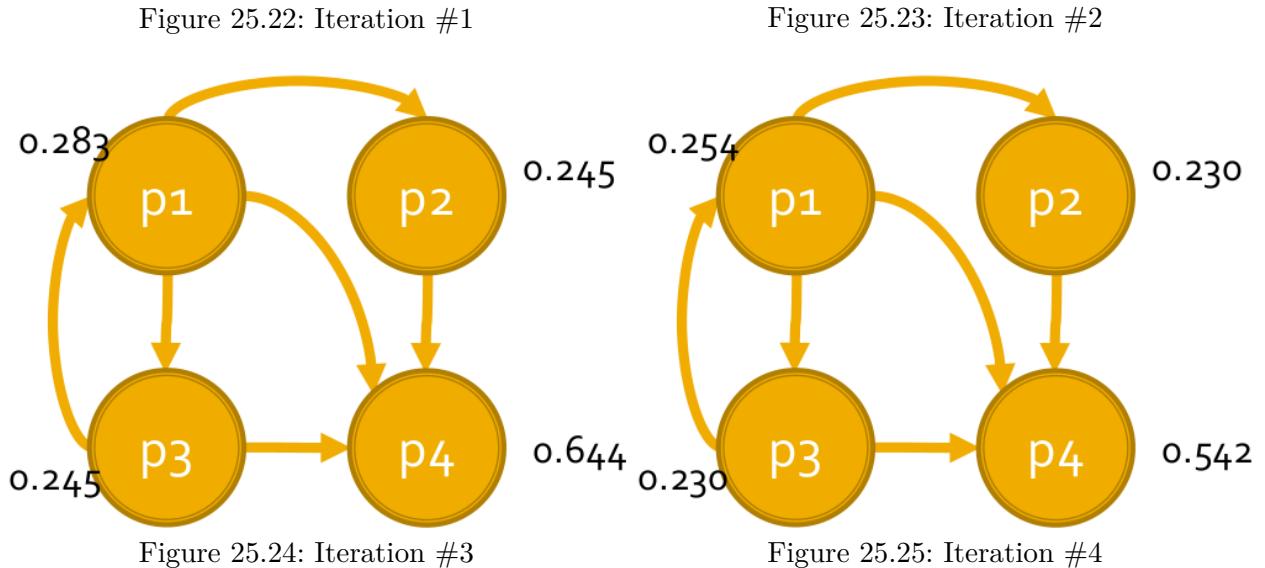
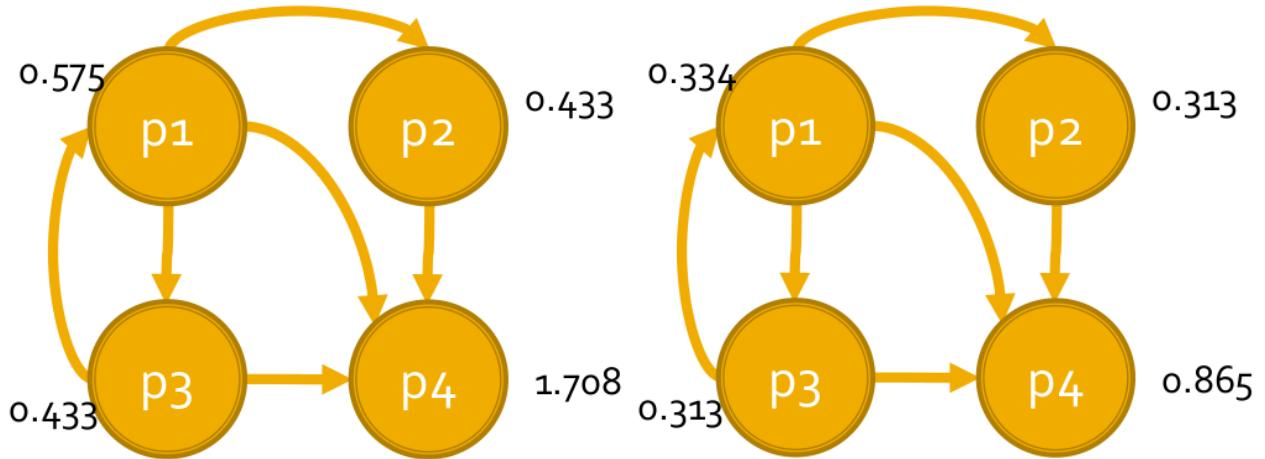


Figure 25.21: Iterations



Implementation

```
1 pageRank(resetProbability, maxIter, tol, sourceId)
```

The `pageRank()` method of the `GraphFrame` class runs the PageRank algorithm on the input graph.

- `resetProbability`: probability of resetting to a random vertex (probability α associated with random jumps);
- `maxIter`: if set, the algorithm is run for a fixed number of iterations; this may not be set if the `tol` parameter is set;
- `tol`: if set, the algorithm is run until the given tolerance; this may not be set if the `numIter` parameter is set;
- `sourceId`: the source vertex for a personalized PageRank (optional parameter).

`pageRank()` returns a new `GraphFrame` that contains the same vertexes and edges of the input graph

- the vertexes of the new graph are characterized by one new attribute, called “`pagerank`”, that stores the PageRank of the vertexes;
- the edges of the new graph are characterized by one new attribute, called “`weight`”, that stores the weight (PageRank contribution) propagated through that edge.

Example

Apply the PageRank algorithm on the following graph and select the user associated with the highest PageRank value.

Figure 25.28: Example graph

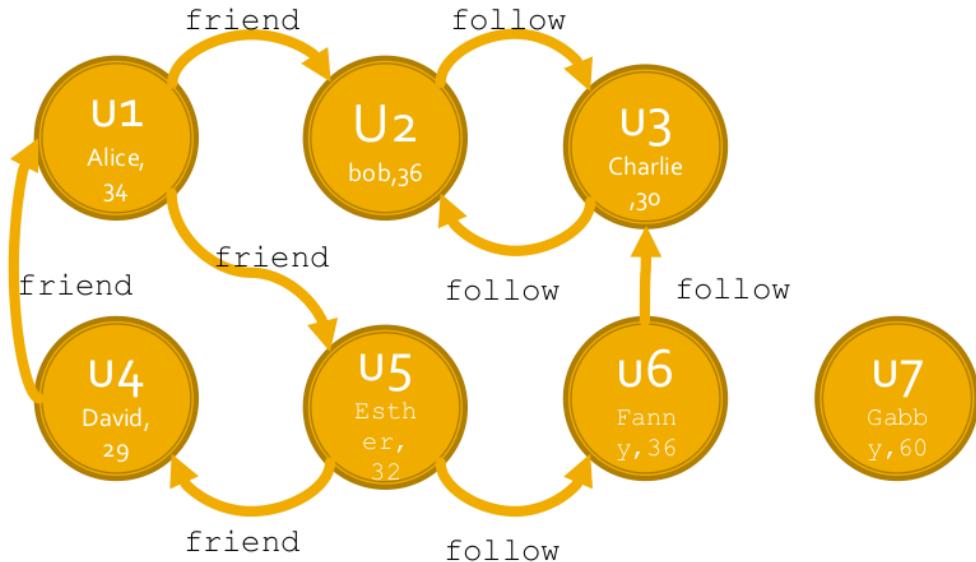
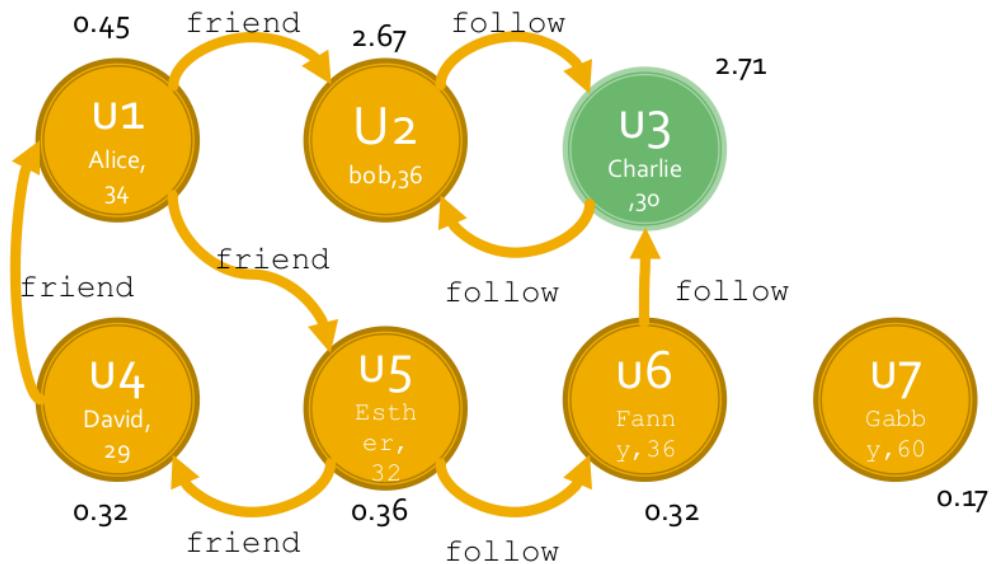


Figure 25.29: Resulting graph



```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## Run the PageRank algorithm
36 pageRanks = g.pageRank(maxIter=30)
37
38 ## Select the maximum value of PageRank
39 maxPageRank = pageRanks.vertices \
40     .agg({"pagerank":"max"}) \
41     .first()["max(pagerank)"]
42
43 ## Select the user with the maximum PageRank
44 pageRanks.vertices \
45     .filter(pageRanks.vertices.pagerank==maxPageRank) \
46     .show()

```

Custom graph algorithms

GraphFrames provides primitives for developing your own other graph algorithms. It is based on message passing approach: the two key components are:

- `aggregateMessages`: it sends messages between vertexes, and aggregate messages for each vertex
- joins: it joins message aggregates with the original graph

 Note

For each user, compute the sum of the ages of adjacent users (count many times the same adjacent user if there are many links).

The resulting table is

Vertex	SumAges
u_1	97
u_2	94
u_3	108
u_4	66
u_5	99
u_6	62

```

1  from graphframes import GraphFrame
2
3  ## Vertex DataFrame
4  v = spark.createDataFrame(
5      [
6          ("u1", "Alice", 34),
7          ("u2", "Bob", 36),
8          ("u3", "Charlie", 30),
9          ("u4", "David", 29),
10         ("u5", "Esther", 32),
11         ("u6", "Fanny", 36),
12         ("u7", "Gabby", 60)
13     ],
14     ["id", "name", "age"]
15 )
16
17 ## Edge DataFrame
18 e = spark.createDataFrame(
19     [
20         ("u1", "u2", "friend"),
21         ("u2", "u3", "follow"),
22         ("u3", "u2", "follow"),
23         ("u6", "u3", "follow"),
24         ("u5", "u6", "follow"),
25         ("u5", "u4", "friend"),
26         ("u4", "u1", "friend"),
27         ("u1", "u5", "friend")
28     ],
29     ["src", "dst", "relationship"]
30 )
31
32 ## Create the graph
33 g = GraphFrame(v, e)
34
35 ## For each user, sum the ages of the adjacent users
36 ## Send the age of each destination of an edge to its source
37 msgToSrc = AggregateMessages.dst["age"]
38
39 ## Send the age of each source of an edge to its destination
40 msgToDst = AggregateMessages.src["age"]
41
42 ## Aggregate messages
43 aggAge = g.aggregateMessages(
44     sum(AggregateMessages.msg),
45     sendToSrc=msgToSrc,
46     sendToDst=msgToDst
47 )
48
49 #Show result
50 aggAge.show()

```

26 Streaming data analytics frameworks

26.1 Introduction

What is streaming processing?

Streaming processing is the act of continuously incorporating new data to compute a result. Input data is unbounded (i.e., it has no beginning and no end). Series of events that arrive at the stream processing system, and the application will output multiple versions of the results as it runs or put them in a storage.

Many important applications must process large streams of live data and provide results in near-real-time

- Social network trends
- Website statistics
- Intrusion detection systems
- ...

The main advantages of stream processing are:

- Vastly higher throughput in data processing
- Low latency: application respond quickly (e.g., in seconds). It can keep states in memory
- More efficient in updating a result than repeated batch jobs, because it automatically incrementalizes the computation

Some requirements and challenges are:

- Scalable to large clusters
- Responding to events at low latency
- Simple programming model
- Processing each event exactly once despite machine failures - Efficient fault-tolerance in stateful computations
- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of state
- Handling load imbalance and stragglers
- Updating your application's business logic at runtime

Stream processing frameworks for big streaming data analytics

Several frameworks have been proposed to process in real-time or in near real-time data streams

- Apache Spark (Streaming component)
- Apache Storm
- Apache Flink
- Apache Samza
- Apache Apex
- Apache Flume
- Amazon Kinesis Streams
- ...

All these frameworks use a cluster of servers to scale horizontally with respect to the (big) amount of data to be analyzed.

Main solutions

There are two main solutions

- **Continuous computation of data streams.** In this case, data are processed as soon as they arrive: every time a new record arrives from the input stream, it is immediately processed and a result is emitted as soon as possible. This is real-time processing.
- **Micro-batch stream processing.** Input data are collected in micro-batches, where each micro-batch contains all the data received in a time window (typically less than a few seconds of data). One micro-batch at a time is processed: every time a micro-batch of data is ready, its entire content is processed and a result is emitted. This is near real-time processing.

Figure 26.1: Continuous computation: one record at a time

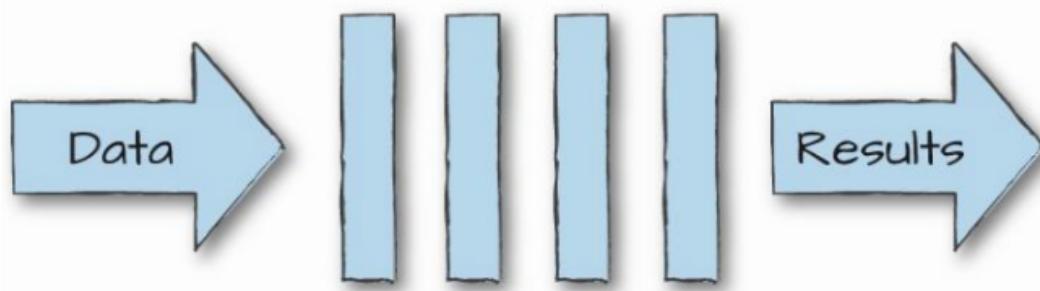
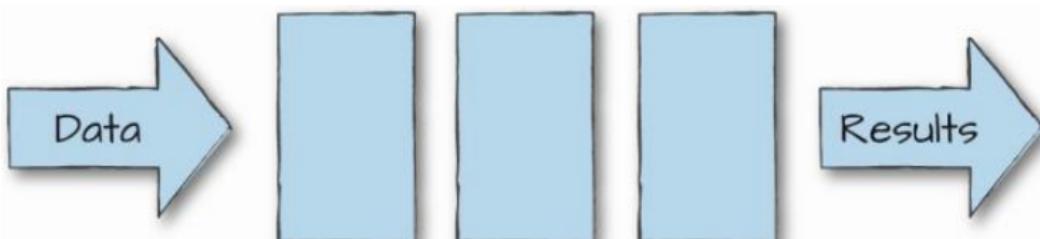


Figure 26.2: Micro-batch computation: one micro-batch at a time



Input data processing and result guarantees

- At-most-once
 - Every input element of a stream is processed once or less
 - It is also called no guarantee
 - The result can be wrong/approximated
- At-least-once
 - Every input element of a stream is processed once or more
 - Input elements are replayed when there are failures
 - The result can be wrong/approximated
- Exactly-once
 - Every input element of a stream is processed exactly once
 - Input elements are replayed when there are failures
 - If elements have been already processed they are not reprocessed
 - The result is always correct
 - Slower than the other processing approaches

26.2 Spark Streaming

What is Spark Streaming

Spark Streaming is a framework for large scale stream processing

- Scales to 100s of nodes
- Can achieve second scale latencies
- Provides a simple batch-like API for implementing complex algorithm
- Micro-batch streaming processing
- Exactly-once guarantees
- Can absorb live data streams from Kafka, Flume, ZeroMQ, Twitter, ...

Figure 26.3: Spark Streaming components



Many important applications must process large streams of live data and provide results in near-real-time

- Social network trends
- Website statistics
- Intrusion detection systems
- ...

The requirements are

- Scalable to large clusters
- Second-scale latencies
- Simple programming model
- Efficient fault-tolerance in stateful computations

Spark discretized stream processing

Spark streaming runs a streaming computation as a series of very small, deterministic batch jobs. It splits each input stream in portions and processes one portion at a time (in the incoming order): the same computation is applied on each portion (called **batch**) of the stream.

So, Spark streaming

- Splits the live stream into batches of X seconds
- Treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches

Figure 26.4: Discretization in batches

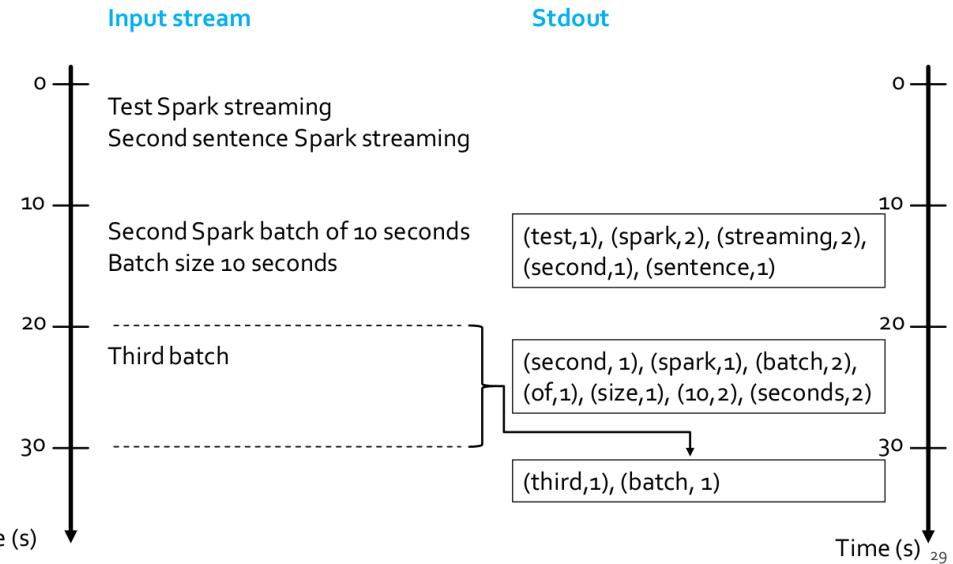


Example

Word count implementation using Spark streaming. Problem specification:

- The input is a stream of sentences
- Split the input stream in batches of 10 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch (i.e., execute the word count application one time for each batch of 10 seconds)

Figure 26.5: Input and output



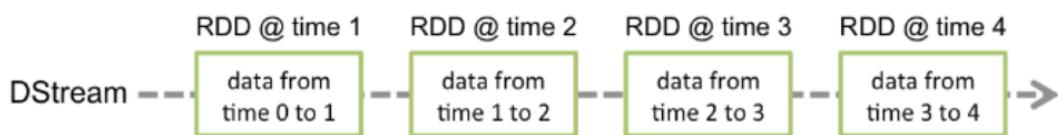
💡 Key concepts

- DStream
 - Sequence of RDDs representing a discretized version of the input stream of data (Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets, ...)
 - One RDD for each batch of the input stream
- Transformations
 - Modify data from one DStream to another
 - Standard RDD operations (map, countByValue, reduce, join, ...)
 - Window and Stateful operations (window, countByValueAndWindow, ...)
- Output Operations/Actions
 - Send data to external entity (saveAsHadoopFiles, saveAsTextFile, ...)

Word count using DStreams

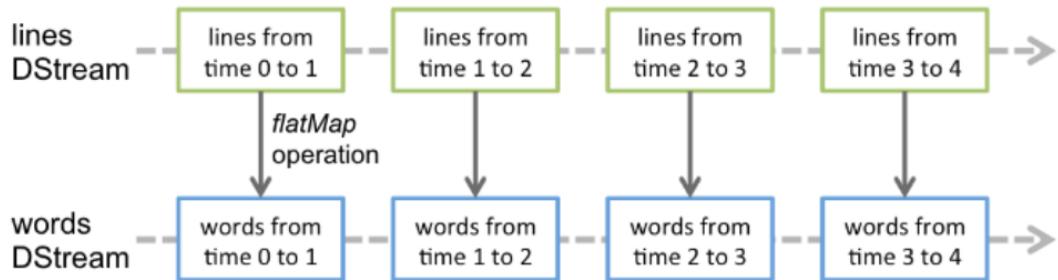
A DStream is represented by a continuous series of RDDs. Each RDD in a DStream contains data from a certain batch/interval.

Figure 26.6: RDDs composing a DStreams



Any operation applied on a DStream translates to operations on the underlying RDDs. These underlying RDD transformations are computed by the Spark engine.

Figure 26.7: Operations in a DStreams



Fault-tolerance

DStreams remember the sequence of operations that created them from the original fault-tolerant input data. Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant: data lost due to worker failure, can be recomputed from input data.

26.3 Spark streaming programs

Basic structure of a Spark streaming program

1. Define a Spark Streaming Context object. Define the size of the batches (in seconds) associated with the Streaming context.
2. Specify the input stream and define a DStream based on it
3. Specify the operations to execute for each batch of data
4. Use transformations and actions similar to the ones available for standard RDDs
5. Invoke the start method, to start processing the input stream
6. Wait until the application is killed or the timeout specified in the application expires: if the timeout is not set and the application is not killed the application will run forever

Spark streaming context

The Spark Streaming Context is defined by using the `StreamingContext(SparkConf sparkC, Duration batchDuration)` constructor of the class `pyspark.streaming.StreamingContext`. The `batchDuration` parameter specifies the size of the batches in seconds

i Example

```

1  from pyspark.streaming import StreamingContext
2  ssc = StreamingContext(sc, 10)

```

The input streams associated with this context will be split in batches of 10 seconds.

After a context is defined, the next steps are

- Define the input sources by creating input Dstreams
- Define the streaming computations by applying transformation and output operations to DStreams

Input streams

The input Streams can be generated from different sources

- TCP socket, Kafka, Flume, Kinesis, Twitter.
- Also a HDFS folder can be used as input stream. This option is usually used during the application development to perform a set of initial tests.

Input: TCP socket

A DStream can be associated with the content emitted by a TCP socket: `socketTextStream(String hostname, int port_number)` is used to create a DStream based on the textual content emitted by a TCP socket.

i Example

```
1 lines = ssc.socketTextStream("localhost", 9999)
```

It stores the content emitted by localhost:9999 in the lines DStream.

Input: (HDFS) folder

A DStream can be associated with the content of an input (HDFS) folder: every time a new file is inserted in the folder, the content of the file is stored in the associated DStream and processed. Pay attention that updating the content of a file does not trigger/change the content of the DStream. `textFileStream(String folder)` is used to create a DStream based on the content of the input folder.

i Example

```
1 lines = textFileStream(inputFolder)
```

Store the content of the files inserted in the input folder in the lines Dstream: every time new files are inserted in the folder their content is stored in the current batch of the stream.

Input: other sources

Usually DStream objects are defined on top of streams emitted by specific applications that emit real-time streaming data (e.g., Apache Kafka, Apache Flume, Kinesis, Twitter). It is also possible to write custom applications for generating streams of data, however Kafka, Flume and similar tools are usually a more reliable and effective solutions for generating streaming data.

Transformations

Analogously to standard RDDs, also DStreams are characterized by a set of transformations that, when applied to DStream objects, return a new DStream Object. The transformation is applied on one batch (RDD) of the input DStream at a time and returns a batch (RDD) of the new DStream (i.e., each batch (RDD) of the input DStream is associated with exactly one batch (RDD) of the returned DStream). Many of the available transformations are the same transformations available for standard RDDs.

Basic transformations

Transformation	Effect
<code>map(func)</code>	It returns a new DStream by passing each element of the source DStream through a function func.
<code>flatMap(func)</code>	each input item can be mapped to 0 or more output items. Returns a new DStream.
<code>filter(func)</code>	It returns a new DStream by selecting only the records of the source DStream on which func returns true.
<code>reduce(func)</code>	It returns a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function func. The function must be associative and commutative so that it can be computed in parallel. Note that the <code>reduce</code> method of DStreams is a transformation.
<code>reduceByKey(func)</code>	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.
<code>combineByKey(...)</code>	when called on a DStream of (K, V) pairs, returns a new DStream of (K, W) pairs where the values for each key are aggregated using the given combine functions. The parameters are: <code>createCombiner</code> , <code>mergeValue</code> , and <code>mergeCombiners</code>
<code>groupByKey()</code>	when called on a DStream of (K, V) pairs, returns a new DStream of $(K, \text{Iterable} < V >)$ pairs where the values for each key is the concatenation of all the values associated with key K (i.e., It returns a new DStream by applying groupByKey on one batch (one RDD) of the input stream at a time).
<code>countByValue()</code>	when called on a DStream of elements of type K , returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each batch of the source Dstream. Note that the <code>countByValue</code> method of DStreams is a transformation.

Transformation	Effect
<code>count()</code>	It returns a new DStream of single-element RDDs by counting the number of elements in each batch (RDD) of the source Dstream (i.e., it counts the number of elements in each input batch (RDD)). Note that the <code>count</code> method of DStreams is a transformation.
<code>union(otherStream)</code>	It returns a new DStream that contains the union of the elements in the source DStream and otherDStream.
<code>join(otherStream)</code>	when called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of $(K, (V, W))$ pairs with all pairs of elements for each key.
<code>cogroup(otherStream)</code>	when called on a DStream of (K, V) and (K, W) pairs, return a new DStream of $(K, Seq[V], Seq[W])$ tuples.

Basic actions

Action | Effect |

`pprint()` | It prints the first 10 elements of every batch of data in a DStream on the standard output of the driver node running the streaming application. It is useful for development and debugging |
`saveAsTextFiles(prefix, [suffix])` | It saves the content of the DStream on which it is invoked as text files: one folder for each batch, and the folder name at each batch interval is generated based on prefix, time of the batch (and suffix): “prefix-TIME_IN_MS[.suffix]” (e.g., `Counts.saveAsTextFiles(outputPathPrefix, "")`). |

Start and run the computations

The `streamingContext.start()` method is used to start the application on the input stream(s). The `awaitTerminationOrTimeout(long millisecns)` method is used to specify how long the application will run.

The `awaitTermination()` method is used to run the application forever

- Until the application is explicitly killed
- The processing can be manually stopped using `streamingContext.stop()`

Points to remember

- Once a context has been started, no new streaming computations can be set up or added to it
- Once a context has been stopped, it cannot be restarted
- Only one StreamingContext per application can be active at the same time
- `stop()` on StreamingContext also stops the SparkContext. To stop only the `StreamingContext`, set the optional parameter of `stop()` called `stopSparkContext` to False

i Example: Spark Streaming version of word count

Problem specification

- Input: a stream of sentences retrieved from localhost:9999
- Task:
 - Split the input stream in batches of 5 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch (i.e., execute the word count problem for each batch of 5 seconds)
 - Store the results also in an HDFS folder

```

1  from pyspark.streaming import StreamingContext
2
3  ## Set prefix of the output folders
4  outputPathPrefix="resSparkStreamingExamples"
5
6  ## Create a configuration object and
7  ## set the name of the applicationconf
8  SparkConf().setAppName("Streaming word count")
9
10 ## Create a Spark Context object
11 sc = SparkContext(conf=conf)
12
13 ## Create a Spark Streaming Context object
14 ssc = StreamingContext(sc, 5)
15
16 ## Create a (Receiver) DStream that will connect to localhost:9999
17 lines = ssc.socketTextStream("localhost", 9999)
18
19 ## Apply a chain of transformations to perform the word count task
20 ## The returned RDDs are DStream RDDs
21 words = lines.flatMap(lambda line: line.split(" "))
22 wordsOnes = words.map(lambda word: (word, 1))
23 wordsCounts = wordsOnes.reduceByKey(lambda v1, v2: v1+v2)
24
25 ## Print the result on the standard output
26 wordsCounts.pprint()
27
28 ## Store the result in HDFS
29 wordsCounts.saveAsTextFiles(outputPathPrefix, "")
30
31 #Start the computation
32 ssc.start()
33
34 ## Run this application for 90 seconds
35 ssc.awaitTerminationOrTimeout(90)
36 ssc.stop(stopSparkContext=False)

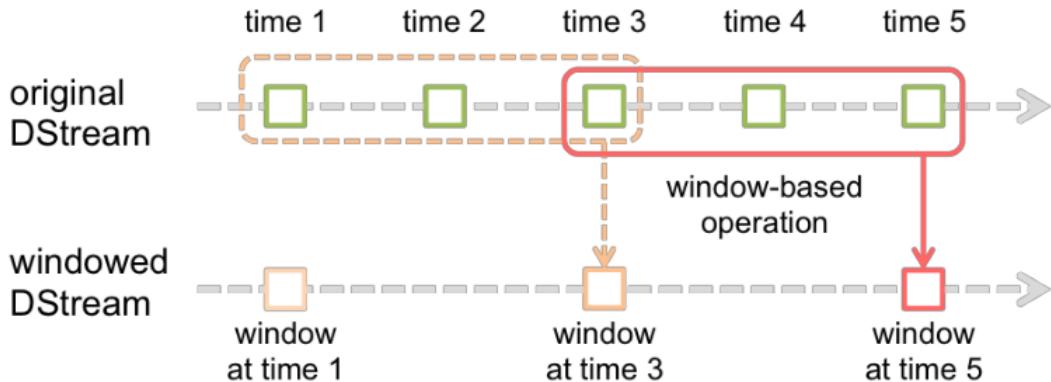
```

26.4 Windowed computation

Spark Streaming also provides windowed computations, allowing to apply transformations over a sliding window of data: each window contains a set of batches of the input stream, and windows can be overlapped (i.e., the same batch can be included in many consecutive windows).

Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.

Figure 26.8: Graphical example



In the example, the operation is applied over the last 3 time units of data (i.e., the last 3 batches of the input DStream), and each window contains the data of 3 batches. It slides by 2 time units.

Parameters

Any window operation needs to specify two parameters

- Window length: the duration of the window (3 in the example)
- Sliding interval: the interval at which the window operation is performed (2 in the example)

These two parameters must be multiples of the batch interval of the source DStream.

i Example: word count and window

Problem specification

- Input: a stream of sentences
- Split the input stream in batches of 10 seconds
- Define windows with the following characteristics
 - Window length: 20 seconds (i.e., 2 batches)
 - Sliding interval: 10 seconds (i.e., 1 batch)
- Print on the standard output, for each window, the occurrences of each word appearing in the window (i.e., execute the word count problem for each window)

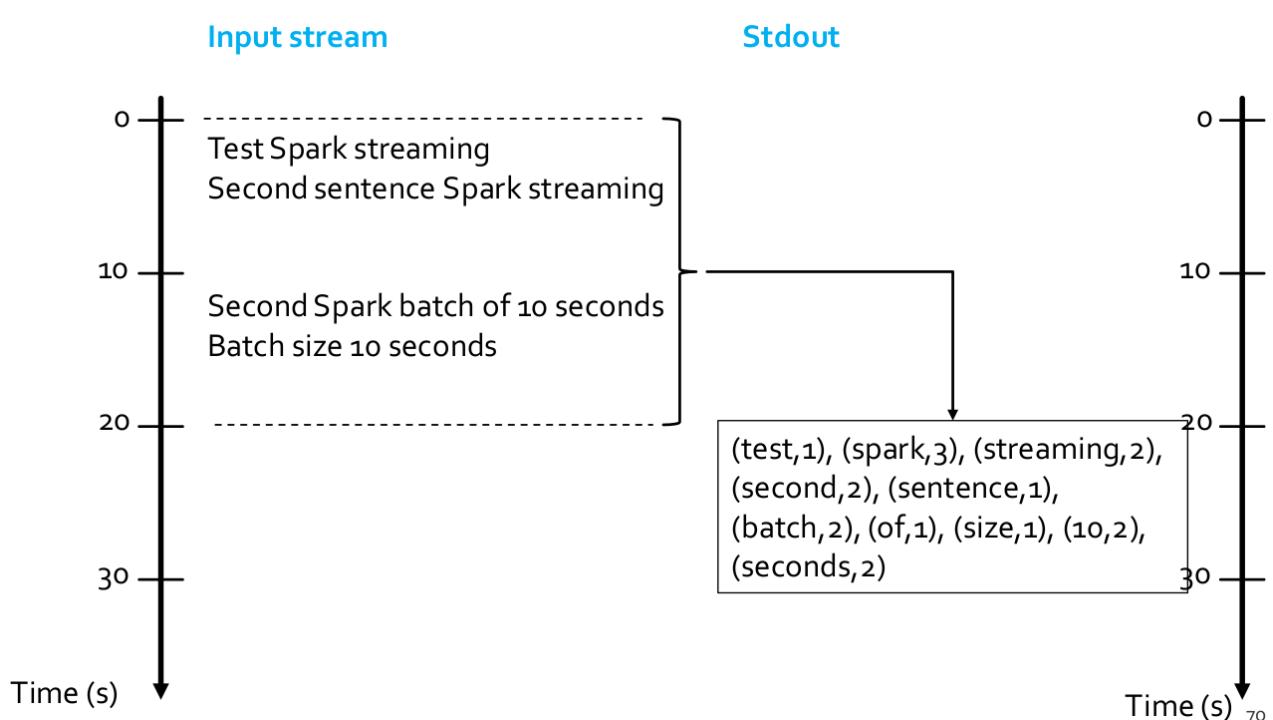


Figure 26.9: Step one

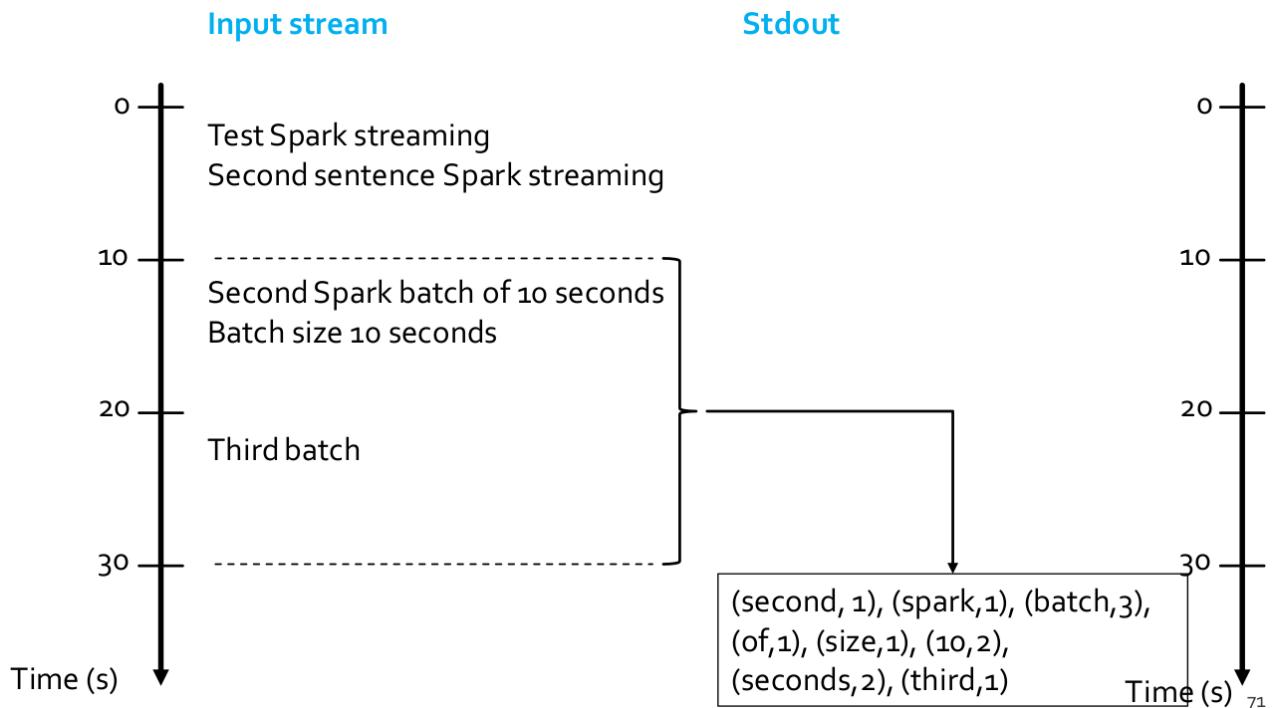


Figure 26.10: Step two

Word count and windows

Basic window transformations

Window transformation	Effect
<code>window(windowLength, slideInterval)</code>	It returns a new DStream which is computed based on windowed batches of the source DStream.
<code>countByWindow(windowLength, slideInterval)</code>	It returns a new single-element stream containing the number of elements of each window. The returned object is a Dstream of Long objects. However, it contains only one value for each window (the number of elements of the last analyzed window).
<code>reduceByKey(reduceFunc, invReduceFunc, windowDuration, slideDuration)</code>	It returns a new single-element stream, created by aggregating elements in the stream over a sliding interval using <code>func</code> . The function must be associative and commutative so that it can be computed correctly in parallel. If <code>invReduceFunc</code> is not <code>None</code> , the reduction is done incrementally using the old window's reduced value.
<code>countByValueAndWindow(windowDuration, slideDuration)</code>	<code>WindowDuration</code> called on a DStream of elements of type K , it returns a new DStream of (K, Long) pairs where the value of each key K is its frequency in each window of the source DStream.
<code>reduceByKeyAndWindow(func, invFunc, windowDuration, slideDuration=None, numPartitions=None)</code>	When called on a DStream of (K, V) pairs, it returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function func over batches in a sliding window. The window duration (length) is specified as a parameter of this invocation <code>(windowDuration)</code> . Notice that, if <code>slideDuration</code> is <code>None</code> , the <code>batchDuration</code> of the <code>StreamingContext</code> object is used (i.e., 1 batch sliding window); if <code>invFunc</code> is provided (is not <code>None</code>), the reduction is done incrementally using the old window's reduced values (i.e., <code>invFunc</code> is used to apply an inverse reduce operation by considering the old values that left the window, e.g., subtracting old counts).

Checkpoint

A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.). For this to be possible, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures, and this result is achieved by means of checkpoints, which are operations that store the data and metadata needed to restart the computation if failures happen. Checkpointing is necessary even for some window transformations and stateful transformations.

Checkpointing is enabled by using the `checkpoint(String folder)` method of `SparkStreamingContext`: the parameter is the folder that is used to store temporary data. This is similar as for processing graphs with GraphFrames library, however, with GraphFrames, the checkpoint was the one of `SparkContext`.

Example: word count and windows

Problem specification

- Input: a stream of sentences retrieved from localhost:9999
- Split the input stream in batches of 5 seconds
- Define windows with the following characteristics
 - Window length: 15 seconds (i.e., 3 batches)
 - Sliding interval: 5 seconds (i.e., 1 batch)
- Print on the standard output, for each window, the occurrences of each word appearing in the window (i.e., execute the word count problem for each window)
- Store the results also in an HDFS folder

First solution

```

1  from pyspark.streaming import StreamingContext
2
3  ## Set prefix of the output folders
4  outputPathPrefix="resSparkStreamingExamples"
5
6  #Create a configuration object and#set the name of the applicationconf
7  SparkConf().setAppName("Streaming word count")
8
9  ## Create a Spark Context object
10 sc = SparkContext(conf=conf)
11
12 ## Create a Spark Streaming Context object
13 ssc = StreamingContext(sc, 5)
14
15 ## Set the checkpoint folder (it is needed by some window transformations)
16 ssc.checkpoint("checkpointfolder")
17
18 ## Create a (Receiver) DStream that will connect to localhost:9999
19 lines = ssc.socketTextStream("localhost", 9999)
20
21 ## Apply a chain of transformations to perform the word count task
22 ## The returned RDDs are DStream RDDs
23 words = lines.flatMap(lambda line: line.split(" "))
24 wordsOnes = words.map(lambda word: (word, 1))
25
26 ## reduceByKeyAndWindow is used instead of reduceByKey
27 ## The duration of the window is also specified
28 wordsCounts = wordsOnes \
29     .reduceByKeyAndWindow(lambda v1, v2: v1+v2, None, 15)
30
31 ## Print the num. of occurrences of each word of the current window
32 ## (only 10 of them)
33 wordsCounts.print()
34
35 ## Store the output of the computation in the folders with prefix
36 ## outputPathPrefix
37 wordsCounts.saveAsTextFiles(outputPathPrefix, "")
38
39 #Start the computation
40 ssc.start()
41 ssc.awaitTermination ()

```

Second solution

```

1  from pyspark.streaming import StreamingContext
2
3  ## Set prefix of the output folders
4  outputPathPrefix="resSparkStreamingExamples"
5
6  ## Create a configuration object and
7  ## set the name of the applicationconf
8  SparkConf().setAppName("Streaming word count")
9
10 ## Create a Spark Context object
11 sc = SparkContext(conf=conf)
12
13 ## Create a Spark Streaming Context object
14 ssc = StreamingContext(sc, 5)
15
16 ## Set the checkpoint folder (it is needed by some window transformations)
17 ssc.checkpoint("checkpointfolder")
18
19 ## Create a (Receiver) DStream that will connect to localhost:9999
20 lines = ssc.socketTextStream("localhost", 9999)
21
22 ## Apply a chain of transformations to perform the word count task
23 ## The returned RDDs are DStream RDDs
24 words = lines.flatMap(lambda line: line.split(" "))
25 wordsOnes = words.map(lambda word: (word, 1))
26
27 ## reduceByKeyAndWindow is used instead of reduceByKey
28 ## The duration of the window is also specified
29 wordsCounts = wordsOnes \
30     .reduceByKeyAndWindow(
31         lambda v1, v2: v1+v2,
32         lambda vnow,
33         vold: vnow-vold, 15
34     )
35
36 ## Print the num. of occurrences of each word of the current window
37 ## (only 10 of them)
38 wordsCounts.pprint()
39
40 ## Store the output of the computation in the folders with prefix
41 ## outputPathPrefix
42 wordsCounts.saveAsTextFiles(outputPathPrefix, "")
43
44 #Start the computation
45 ssc.start()
46
47 ## Run this application for 90 seconds
48 ssc.awaitTerminationOrTimeout(90)
49 ssc.stop(stopSparkContext=False)

```

- ① In this solution the inverse function is also specified in order to compute the result incrementally

26.5 Stateful computation

`updateStateByKey` transformation

The `updateStateByKey` transformation allows maintaining a state for each key. The value of the state of each key is continuously updated every time a new batch is analyzed.

The use of `updateStateByKey` is based on two steps

- Define the state: the data type of the state associated with the keys can be an arbitrary data type
- Define the state update function: specify with a function how to update the state of a key using the previous state and the new values from an input stream associated with that key

In every batch, Spark will apply the state update function for all existing keys. For each key, the update function is used to update the value associated with a key by combining the former value and the new values associated with that key; in other words, for each key, the call method of the function is invoked on the list of new values and the former state value and returns the new aggregated value for the considered key.

i Example: word count and `updateStateByKey` transformation

By using the `updateStateByKey`, the application can continuously update the number of occurrences of each word. The number of occurrences stored in the DStream returned by this transformation is computed over the union of all the batches (from the first one to the current one). For efficiency reasons, the new value for each key is computed by combining the last value for that key with the values of the current batch for the same key.

Problem specification:

- Input: a stream of sentences retrieved from localhost:9999
- Split the input stream in batches of 5 seconds
- Print on the standard output, every 5 seconds, the occurrences of each word appearing in the stream (from time 0 to the current time) (i.e., execute the word count problem from the beginning of the stream to current time)
- Store the results also in an HDFS folder

```

1  from pyspark.streaming import StreamingContext
2
3  ## Set prefix of the output folders
4  outputPathPrefix="resSparkStreamingExamples"
5
6  #Create a configuration object and#set the name of the applicationconf
7  SparkConf().setAppName("Streaming word count")
8
9  ## Create a Spark Context object
10 sc = SparkContext(conf=conf)
11
12 ## Create a Spark Streaming Context object
13 ssc = StreamingContext(sc, 5)
14
15 ## Set the checkpoint folder (it is needed by some window transformations)
16 ssc.checkpoint("checkpointfolder")
17
18 ## Create a (Receiver) DStream that will connect to localhost:9999
19 lines = ssc.socketTextStream("localhost", 9999)
20
21 ## Apply a chain of transformations to perform the word count task
22 ## The returned RDDs are DStream RDDs
23 words = lines.flatMap(lambda line: line.split(" "))
24 wordsOnes = words.map(lambda word: (word, 1))
25
26 ## Define the function that is used to update the state of a key at a time
27 def updateFunction(newValues, currentCount): ①
28     if currentCount is None:
29         currentCount = 0
30
31     ## Sum the new values to the previous state for the current key
32     return sum(newValues, currentCount) ②
33
34 ## DStream made of cumulative counts for each key that get updated
35 ## in every batch
36 totalWordsCounts = wordsOnes.updateStateByKey(updateFunction) ③
37
38 ## Print the num. of occurrences of each word of the current window
39 ## (only 10 of them)
40 totalWordsCounts.print()
41
42 ## Store the output of the computation in the folders with prefix
43 ## outputPathPrefix
44 totalWordsCounts.saveAsTextFiles(outputPathPrefix, "")
45
46 ## Start the computation
47 ssc.start()
48
49 ## Run this application for 90 seconds
50 ssc.awaitTerminationOrTimeout(90)      370
51 ssc.stop(stopSparkContext=False)

```

- ① `currentCount`: current state/value for the current key | `newValues`: list of new integer values for the current key
- ② `sum(newValues, currentCount)`: Combine current state and new values
- ③ `updateFunction`: this function is invoked one time for each key

26.6 Transform transformation

Some types of transformations are not available for DStreams (e.g., `sortBy()`, `sortByKey()`, `distinct()`), moreover, sometimes it is needed to combine DStreams and RDDs. For example, the functionality of joining every batch in a data stream with another dataset (a standard RDD) is not directly exposed in the DStream API. The `transform()` transformation can be used in these situations.

Transformation	Effect
<code>transform(func)</code>	It is a specific transformation of DStreams that returns a new DStream by applying an RDD-to-RDD function to every RDD of the source Dstream. This can be used to apply arbitrary RDD operations on the DStream

i Example

Problem specification

- Input: a stream of sentences retrieved from localhost:9999
- Split the input stream in batches of 5 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch. The pairs must be returned/displayed sorted by decreasing number of occurrences (per batch)
- Store the results also in an HDFS folder

```
1  from pyspark.streaming import StreamingContext
2
3  ## Set prefix of the output folders
4  outputPathPrefix="resSparkStreamingExamples"
5
6  #Create a configuration object and#set the name of the applicationconf
7  SparkConf().setAppName("Streaming word count")
8
9  ## Create a Spark Context object
10 sc = SparkContext(conf=conf)
11
12 ## Create a Spark Streaming Context object
13 ssc = StreamingContext(sc, 5)
14
15 ## Create a (Receiver) DStream that will connect to localhost:9999
16 lines = ssc.socketTextStream("localhost", 9999)
17
18 ## Apply a chain of transformations to perform the word count task
19 ## The returned RDDs are DStream RDDs
20 words = lines.flatMap(lambda line: line.split(" "))
21 wordsOnes = words.map(lambda word: (word, 1))
22 wordsCounts = wordsOnes.reduceByKey(lambda v1, v2: v1+v2)
23
24 ## Sort the content/the pairs by decreasing value (# of occurrences)
25 wordsCountsSortByKey = wordsCounts \
26     .transform(lambda batchRDD: batchRDD.sortBy(lambda pair: -1*pair[1]))
27
28 ## Print the result on the standard output
29 wordsCountsSortByKey.print()
30
31 ## Store the result in HDFS
32 wordsCountsSortByKey.saveAsTextFiles(outputPathPrefix, "")
33
34 #Start the computation
35 ssc.start()
36
37 ## Run this application for 90 seconds
38 ssc.awaitTerminationOrTimeout(90)
39 ssc.stop(stopSparkContext=False)
```

27 Spark structured streaming

27.1 What is Spark structured streaming?

Structured Streaming is a scalable and fault-tolerant stream processing engine that is built on the Spark SQL engine, and input data are represented by means of (streaming) DataFrames. Structured Streaming uses the existing Spark SQL APIs to query data streams (the same methods used for analyzing static DataFrames).

A set of specific methods are used to define

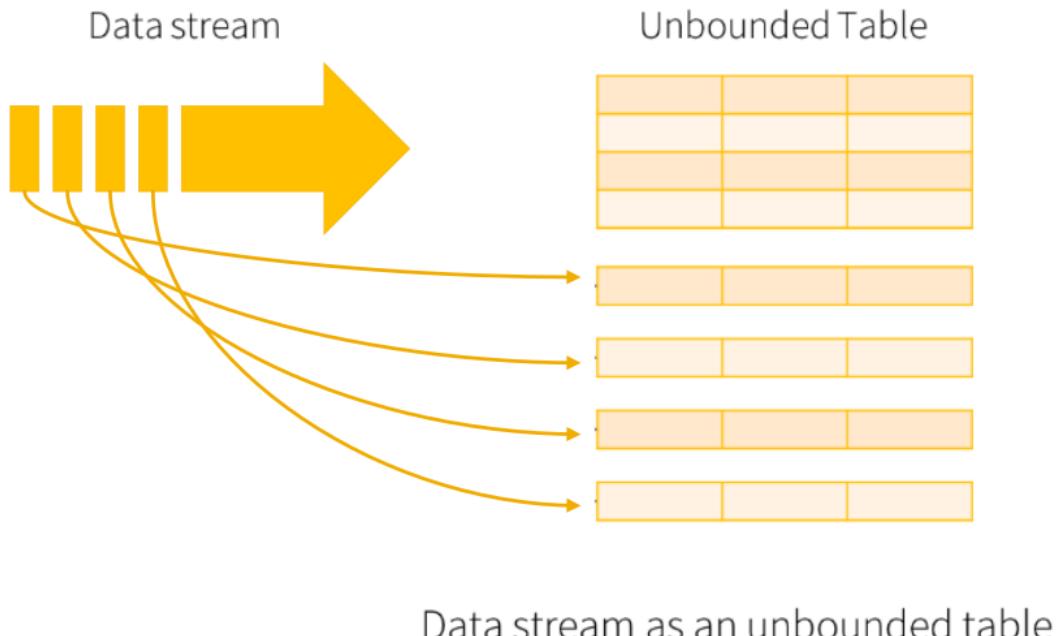
- Input and output streams
- Windows

Input data model

Each input data stream is modeled as a table that is being continuously appended: every time new data arrive they are appended at the end of the table (i.e., each data stream is considered an unbounded input table).

New input data in the stream are new rows appended to an unbounded table

Figure 27.1: Input stream



Queries

The expressed queries are incremental queries that are run incrementally on the unbounded input tables.

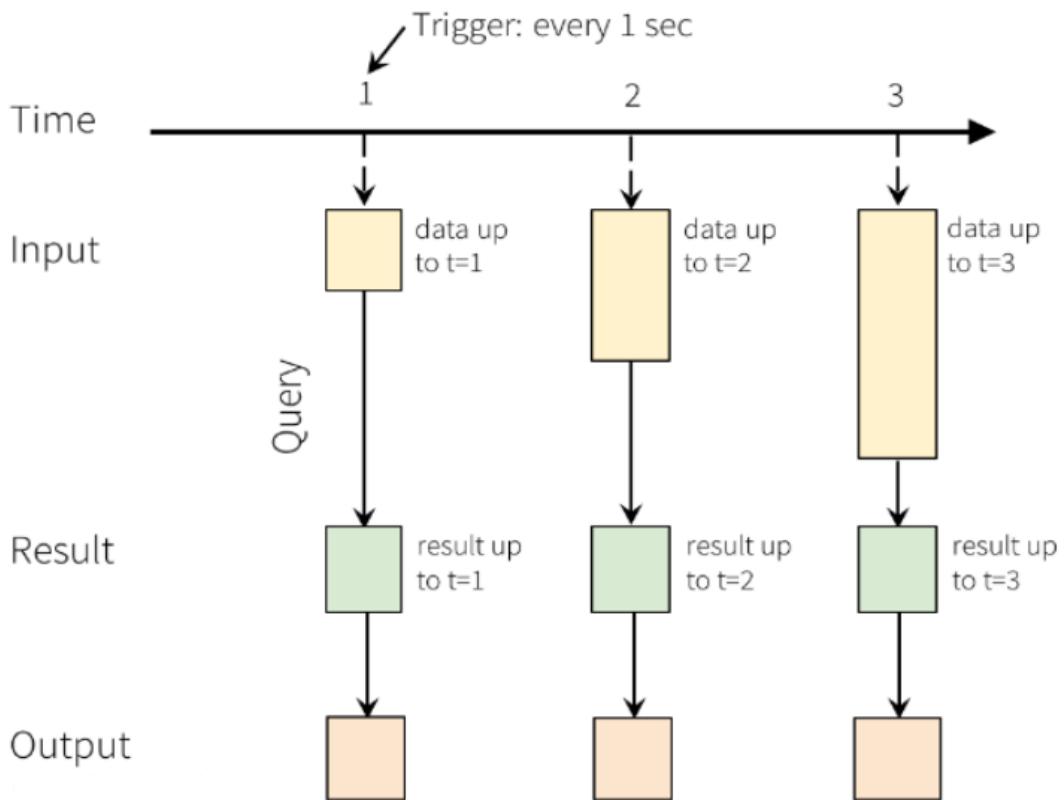
- The arrival of new data triggers the execution of the incremental queries
- The result of a query at a specific timestamp is the one obtained by running the query on all the data arrived until that timestamp (i.e., stateful queries are executed).
- Aggregation queries combine new data with the previous results to optimize the computation of the new results.

The queries can be executed

- As micro-batch queries with a fixed batch interval: this is the standard behavior, with exactly-once fault-tolerance guarantees
- As continuous queries: this is experimental behavior, with at-least-once fault-tolerance guarantees

In this example the (micro-batch) query is executed every 1 second

Figure 27.2: Query execution

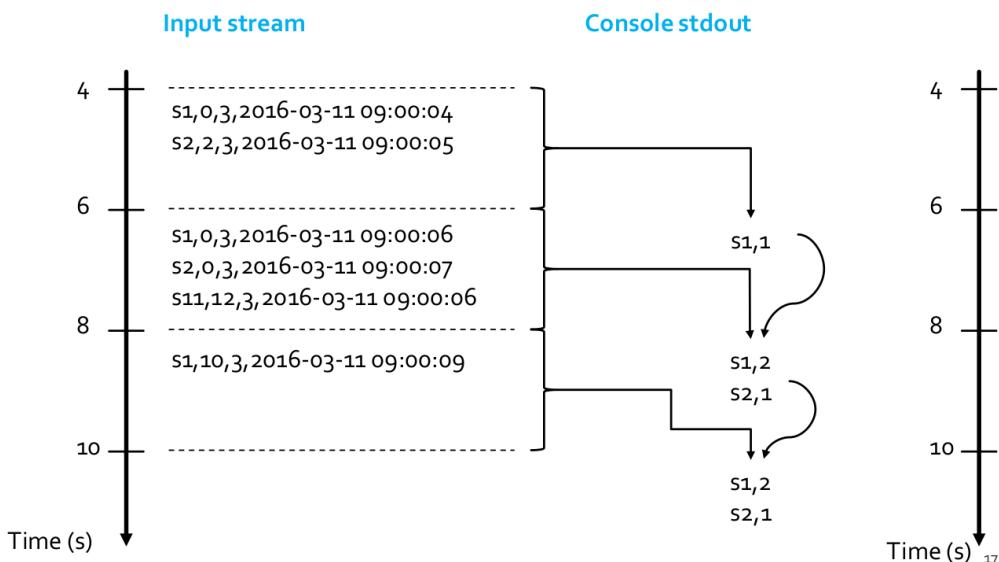


Note that every time the query is executed, all data received so far are considered.

Example

- Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp #– Each input reading has the format `stationId,# free slots,#used slots,timestamp`
- For each stationId, print on the standard output the total number of received input readings with a number of free slots equal to 0
 - Print the requested information when new data are received by using the micro-batch processing mode
 - Suppose the batch-duration is set to 2 seconds

Figure 27.3: Example execution



27.2 Key concepts

- Input sources
- Transformations
- Outputs
 - External destinations/sinks
 - Output Modes
- Query run/execution
- Triggers

Input sources

- File source
 - Reads files written in a directory as a stream of data
 - Each line of the input file is an input record
 - Supported file formats are text, csv, json, orc, parquet, ...
- Kafka source
 - Reads data from Kafka
 - Each Kafka message is one input record
- Socket source (for debugging purposes)
 - Reads UTF8 text data from a socket connection
 - This type of source does not provide end-to-end fault-tolerance guarantees
- Rate source (for debugging purposes)
 - Generates data at the specified number of rows per second
 - Each generated row contains a timestamp and value of type long

The `readStream` property of the `SparkSession` class is used to create `DataStreamReader`s: the methods `format()` and `option()` of the `DataStreamReader` class are used to specify the input streams (e.g., type, location). The method `load()` of the `DataStreamReader` class is used to return `DataFrames` associated with the input data streams.

Note

In this example the (streaming) `DataFrame` `recordsDF` is created and associated with the input stream of type socket

- Address: localhost
- Input port: 9999

```

1 recordsDF = spark.readStream \
2     .format("socket") \
3     .option("host", "localhost") \
4     .option("port", 9999) \
5     .load()

```

Transformations

Transformations are the same of `DataFrames`, however there are restrictions on some types of queries/transformations that cannot be executed incrementally.

Unsupported operations:

- Multiple streaming aggregations (i.e. a chain of aggregations on a streaming `DataFrame`)

- Limit and take first N rows
- Distinct operations
- Sorting operations are supported on streaming DataFrames only after an aggregation and in complete output mode
- Few types of outer joins on streaming DataFrames are not supported

Outputs

- Sinks: they are instances of the class DataStreamWriter and are used to specify the external destinations and store the results in the external destinations
- File sink: it stores the output to a directory; supported file formats are text, csv, json, orc, parquet, ...
- Kafka sink: it stores the output to one or more topics in Kafka
- Foreach sink: it runs arbitrary computation on the output records
- Console sink (for debugging purposes): it prints the computed output to the console every time a new batch of records has been analyzed; this should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory after each computation
- Memory sink (for debugging purposes): the output is stored in memory as an in-memory table; this should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory

Output mods

We must define how we want Spark to write output data in the external destinations. The supported output modes depend on the query type, and the possible output mods are the following

Append

This is the default mode. Only the new rows added to the computed result since the last trigger (computation) will be outputted. This mode is supported for only those queries where rows added to the result is never going to change: this mode guarantees that each row will be output only once. So, the supported queries are only select, filter, map, flatMap, filter, join, etc.

Complete

The whole computed result will be outputted to the sink after every trigger (computation). This mode is supported for aggregation queries.

Update

Only the rows in the computed result that were updated since the last trigger (computation) will be outputted.

The complete list of supported output modes for each query type is available in the [Apache Spark documentation](#).

Code

The `writeStream` property of the `SparkSession` class is used to create `DataStreamWriter`s. The methods `outputMode()`, `format()`, and `option()` of the `DataStreamWriter` class are used to specify the output destination (data format, location, output mode, etc.).

Note

Example The `DataStreamWriter` “`streamWriterRes`” is created and associated with the console. The output mode is set to append.

```
1 streamWriterRes = stationIdTimestampDF \
2     .writeStream \
3     .outputMode("append") \
4     .format("console")
```

Query run/execution

To start executing the defined queries/structured streaming applications you must explicitly invoke the `start()` action on the defined sinks (`DataStreamWriter` objects associated with the external destinations in which the results will be stored). It is possible to start several queries in the same application, and structured streaming queries run forever (they must be explicitly stop/kill).

Triggers

For each Spark structured streaming query it is possible to specify when new input data must be processed, and whether the query is going to be executed as a micro-batch query with a fixed batch interval or as a continuous processing query (experimental). The trigger type for each query is specified by means of the `trigger()` method of the `DataStreamWriter` class.

Trigger types

No trigger type is explicitly specified, by default the query will be executed in micro-batch mode, where each micro-batch is generated and processed as soon as the previous micro-batch has been processed.

Fixed interval micro-batches

The query will be executed in micro-batch mode. Micro-batches will be processed at the user-specified intervals: the parameter `processingTime` of the trigger `method()` is used to specify the micro-batch size, and, if the previous micro-batch completes within its interval, then the engine will wait until the interval is over before processing the next micro-batch; if the previous micro-batch takes longer than the interval to complete (i.e. if an interval boundary is missed), then the next micro-batch will start as soon as the previous one completes.

One-time micro-batch

The query will be executed in micro-batch mode, but the query will be executed only one time on one single micro-batch containing all the available data of the input stream; after the single execution the query stops on its own. This trigger type is useful when the goal is to periodically spin up a cluster, process everything that is available since the last period, and then shutdown the cluster. In some case, this may lead to significant cost savings.

Continuous with fixed checkpoint interval (experimental)

The query will be executed in the new low-latency, continuous processing mode. It offers at-least-once fault-tolerance guarantees.

Spark structured streaming examples

Example 1

- Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp #– Each input reading has the format: “stationId”, “# free slots”, “#used slots”, “timestamp”
- Output
 - For each input reading with a number of free slots equal to 0 print on the standard output the value of stationId and timestamp
 - Use the standard micro-batch processing mode

```

1  from pyspark.sql.types import *
2  from pyspark.sql.functions import split
3
4  ## Create a "receiver" DataFrame that will connect to localhost:9999
5  recordsDF = spark.readStream \
6      .format("socket") \
7      .option("host", "localhost") \
8      .option("port", 9999) \
9      .load()
10
11 ## The input records are characterized by one single column called value
12 ## of type string
13 ## Example of an input record: s1,0,3,2016-03-11 09:00:04
14 ## Define four more columns by splitting the input column value
15 ## New columns:
16 ## - stationId
17 ## - freeslots
18 ## - usedslots

```

```

19 ## - timestamp
20 readingsDF = recordsDF \
21   .withColumn("stationId", split(recordsDF.value, ',')[0].cast("string")) \
22   .withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer")) \
23   .withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer")) \
24   .withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp")) ①
25
26 ## Filter data
27 ## Use the standard filter transformation
28 fullReadingsDF = readingsDF.filter("freeslots=0") ②
29
30 ## Select stationid and timestamp
31 ## Use the standard select transformation
32 stationIdTimestampDF = fullReadingsDF.select("stationId", "timestamp")
33
34 ## The result of the structured streaming query will be stored/printed on
35 ## the console "sink".
36 ## append output mode
37 queryFilterStreamWriter = stationIdTimestampDF \
38   .writeStream \
39   .outputMode("append") \
40   .format("console")
41
42 ## Start the execution of the query (it will be executed until it is explicitly stopped)
43 queryFilter = queryFilterStreamWriter.start()

```

- ① `withColumn()` is used to add new columns (it is a standard DataFrame method). It returns a DataFrame with the same columns of the input DataFrame and the new defined column. For each new column it is possible to specify name (e.g. “stationId”) and the SQL function that is used to define its value in each record. The `cast()` method is used to specify the data type of each defined column.

- ② `filter` and `select` are standard DataFrame transformations

Example 2

- Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of bike sharing system in a specific timestamp #
 - Each input reading has the format: “stationId”, “# free slots”, “#used slots”, “timestamp”
- Output
 - For each stationId, print on the standard output the total number of received input readings with a number of free slots equal to 0
 - Print the requested information when new data are received by using the standard micro-batch processing mode

```

1  from pyspark.sql.types import *
2  from pyspark.sql.functions import split
3
4  ## Create a "receiver" DataFrame that will connect to localhost:9999
5  recordsDF = spark.readStream \
6      .format("socket") \
7      .option("host", "localhost") \
8      .option("port", 9999) \
9      .load()
10
11 ## The input records are characterized by one single column called value
12 ## of type string
13 ## Example of an input record: s1,0,3,2016-03-11 09:00:04
14 ## Define four more columns by splitting the input column value
15 ## New columns:
16 ## - stationId
17 ## - freeslots
18 ## - usedslots
19 ## - timestamp
20 readingsDF = recordsDF \
21     .withColumn("stationId", split(recordsDF.value, ',')[0].cast("string")) \
22     .withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer")) \
23     .withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer")) \
24     .withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))
25
26 ## Filter data
27 ## Use the standard filter transformation
28 fullReadingsDF = readingsDF.filter("freeslots=0")
29
30 ## Count the number of readings with a number of free slots equal to 0
31 ## for each stationId
32 ## The standard groupBy method is used
33 countsDF = fullReadingsDF \
34     .groupBy("stationId") \
35     .agg({"*": "count"})(1)
36
37 ## The result of the structured streaming query will be stored/printed on
38 ## the console "sink"
39 ## complete output mode
40 ## (append mode cannot be used for aggregation queries)
41 queryCountStreamWriter = countsDF \
42     .writeStream \
43     .outputMode("complete") \
44     .format("console")
45
46 ## Start the execution of the query (it will be executed until it is explicitly stopped)

```

⁴⁷ `queryCount = queryCountStreamWriter.start()`

- ① `groupBy` and `agg` are standard DataFrame transformations

27.3 Event time and window operations

Input streaming records are usually characterized by a time information: it is usually called event-time, and it is the time when the data was generated. For many applications, you want to operate by taking into consideration the event-time and windows containing data associated with the same event-time range.

Example

Compute the number of events generated by each monitored IoT device every minute based on the event-time. For each window associated with one distinct minute, consider only the data with an event-time inside that minute/window and compute the number of events for each IoT device: one computation for each minute/window. You want to use the time when the data was generated (i.e., the event-time) rather than the time Spark receives them.

Spark allows defining windows based on the time-event input column, and then apply aggregation functions over each window.

For each structured streaming query on which you want to apply a window computation you must specify

- the name of the time-event column in the input (streaming) DataFrame
- the characteristics of the (sliding) windows
 - `windowDuration`
 - `slideDuration`

Do not set it if you want non-overlapped windows, (i.e., if you want a `slideDuration` equal to `windowDuration`). You can set different window characteristics for each query of your application.

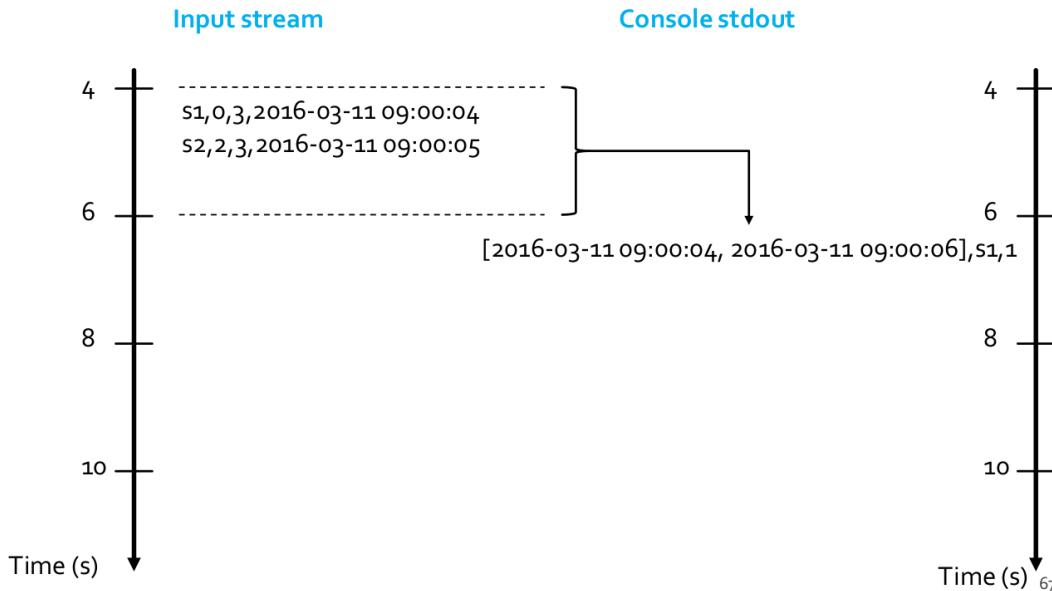
The `window(timeColumn, windowDuration, slideDuration=None)` function is used inside the standard `groupBy()` one to specify the characteristics of the windows. Notice that windows can be used only with queries that are applying aggregation functions.

Event time and window operations: example 1

- Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp #– Each input reading has the format: “stationId”, “# free slots”, “#used slots”, “timestamp”
 - timestamp is the event-time column
- Output

- For each stationId, print on the standard output the total number of received input readings with a number of free slots equal to 0 in each window
- The query is executed for each window
- Set `windowDuration` to 2 seconds and no `slideDuration` (i.e., non-overlapped windows)

Figure 27.4: Step 1



The returned result has a column called `window`. It contains the time slot associated with the window [from timestamp, to timestamp)

Figure 27.5: Step 2

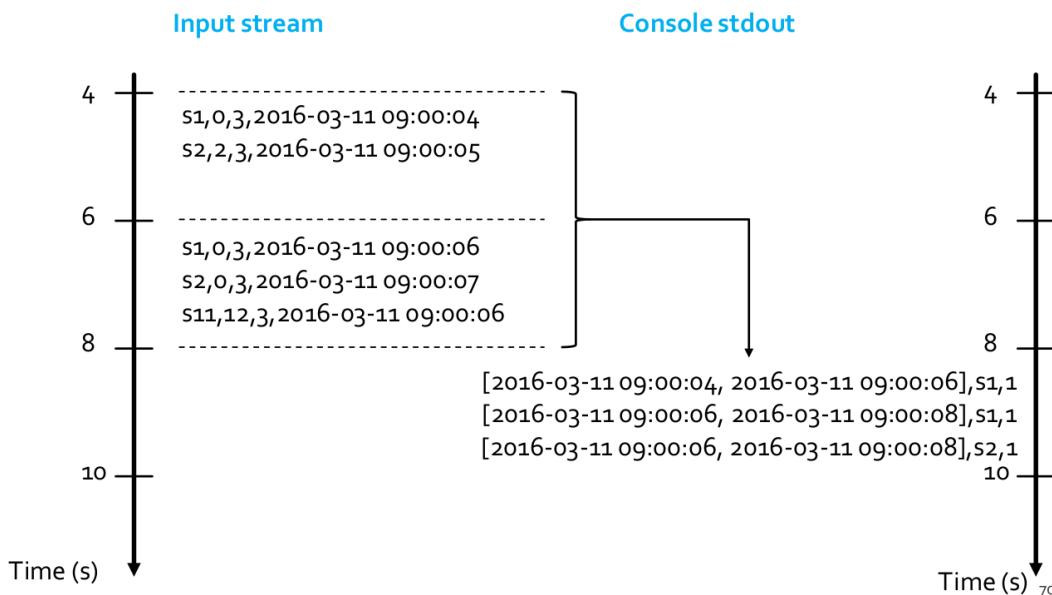
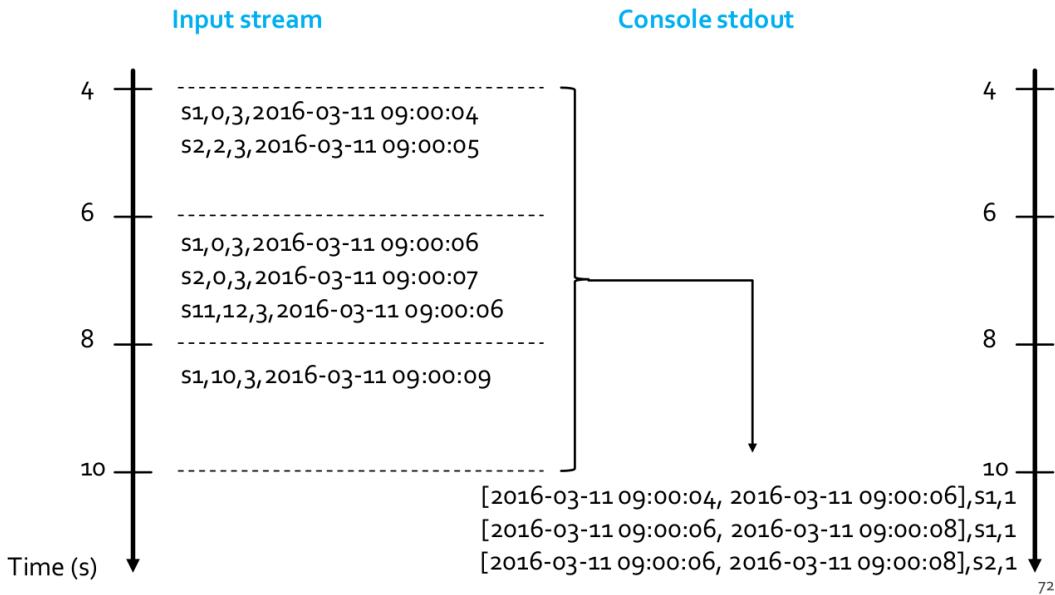


Figure 27.6: Step 3



```

1  from pyspark.sql.types import *
2  from pyspark.sql.functions import split
3  from pyspark.sql.functions import window
4
5  ## Create a "receiver" DataFrame that will connect to localhost:9999
6  recordsDF = spark.readStream \
7      .format("socket") \
8      .option("host", "localhost") \
9      .option("port", 9999) \
10     .load()
11
12 ## The input records are characterized by one single column called value
13 ## of type string
14 ## Example of an input record: s1,0,3,2016-03-11 09:00:04
15 ## Define four more columns by splitting the input column value
16 ## New columns:
17 ## - stationId
18 ## - freeslots
19 ## - usedslots
20 ## - timestamp
21 readingsDF = recordsDF \
22     .withColumn("stationId", split(recordsDF.value, ',')[0].cast("string")) \
23     .withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer")) \
24     .withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer")) \
25     .withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))
26
27 ## Filter data

```

```

28 ## Use the standard filter transformation
29 fullReadingsDF = readingsDF.filter("freeslots=0")
30
31 ## Count the number of readings with a number of free slots equal to 0
32 ## for each stationId in each window.
33 ## windowDuration = 2 seconds
34 ## no overlapping windows
35 countsDF = fullReadingsDF \
36     .groupBy(window(fullReadingsDF.timestamp, "2 seconds"), "stationId") \
37     .agg({"*": "count"}) \
38     .sort("window")
39
40 ## The result of the structured streaming query will be stored/printed on
41 ## the console "sink"
42 ## complete output mode
43 ## (append mode cannot be used for aggregation queries)
44 queryCountWindowStreamWriter = countsDF \
45     .writeStream \
46     .outputMode("complete") \
47     .format("console") \
48     .option("truncate", "false")
49
50 ## Start the execution of the query (it will be executed until it is explicitly stopped)
51 queryCountWindow = queryCountWindowStreamWriter.start()

```

Late data

Sparks handles data that have arrived later than expected based on its event-time; these are called late data. Spark has full control over updating old aggregates when there are late data: every time new data are processed the result is computed by combining old aggregate values and the new data by considering the event-time column instead of the time Spark receives the data.

Late data example

- Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp #– Each input reading has the format: “stationId”, “# free slots”, “#used slots”, “timestamp”
 - timestamp is the event-time column
- Output
 - For each stationId, print on the standard output the total number of received input readings with a number of free slots equal to 0 in each window
 - The query is executed for each window

- Set `windowDuration` to 2 seconds and no `slideDuration` (i.e., non-overlapped windows)

Figure 27.7: Step 1

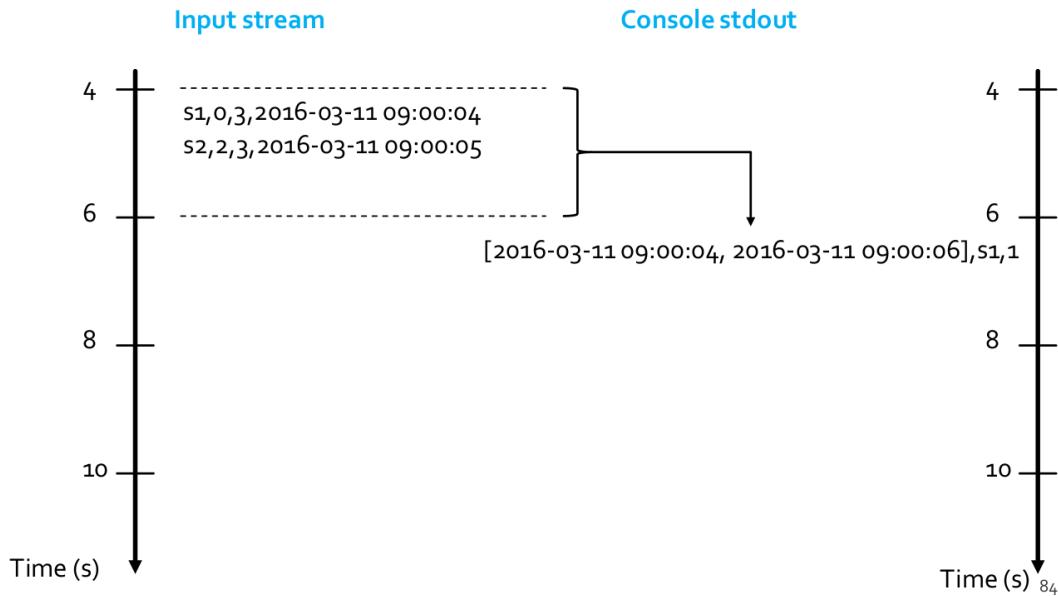


Figure 27.8: Step 2

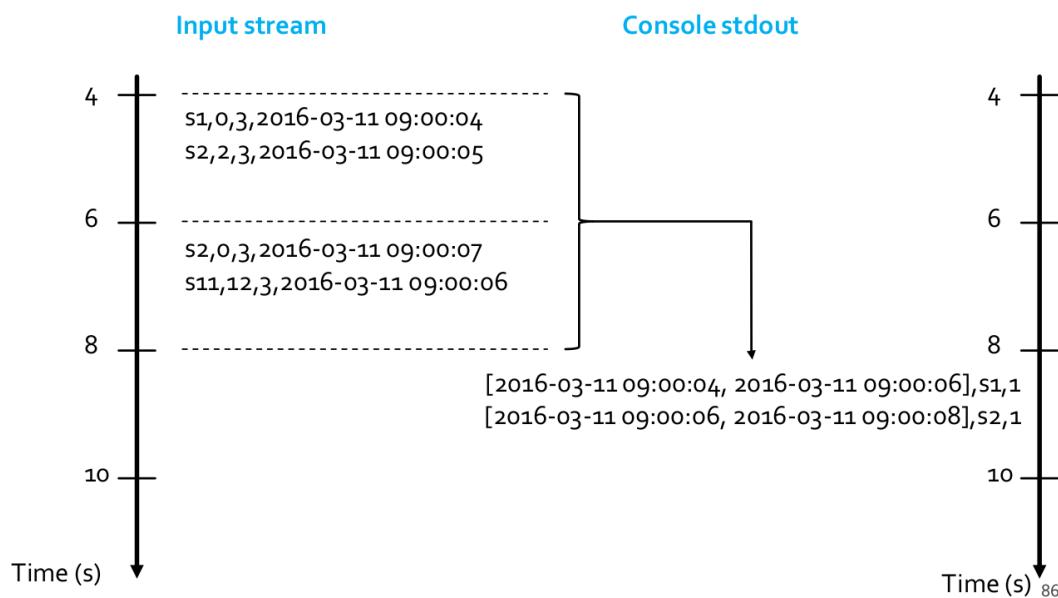
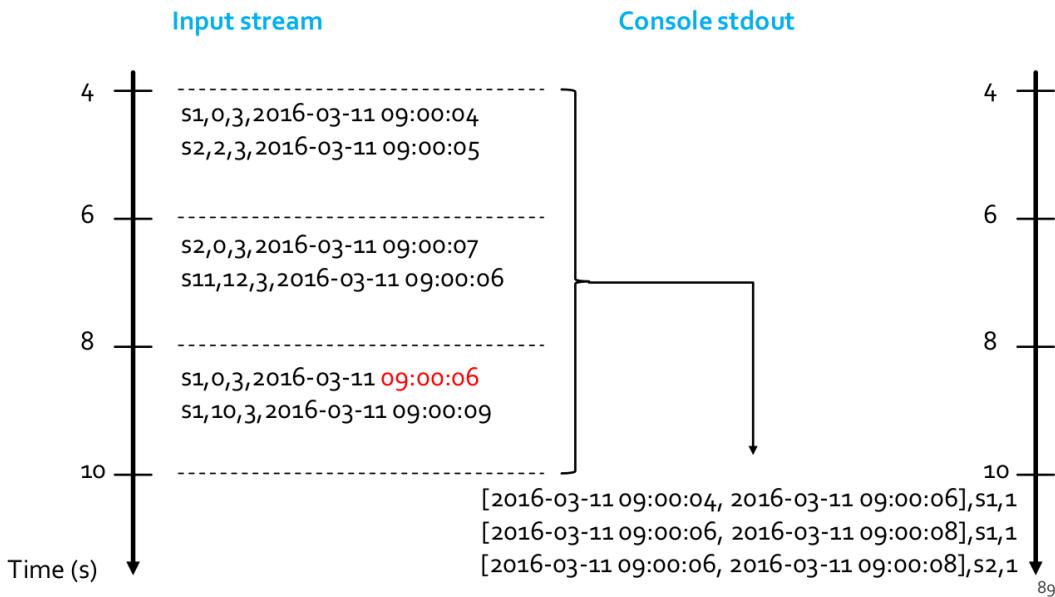


Figure 27.9: Step 3



Notice that late data that was generated at **2016-03-11 09:00:06**, but arrived at **2016-03-11 09:00:08**: the result consider also late data and assign them to the right window by considering the event-time information.

The code is the same of the previous example (Event time and window operations: example 1): late data are automatically handled by Spark.

Event time and window operations: example 2

- Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp #– Each input reading has the format: “stationId”, “# free slots”, “#used slots”, “timestamp”
 - timestamp is the event-time column
- Output
 - For each window, print on the standard output the total number of received input readings with a number of free slots equal to 0
 - The query is executed for each window
 - Set windowDuration to 2 seconds and no slideDuration (i.e., non-overlapped windows)

Figure 27.10: Step 1

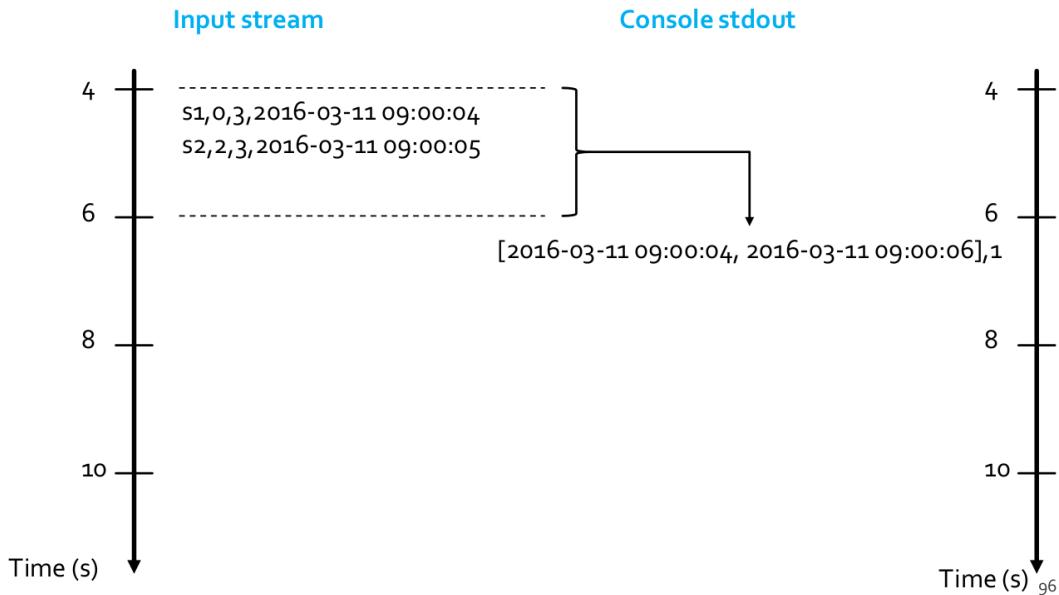


Figure 27.11: Step 2

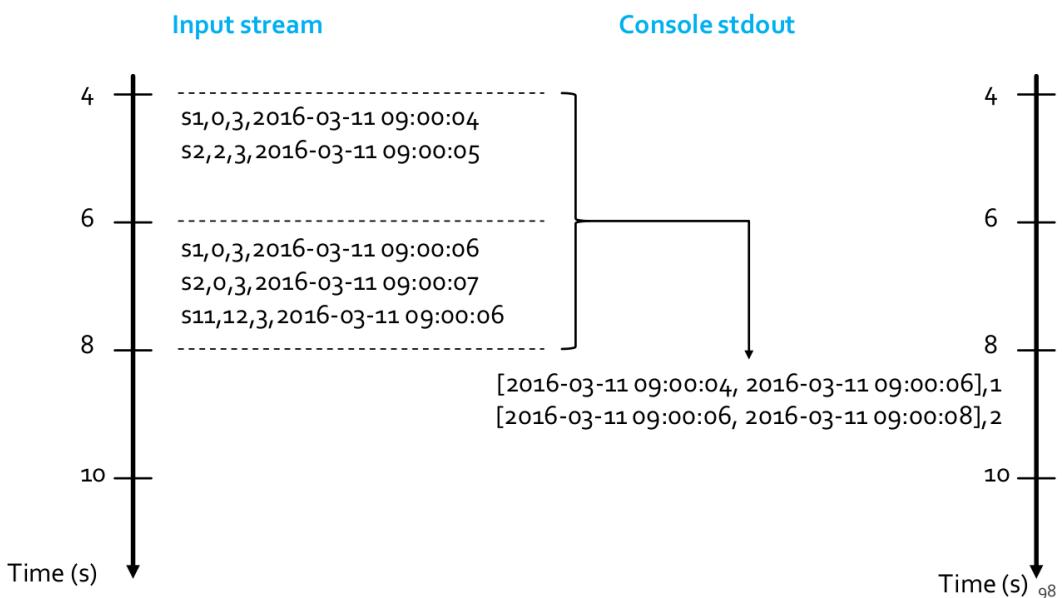
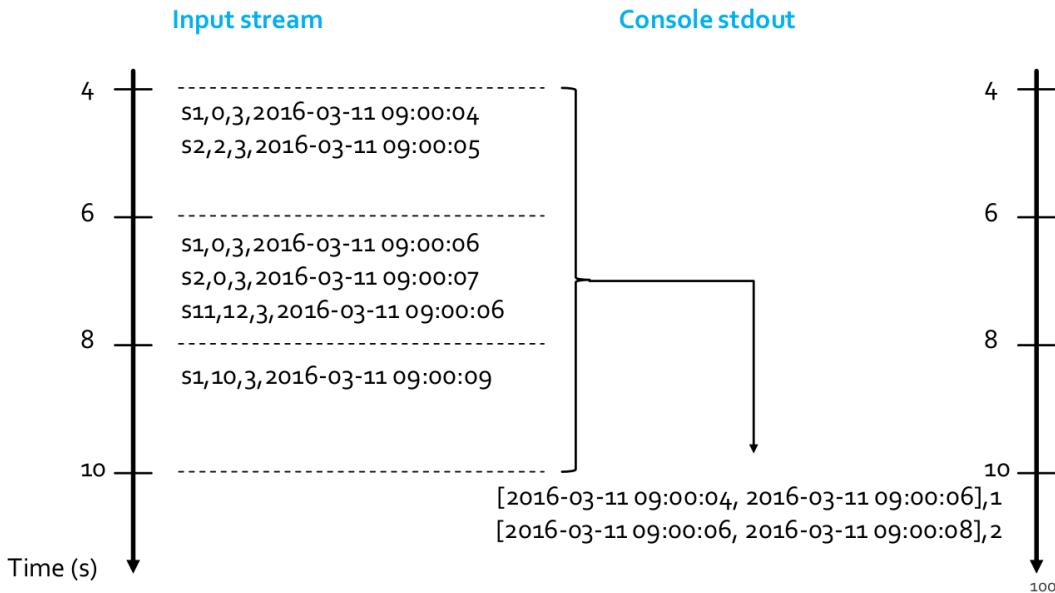


Figure 27.12: Step 3



```

1  from pyspark.sql.types import *
2  from pyspark.sql.functions import split
3  from pyspark.sql.functions import window
4
5  ## Create a "receiver" DataFrame that will connect to localhost:9999
6  recordsDF = spark.readStream \
7      .format("socket") \
8      .option("host", "localhost") \
9      .option("port", 9999) \
10     .load()
11
12 ## The input records are characterized by one single column called value
13 ## of type string
14 ## Example of an input record: s1,0,3,2016-03-11 09:00:04
15 ## Define four more columns by splitting the input column value
16 ## New columns:
17 ## - stationId
18 ## - freeslots
19 ## - usedslots
20 ## - timestamp
21 readingsDF = recordsDF \
22     .withColumn("stationId", split(recordsDF.value, ',')[0].cast("string")) \
23     .withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer")) \
24     .withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer")) \
25     .withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))
26
27 ## Filter data

```

```

28 ## Use the standard filter transformation
29 fullReadingsDF = readingsDF.filter("freeslots=0")
30
31 ## Count the number of readings with a number of free slots equal to 0
32 ## for in each window.
33 ## windowDuration = 2 seconds
34 ## no overlapping windows
35 countsDF = fullReadingsDF \
36     .groupBy(window(fullReadingsDF.timestamp, "2 seconds")) \
37     .agg({"*": "count"}) \
38     .sort("window")
39
40 ## The result of the structured streaming query will be stored/printed on
41 ## the console "sink"
42 ## complete output mode
43 ## (append mode cannot be used for aggregation queries)
44 queryCountWindowStreamWriter = countsDF \
45     .writeStream \
46     .outputMode("complete") \
47     .format("console") \
48     .option("truncate", "false")
49
50 ## Start the execution of the query (it will be executed until it is
51 ## explicitly stopped)
52 queryCountWindow = queryCountWindowStreamWriter.start()

```

Watermarking

Watermarking is a feature of Spark that allows the user to specify the threshold of late data, and allows the engine to accordingly clean up old state. Results related to old event-times are not needed in many real streaming applications: they can be dropped to improve the efficiency of the application, since keeping the state of old results is resource expensive; in this way every time new data are processed only recent records are considered.

Specifically, to run windowed queries for days, it is necessary for the system to bound the amount of intermediate in-memory state it accumulates. This means the system needs to know when an old aggregate can be dropped from the in-memory state because the application is not going to receive late data for that aggregate any more; to enable this, in Spark 2.1, watermarking has been introduced.

Watermarking lets the Spark Structured Streaming engine automatically track the current event time in the data and attempt to clean up old state accordingly. It allows to define the watermark of a query by specifying the event time column and the threshold on how late the data is expected to be in terms of event time: for a specific window ending at time T , the engine will maintain state and allow late data to update the state/the result until max event time seen by the engine $< T + \text{late threshold}$. In other words, late data within the threshold will be aggregated, but data later than $T + \text{threshold}$ will be dropped.

27.4 Join operations

Spark Structured Streaming manages also join operations

- Between two streaming DataFrames
- Between a streaming DataFrame and a static DataFrame

The result of the streaming join is generated incrementally.

When joining between two streaming DataFrames, for both input streams, past input streaming data must be buffered/recorded in order to be able to match every future input record with past input data and accordingly generate joined results. Too many resources are needed for storing all the input data, hence, old data must be discarded. Watermark thresholds must be defined on both input streams such that the engine knows how delayed the input can be and drop old data.

The methods `join()` and `withWatermark()` are used to join streaming DataFrames: the join method is similar to the one available for static DataFrame.

```

1  from pyspark.sql.functions import expr
2  impressions = spark.readStream. ...
3  clicks = spark.readStream. ...
4
5  ## Apply watermarks on event-time columns
6  impressionsWithWatermark = impressions \
7      .withWatermark("impressionTime", "2 hours")
8
9  clicksWithWatermark = clicks \
10     .withWatermark("clickTime", "3 hours")
11
12 ## Join with event-time constraints
13 impressionsWithWatermark.join(
14     clicksWithWatermark,
15     expr(
16         """
17             clickAdId = impressionAdId AND
18             clickTime >= impressionTime AND
19             clickTime <= impressionTime + interval 1 hour"""
20     )
21 )
```

28 Streaming data analytics frameworks

28.1 Stream processing frameworks for (big) streaming data analytics

Several frameworks have been proposed to process in real-time or in near real-time data streams:

- Apache Spark (Streaming component)
- Apache Storm
- Apache Flink
- Apache Samza
- Apache Apex
- Apache Flume
- Amazon Kinesis Streams
- ...

All these frameworks use a cluster of servers to scale horizontally with respect to the (big) amount of data to be analyzed.

Comparison among state of the art streaming frameworks

Apache Spark Streaming

- Micro-batch applications
- Processes each record exactly once

Apache Storm

- Continuous/real-time computation: very low latency
- Processes each record at least once in real-time: each record could be processed multiple times, hence may update mutable state twice
- Apache Storm Trident API: it is a running modality of Apache Storm that processes each record exactly once (micro-batch); slower than the Apache Storm version

Apache Flink

- Continuous/real-time stateful computations over data streams: low latency
- Processes each record exactly once

28.2 Introduction to Apache Storm

Apache Storm™ is a distributed framework that is used for real-time processing of data streams (e.g., Tweets analysis, Log processing). Currently, it is an [open source project](#) of the Apache Software Foundation. It is implemented in Clojure and Java (12 core committers, plus about 70 contributors).

Storm was first developed by Nathan Marz at BackType, a company that provided social search applications. Later (2011), BackType was acquired by Twitter, and now it is a critical part of their infrastructure. Currently, Storm is a project of the Apache Software Foundation (since 2013).

Data processing

- Continuous computation: Storm can do continuous computation on data streams in real time; it can process each message as it comes (an example of continuous computation is streaming trending topics detection on Twitter)
- Real-time analytics: Storm can analyze and extract insights or complex knowledge from data that come from several real-time data streams

Features of Storm

Storm is

- Distributed: Storm is a distributed system than can run on a cluster of commodity servers.
- Horizontally scalable: Storm allows adding more servers (nodes) to your Storm cluster and increase the processing capacity of your application. It is linearly scalable with respect to the number of nodes, which means that you can double the processing capacity by doubling the nodes.
- Fast: Storm has been reported to process up to 1 million tuples per second per node.
- Fault tolerant: Units of work are executed by worker processes in a Storm cluster. When a worker dies, Storm will restart that worker (on the same node or on to another node).
- Reliable - Guaranteed data processing: Storm provides guarantees that each message (tuple) will be processed at least once; in case of failures, Storm will replay the lost tuples, and it can be configured to process each tuple only once.
- Easy to operate: Storm is simple to deploy and manage. Once the cluster is deployed, it requires little maintenance.
- Programming language agnostic: Even though the Storm platform runs on Java Virtual Machine, the applications that run over it can be written in any programming language that can read and write to standard input and output streams.

28.3 Storm core concepts

Storm can be considered a distributed Function Programming-like processing of data streams. It applies a set of functions, in a specific order, on the elements of the input data streams and emits new data streams, however, each function can store its state by means of variables, and so it is not pure functional programming.

Main concepts

- Tuple
- Data Stream
- Spout
- Bolt
- Topology

Data model

The basic unit of data that can be processed by a Storm application is called a tuple: each tuple is a predefined list of fields. The data type of each field can be common data types, (e.g., byte, char, string, integer), or your own data types, which can be serialized as fields in a tuple. Each field of a tuple has a name.

A tuple is dynamically typed, that is, you just need to define the names of the fields in a tuple and not their data type.

Storm processes streams of tuples. Each stream

- is an unbounded sequence of tuples
- has a name
- is composed of homogenous tuples (i.e., tuples with the same structure)

However, each applications can process multiple, heterogenous, data streams.

i Example

Tuple

```
(1.1.1.1, "foo.com")
```

Stream of tuples

```
(1.1.1.1, "foo.com")
(2.2.2.2, "bar.net")
(3.3.3.3, "foo.com")
...
```

Spout

Spout is the component generating/handling the input data stream. Spouts read or listen to data from external sources and publish them (emit in Storm terminology) into streams.

i Examples

- A spout can be used to connect to the Twitter API and emit a stream of tweets
- A spout can be used to read a log file and emit a stream of composed of the its lines

Each spout can emit multiple streams, with different schemas; for example, it is possible to implement a spout that reads 10-field records from a log file and emits them as two different streams of 7-tuples and 4-tuples, respectively.

Spouts can be

- unreliable (fire-and-forget)
- reliable (can replay failed tuples)

Bolt

Bolt is the component that is used to apply a function over each tuple of a stream. Bolts consume one or more streams, emitted by spouts or other bolts, and potentially produce new streams.

Bolts can be used to

- filter or transform the content of the input streams and emit new data streams that will be processed by other bolts
- process the data streams and store/persist the result of the computation in some of “storage” (files, Databases, ..)

Each bolt can emit multiple streams, with different schemas.

Examples

- A bolt can be used to extract one field from each tuple of its input stream
- A bolt can be used to join two streams, based on a common field
- A bolt can be used to count the occurrences of a set of URLs

The input streams of a Storm cluster are handled by spouts

- Each spout passes the data streams to bolts, which transform them in some way
- Each bolt either persists the data in some sort of storage or passes it to some other bolts

A Storm program is a chain of bolts making some computations/transformations on the data exposed by spouts and bolts.

Topology

A Storm topology is an abstraction that defines the graph of the computation: it specifies which spouts and bolts are used and how they are connected. A topology can be represented by a direct acyclic graph (DAG), where each node does some kind of processing and eventually forwards it to the next node(s) in the flow (i.e., a topology in Storm wires data and functions via a DAG).

Figure 28.1: Topology example

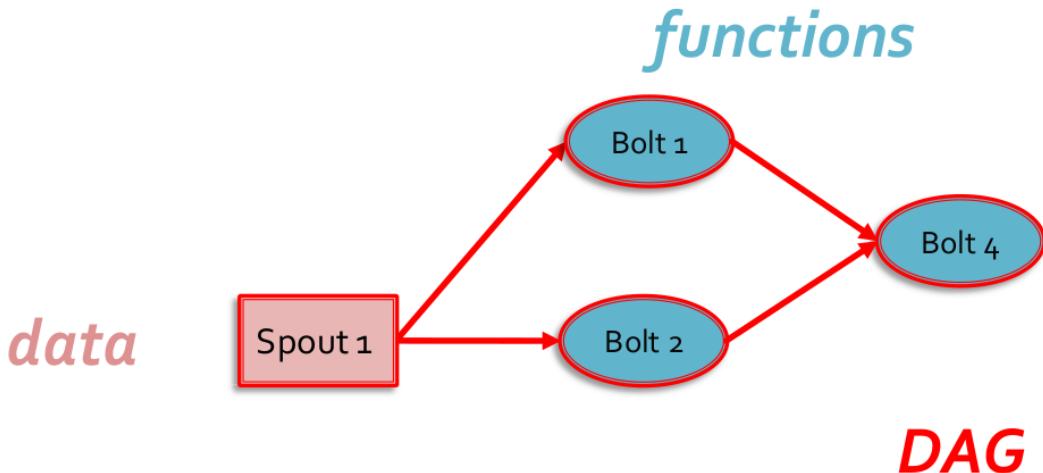
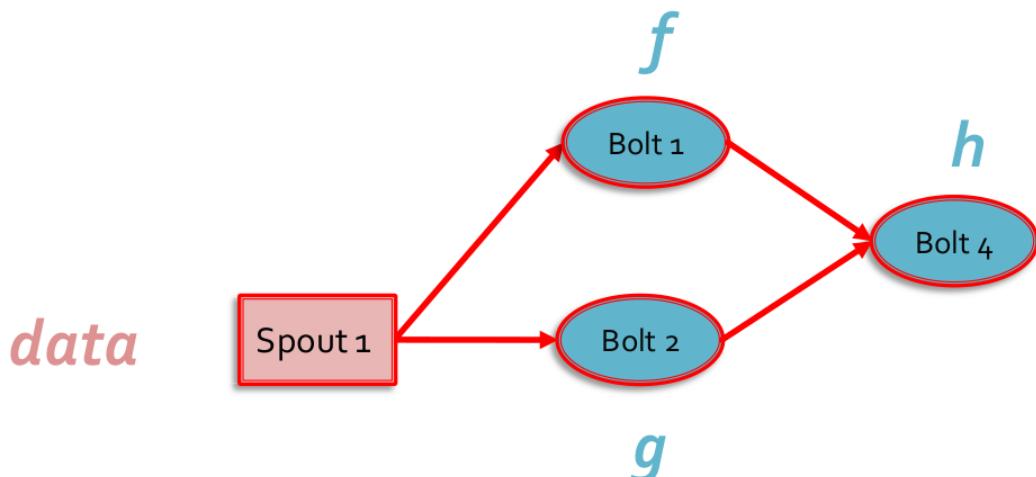
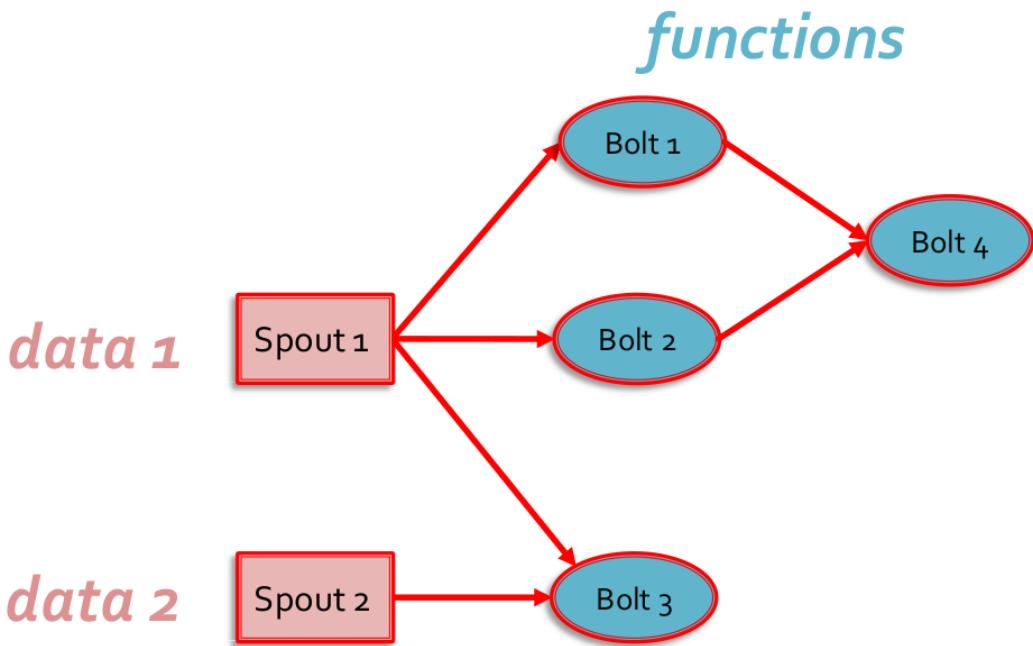


Figure 28.2: Functional programming



$h(f(data), g(data))$

Figure 28.3: Topology example



- Notice that there are two input data streams in this topology
- There are also two output data streams

Topology execution

The topology is executed on the servers of the cluster running Storm. The system automatically decides which parts of the topology are executed by each server of the cluster

- Each topology runs until it is explicitly killed
- Each cluster can run multiple topologies at the same time

Worker processes

- Each node in the cluster can run one or more JVMs called worker processes that are responsible for processing a part of the topology.
- Each topology executes across one or more worker processes
- Each worker process is bound to one of the topologies and can execute multiple components (spouts and/or bolts) of that topology; hence, even if multiple topologies are run at the same time, none of them will share any of the workers

Executor

- Within each worker process, there can be multiple threads that execute parts of the topology. Each of these threads is called an executor
- An executor can execute only one of the components of the topology, that is, any one spout or bolt in the topology, but it may run one or more tasks for the same component

- Each spout or bolt can be associated with many executors and hence executed in parallel

Tasks

A task is the most granular unit of task execution in Storm: each task is an instance of a spout or bolt, and it performs the actual data processing

- Each spout or bolt that you implement in your code executes as many tasks across the cluster
- Each task can be executed alone or with another task of the same type (in the same executor)

The number of tasks for a component is always the same throughout the lifetime of a topology (it is set when the topology is submitted), but the number of executors (threads) for a component can change over time (i.e., it is possible to add/remove executors for each component).

The parallelism of the topology is given by the number of executors (i.e., number of threads). For each spout/bolt the application can specify

- The number of executors: this value can be changed at runtime
- The number of tasks: this value is set before submitting the topology and cannot be change at runtime