Introducton to Data Engineering 8

Paul Blondel

UTSEUS, Shanghai

March 30st, 2021

‘There was 5 exabytes of information created between the dawn of civilization through 2003, but that much information is now created every 2 days, and the pace is increasing.’

Eric Schmidt, PDG Google, 2010

1. What is Big Data?

2. Hadoop
    - HDFS: designed for massive storage
    - Hadoop MapReduce: towards massive parallel computation
    - Pig and Pig Latin
    - Examples of Pig commands
    - Hadoop: pros and cons

3. Spark
    - What is Spark?
    - Comparison with Hadoop MapReduce
    - Spark's Resilient Distributed Dataset (RDDs)
    - PySpark: Spark with Python!

1. What is Big Data?
2. Hadoop
   - HDFS: designed for massive storage
   - Hadoop MapReduce: towards massive parallel computation
   - Pig and Pig Latin
   - Examples of Pig commands
   - Hadoop: pros and cons
3. Spark
   - What is Spark?
   - Comparison with Hadoop MapReduce
   - Spark's Resilient Distributed Dataset (RDDs)
   - PySpark: Spark with Python!

What is Big Data?

Big Data refers to **datasets that become so large, so various and changing so rapidly** that they are difficult to work with using traditional database management or information management tools.

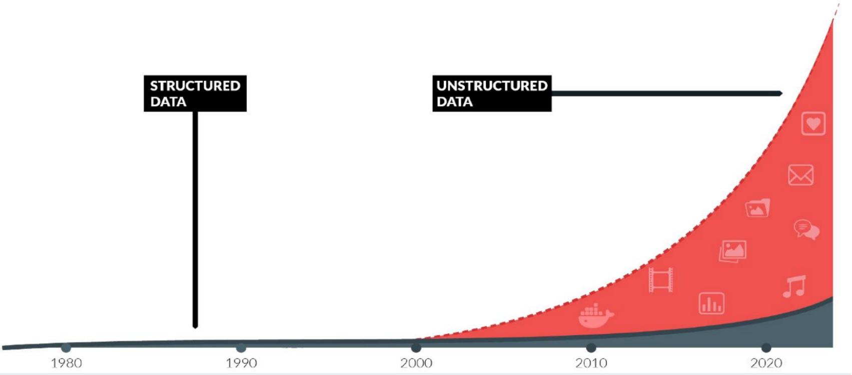


Figure: Exponential growth of data in the recent years

When refering to Big Data it is common to talk about the three Vs: **high Volume** of data, **high Velocity** of data input and **high Variety** of data (structured+unstructured):

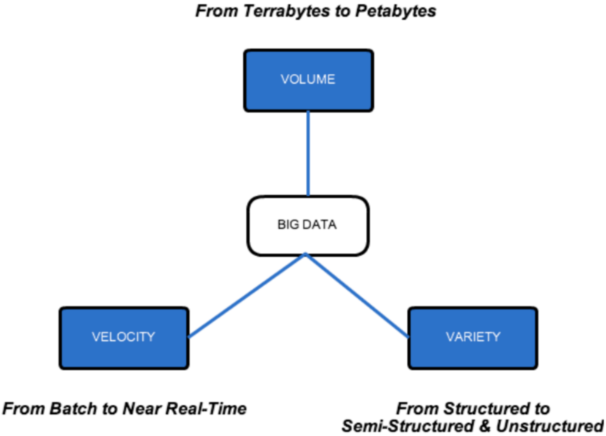

From Terrabytes to Petabytes
VOLUME
BIG DATA
VELOCITY
VARIETY
From Batch to Near Real-Time
From Structured to
Semi-Structured & Unstructured

Figure: The three Vs.

- **High Volume**:
  - Data cannot fit anymore inside one server (Ex: Facebook in 2018: 7 Petabytes (10E15) of new data per day
  - Exponential trend: 90% of the total data created was created in the last two years!
- **High Velocity**:
  - Increasing flows of data need to be analyzed (ex: stock market inform, user behavior of e-commerce website)
  - Information needs more and more to be analyzed in near real-time
- **High Variety**:
  - Data can be raw, semi-structured or even unstructured (Ex: text, images, videos, metadata, computer code etc.)

Structured VS unstructured data, what is the difference?

- **Structured data**: information (words, signs, numbers...) controlled by repositories and presented in boxes (fields of a database) allowing their interpretation and processing by computers
- **Unstructured data**: the rest, everything that is not organized in a database (email, images, videos, etc.)

1 What is Big Data?

2 Hadoop

- HDFS: designed for massive storage
- Hadoop MapReduce: towards massive parallel computation
- Pig and Pig Latin
- Examples of Pig commands
- Hadoop: pros and cons

3 Spark

- What is Spark?
- Comparison with Hadoop MapReduce
- Spark's Resilient Distributed Dataset (RDDs)
- PySpark: Spark with Python!

- Processing such large amounts of data **requires a new paradigm**...
- Indeed, a single server machine, even a high-end one is unable to process so much information

Solution

Distribute the data on several server machines and use massive parallel calculation and distributed storage with Hadoop technology

Before Big Data: adding memory and CPU power permitted to **scale vertically** a server capacity

Figure: Vertical scaling

However this approach has a **lot of limitations when dealing with Big Data**... To deal with Big Data, you have to **horizontally scale** a cluster of machines which is easier to scale and more robust:

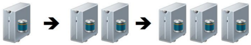


Figure: Horizontal scaling of a cluster

Cluster

A cluster of machines is a set of computers that work together so they **can be viewed from outside as one single system**

A cluster of machines using Hadoop:

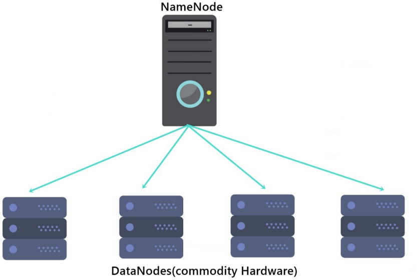


Figure: A Hadoop cluster made of a Namenode and Datanodes

Hadoop

Hadoop is a technology permitting the storage and the processing of massive amount of data on a cluster

- **Storage with Hadoop cluster**:
  - All the **machines are connected to each other in order to share storage space** using **HDFS**
  - Ex: Cloud is an example of a distributed storage space: files are stored on different machines and duplicated
- **Processing with Hadoop cluster**:
  - The execution of a program is also distributed: it is **executed on one or more machines of the cluster**
  - Specifically with Hadoop: the **MapReduce paradigm** allows a cluster of computers to execute tasks in chunks in an extremely parallel manner

To store massive data on a cluster we need:

- A **special file system allowing to see only one space containing gigantic and very numerous files** (**HDFS**)
- Specifically designed **databases working on cluster of machines** (HBase, Cassandra, ElasticSearch)

To compute massive computation on a cluster we need:

- A new processing paradigm:
  - This new paradigm is called **MapReduce**
  - With this: **easy to parallelize executions at a massive scale**

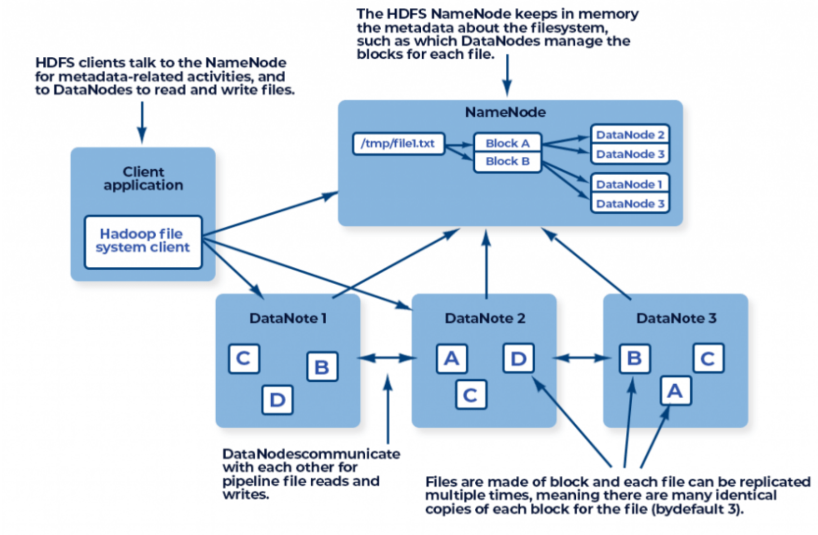
Content of the course

We will talk about Hadoop's **HDFS** and Hadoop **MapReduce** and **Spark**

HDFS (Hadoop Distributed File System) is the **distributed file system** (across a cluster) of Hadoop:

- Inspired by "Google File System"
- HDFS is composed of two services:
  - **Namenode**: contains all the **names and the locations of the block files** in the FS (like a big phone book)
  - **Datanode**: **store the blocks of the stored files**
- A **file written to HDFS is divided into blocks** (64MB to 256MB each)
- **Blocks are replicated and distributed across multiple Datanodes** (usually 3x)
- The FS queries the Namenode to **learn the stucture of the file tree and find where are the blocks**
- **Clients access data directly from the Datanode**, with having this knowledge

How HDFS read and write data in the cluster:



The HDFS NameNode keeps in memory the metadata about the filesystem, such as which DataNodes manage the blocks for each file.

HDFS clients talk to the NameNode for metadata-related activities, and to DataNodes to read and write files.

DataNodescommunicate with each other for pipeline file reads and writes.

Files are made of block and each file can be replicated multiple times, meaning there are many identical copies of each block for the file (bydefault 3).

HDFS is a distributed file system:

- **Files and folders are organized in a tree** (like Unix)
- Files are **stored on a large number of machines** in such a way that the **exact position of files is invisible**
- **File access is transparent** whatever the machines that contain the files
- Files are copied in several copies for **robustness** and allow **multiple simultaneous accesses**

HDFS **allows to see all folders and files of these thousands of machines as a single machines as a single tree** containing Po of data, as if they were on the local hard drive.

The "**hdfs dfs**" command and its options allows to manage the files and folders:

```
hdfs dfs -help
hdfs dfs -ls [names...] (no -l option)
hdfs dfs -cat name
hdfs dfs -mv old new
hdfs dfs -cp old new
hdfs dfs -mkdir folder
hdfs dfs -rm -f -r folder (no -en option)
hdfs dfs -chmod 777 /stuff
```

These commands take some time to react (HDFS is slow) mainly these are programs written in Java loading a lot of Java libraries (jars).

Commands to exchange content between HDFS and the "world":

```
hdfs dfs -copyFromLocal filesource filedestination
hdfs dfs -put filesource [filedestination]
```

To extract a file from HDFS, there are two possible commands:

```
hdfs dfs -copyToLocal filesource destination files
hdfs dfs -get filesource [filedestination]
```

Other examples:

```
hdfs dfs -mkdir -p books
wget http://www.textfiles.com/etext/FICTION/dracula
hdfs dfs -put dracula books
hdfs dfs -ls books
hdfs dfs -get books/center_earth
```

HDFS benefits:

- HDFS is **compatible with a lots of Big Data tools** (from the Hadoop ecosystem and other)
- A **highly scalable storage** model
- Availability of support (Hadoop "vendors")

HDFS disadvantages:

- Configuration is **complex and fragile**
- **Only supports replication to avoid data loss**
- **No possibility to gather clusters** of HDFS together
- The **need to query the Namenode for each read**
- **Number of blocks limited by the Namenode memory space**...
- Relatively **slow**

As mentioned before, MapReduce is the paradigm to use to apply computation on massive data distributed on a Hadoop cluster.

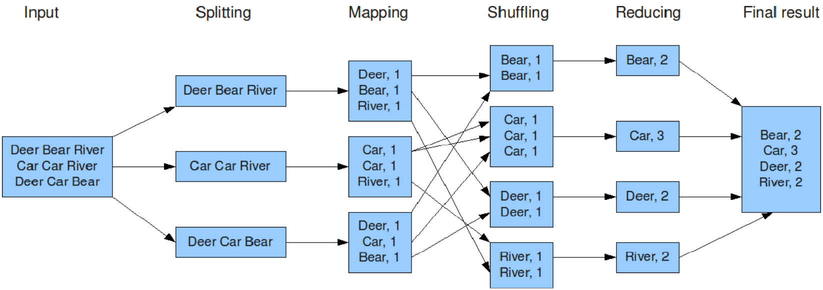
Let's see in more details what is MapReduce ?

- Invented in 2004 by Google
- Based on a simple **Divide and Conquer** approach
- However: **not all problems can be tackled with MapReduce** (must be a parallelizable problem)

MapReduce uses the Hadoop architecture:

- **Input data is on HDFS** (blocks in the HDFS, spread over several machines)
- **For each block is applied a map function** which takes:
    - in input **pairs** (key, value)
    - returns **pairs** (key, value)
- **Results are grouped by keys** (shuffle) and **distributed on the different machines**
- For **each group of pairs with the same key a reduce function is applied** taking a (key,value) input and returns **pairs** (key,value)

The counting words example to illustrate how MapReduce works:



A MapReduce framework is generally composed of these operations:

- **Split**: Divides the input data into parallel streams to provide the cluster nodes (only needed if data already stored)
- **Read**: Reads the data streams by splitting them into units to be process (for a text file: 1 unit = 1 line)
- **Map**[1]: Each worker node applies the map function to the local data and writes the output to temporary storage (the result of which is stored in <key,value> pairs)
- **Shuffle**: Worker nodes redistribute data according to the output keys (produced by the map function) so that all data belonging to a key is on the same worker node
- **Reduce**[1]: The worker nodes now process each group of output data by key in parallel

[1] Map and Reduce: the most important operations

Developers are **responsible to develop the mapper and the reducer** (sometimes a **combiner** to "pre-reduce data" before mapping as well).

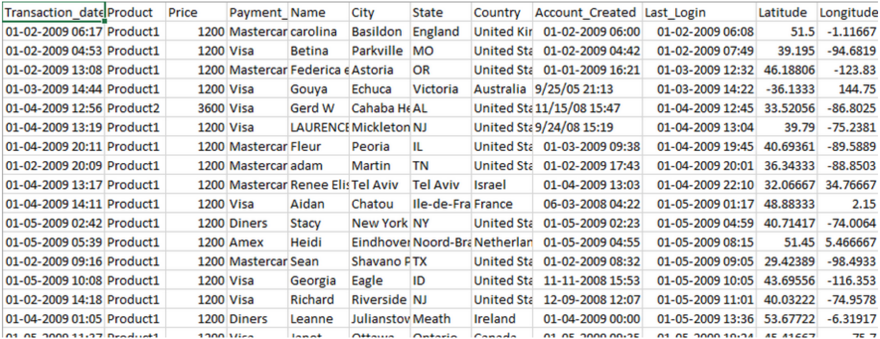
The following pseudo-code describes a mapper and a reducer for counting distinct words in a massive text:

```
Map(String input_key, String input_values) :
   foreach word w in input_values:
      EmitIntermediate( w, "‘1"’);
Reduce (String key, Iterator intermediate_values):
   int result=0;
   foreach v in intermediate_values:
      result += ParseInt( v );
      Emit( key, String( result ));
```

The main Hadoop programming language

On Hadoop mappers and reducers are coded in the **Java programming language**

Now let's see a real example of a **mapper** and a **reducer** coded in Java[1]. The goal here is to count distinct countries.



Transaction_date	Product	Price	Payment_	Name	City	State	Country	Account_Created	Last_Login	Latitude	Longitude
01-02-2009 06:17	Product1	1200	Mastercar	carolina	Basildon	England	United Kir	01-02-2009 06:00	01-02-2009 06:08	51.5	-1.11667
01-02-2009 04:53	Product1	1200	Visa	Betina	Parkville	MO	United Sta	01-02-2009 04:42	01-02-2009 07:49	39.195	-94.6819
01-02-2009 13:08	Product1	1200	Mastercar	Federica e	Astoria	OR	United Sta	01-01-2009 16:21	01-03-2009 12:32	46.18806	-123.83
01-03-2009 14:44	Product1	1200	Visa	Gouya	Echuca	Victoria	Australia	9/25/05 21:13	01-03-2009 14:22	-36.1333	144.75
01-04-2009 12:56	Product2	3600	Visa	Gerd W	Cahaba He	AL	United Sta	11/15/08 15:47	01-04-2009 12:45	33.52056	-86.8025
01-04-2009 13:19	Product1	1200	Visa	LAURENCE	Mickleton	NJ	United Sta	9/24/08 15:19	01-04-2009 13:04	39.79	-75.2381
01-04-2009 20:11	Product1	1200	Mastercar	Fleur	Peoria	IL	United Sta	01-03-2009 09:38	01-04-2009 19:45	40.69361	-89.5889
01-02-2009 20:09	Product1	1200	Mastercar	adam	Martin	TN	United Sta	01-02-2009 17:43	01-04-2009 20:01	36.34333	-88.8503
01-04-2009 13:17	Product1	1200	Mastercar	Renee Eli:	Tel Aviv	Tel Aviv	Israel	01-04-2009 13:03	01-04-2009 22:10	32.06667	34.76667
01-04-2009 14:11	Product1	1200	Visa	Aidan	Chatou	Ile-de-Fra	France	06-03-2008 04:22	01-05-2009 01:17	48.88333	2.15
01-05-2009 02:42	Product1	1200	Diners	Stacy	New York	NY	United Sta	01-05-2009 02:23	01-05-2009 04:59	40.71417	-74.0064
01-05-2009 05:39	Product1	1200	Amex	Heidi	Eindhover	Noord-Bra	Netherlan	01-05-2009 04:55	01-05-2009 08:15	51.45	5.466667
01-02-2009 09:16	Product1	1200	Mastercar	Sean	Shavano F	TX	United Sta	01-02-2009 08:32	01-05-2009 09:05	29.42389	-98.4933
01-05-2009 10:08	Product1	1200	Visa	Georgia	Eagle	ID	United Sta	11-11-2008 15:53	01-05-2009 10:05	43.69556	-116.353
01-02-2009 14:18	Product1	1200	Visa	Richard	Riverside	NJ	United Sta	12-09-2008 12:07	01-05-2009 11:01	40.03222	-74.9578
01-04-2009 01:05	Product1	1200	Diners	Leanne	Julianstov	Meath	Ireland	01-04-2009 00:00	01-05-2009 13:36	53.67722	-6.31917
01-05-2009 11:37	Product1	1200	Visa	Janet	Ottawa	Ontario	Canada	01-05-2009 09:25	01-05-2009 10:24	45.41667	-75.7

Figure: Sample of all the data contained in the HDFS (presented as columns)

[1] You can find the corresponding example here: https://www.guru99.com/create-your-first-hadoop-program.html

The mapper of this example (see the content lines 15 to 17):

```
1    package SalesCountry;
2
3    import java.io.IOException;
4
5    import org.apache.hadoop.io.IntWritable;
6    import org.apache.hadoop.io.LongWritable;
7    import org.apache.hadoop.io.Text;
8    import org.apache.hadoop.mapred.*;
9
10   public class SalesMapper extends MapReduceBase implements Mapper <LongWritable, Text, Text, IntWritable> {
11           private final static IntWritable one = new IntWritable(1);
12
13           public void map(LongWritable key, Text value, OutputCollector <Text, IntWritable> output,
14           Reporter reporter) throws IOException {
15                   String valueString = value.toString();
16                   String[] SingleCountryData = valueString.split(",");
17                   output.collect(new Text(SingleCountryData[7]), one);
18           }
19   }
```

The reducer of this example (see the content lines 15 to 22):

```
 1    package SalesCountry;
 2
 3    import java.io.IOException;
 4    import java.util.*;
 5
 6    import org.apache.hadoop.io.IntWritable;
 7    import org.apache.hadoop.io.Text;
 8    import org.apache.hadoop.mapred.*;
 9
10    public class SalesCountryReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text,
11    IntWritable> {
12            public void reduce(Text t_key, Iterator<IntWritable> values,
13            OutputCollector<Text,IntWritable> output,
14            Reporter reporter) throws IOException {
15                    Text key = t_key;
16                    int frequencyForCountry = 0;
17                    while (values.hasNext()) {
18                          // replace type of value with the actual type of our value
19                          IntWritable value = (IntWritable) values.next();
20                          frequencyForCountry += value.get();
21                    }
22                    output.collect(key, new IntWritable(frequencyForCountry));
23            }
24    }
```

YARN (Yet Another Resource Negotiator) is used in the Hadoop ecosystem launch MapReduce "jobs" coded in Java:

- YARN allows to **manage jobs on a cluster** of machines
- YARN can **move a process from one machine to another** in case of failure or progress judged too slow
- YARN is **transparent for the user**
- YARN **makes sure that it is executed executed as quickly as possible**

YARN does the MapReduce:

1. At the beginning, YARN **queries the location of the data from the Namenode** (and decompress on Datanodes)
2. YARN **splits building pairs** to be supplied to the Map tasks
3. YARN **creates Map processes** on each machine containing part of the data
4. YARN **sorts the pairs coming out of Map according to their key** and sends them to the machine running the **Reduce task concerned by this key**
5. The **Reduce tasks receive a list of pairs and perform the reduction of values** (max, sum, avg...)

```
yarn jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-files mapper.py,reducer.py \
-mapper mapper.py -reducer reducer.py \
-input livres -output sortie
```

There is an easier way to perform MapReduce jobs: **PIG** (and **Pig Latin**)

Pig was originally created by "Yahoo!". Pig allows to **write useful treatments on data without undergoing the complexity of Java**.

Pig's goal

The goal is to make Hadoop accessible to people that are not Computer Science scientists: physicists, statisticians, mathematicians...

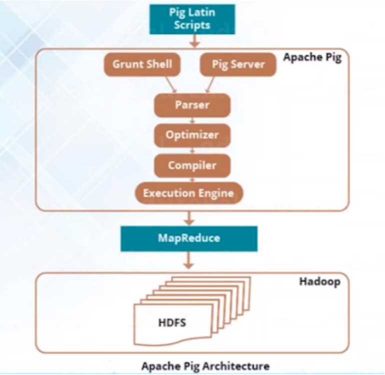
Differences between Pig and Hadoop MapReduce:

Pig	**Hadoop MapReduce**
Data flow language	Data processing framework
High-level language	Low-level and rigid
Can easily perform a Join	Perform join between datasets very difficult
Basic knowledge of SQL is enough for Apache Pig	Need to be familiar with Java
Use a multi-query approach	Require almost 20 times more code for same task
No need for compilation	Go through a long compilation process

There are some similarities between SQL and Pig Latin (common keywords: JOIN, ORDER, LIMIT...) but their principle is different:

- **In SQL**: You **build queries that describe the data to obtain**. We don't know how the SQL engine will calculate the result. We only know that internally, the query will be broken down into loops and comparisons on the data and making the best use of the indexes
- **In Pig Latin**: We **build programs that contain the explicit instructions**. We describe exactly how the result should be obtained, what calculations should be result is to be obtained, what calculations are to be made and in what order.

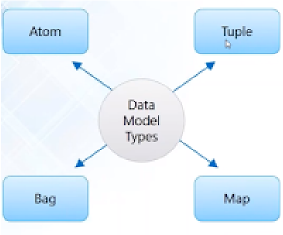
- A Pig Latin program consists of **a series of transformations to apply to input data** to produce output data
- **These operations describe a data flow**, translated into an executable representation
- Underneath, **these transformations are actually performed by series of hidden MapReduce jobs**
- Pig allows the programmer to **focus on data** rather than the nature of execution
- Pig Latin is a **relatively easy language** using familiar keywords from data processing (Join, Group, Filter, etc)



Apache Pig Architecture

- **Pig Latin**: Series of transformations applied to input data to produce results
- **Pig Latin Script**: Contains the pig commands in a file (.pig)
- **Grunt Shell**: Interactive shell for running pig commands
- **Parser**: Checks the syntax of the script and other miscellaneous checks
- **Optimizer**: Carries out the logical optimizations
- **Compiler**: Compiles the optimized logical plan into a series of MapReduce jobs
- **Execution Engine**: Submits MapReduce jobs to Hadoop in a sorted order

The Pig Latin data model:



- **Atom**: Any single value in Pig Latin, stored as string and can be used as string and number (Ex: 1, 'a', 34)
- **Tuple**: Similar to a row in a table of relational database (Ex: ("Jean", 35, "Teacher"))
- **Bag**: Unordered set of tuples (Ex: (1,Linkin Park,7,California), (Metalica,8),(Mega Death,5,Los Angeles))
- **Map**: A set of key-value pairs (Ex: Jean#37)
- **Relation**: A bag of tuples...

Introduction to Pig Latin:

Language providing **statements** as basic constructs which include **expressions**, **schemas** and **ends with a semicolon** (;)

In general (expect for the LOAD and STORE operations), Pig Latin statements **take a relation as input and produce another relation as output**

Notes

- When entering a LOAD statement in the Grunt shell: a semantic check of Pig Latin is carried out
- To see schema of loaded content: DESCRIBE (see content: DUMP)
- See more operators below...

List of relational operators of Pig Latin:

Loading and Storing

LOAD	Load the data from HDFS into a relation Relation_Name = LOAD A_File_Path USING A_Function as schema; (Functions can be: BinStorage, JsonLoader, PigStorage, TextLoader)
STORE	Save a relation to HDFS STORE Relation_Name INTO A_File_Path [USING A_Function];

Filtering

FILTER	Remove unwanted rows from a relation Relation_Name_2 = FILTER Relation_Name BY (condition);
DISTINCT	Remove duplicate rows from a relation Relation_Name_2 = DISTINCT Relatin_Name;
FOREACH, GENERATE	Generate data transformations based on columns of data Relation_name_2 = FOREACH Relatin_Name GENERATE (required data);
STREAM	Transform a relation using an external program

Grouping and Joining

JOIN	Join two or more relations Relation_Name = JOIN Relation_1 BY Field_A, Relation_2 BY Field_B;
COGROUP	Group the data in two or more relations Cogroup_Data = COGROUP Field_B by Field_A, Field_C by Field_A;
GROUP	Group the data in a single relation Group_data = GROUP Relation_name BY age;
CROSS	Create the cross product of two or more relations Relation_C = CROSS Relation_A, Relation_B;

Sorting

ORDER	Arrange a relation in a sorted order based on one or more fields (ascending or descending) Relation_2 = ORDER Relation_1 BY [ASC|DESC] Field_A;
LIMIT	Get a limited number of tuples from a relation Relation_2 = LIMIT Relation_1 N;

Combining and Splitting

UNION	Combine two or more relations into a single relation Relation_C = UNION Relation_A, Relation_B;
SPLIT	Split a single relation into two or more relations SPLIT Relation_1 INTO Relation_2 IF expression,

Diagnostic Operators

DUMP	Print the content of a relation on the console dump Relation_Name;
DESCRIBE	Describe the schema of a relation describe Relation_name;
EXPLAIN	View the logical, physical or MapReduce execution plans to compute a relation explain Relation_name;
ILLUSTRATE	View the step-by-step execution of a series of statements illustrate Relation_name;

Examples of Pig commands

Imagine we have a dept.csv file stored on HDFS, here load it and perform some basic actions on the data:

```
grunt> dept = LOAD ’/pig/dept.csv’ USING PigStorage(’,’) as (deptno:int,dname:chararray,loc:chararray);
grunt> dump dept;
(10,ACCOUNTING,NEW YORK)
(20,RESEARCH,DALLAS)
(30,SALES,CHICAGO)
(40,OPERATIONS,BOSTON)
grunt> describe dept;
dept: {deptno: int,dname: chararray,loc: chararray}
```

Let’s group the data and see what we get:

```
grunt> group_multiple = group dept by (deptno,dname);
grunt> dump group_multiple;
((10,ACCOUNTING),{(10,ACCOUNTING,NEW YORK)})
((20,RESEARCH),{(20,RESEARCH,DALLAS)})
((30,SALES),{(30,SALES,CHICAGO)})
((40,OPERATIONS),{(40,OPERATIONS,BOSTON)})
```

Here we perform a JOIN operation:

```
grunt> e1 = load ’/pig/emp.csv’ using PigStorage(’,’) as
(EMPNO:int,ENAME:chararray,JOB:chararray,MGR:int,HIREDATE:datetime,SAL:int,COMM:int,DEPTNO:int);
grunt> e2 = load ’/pig/emp2.csv’ using PigStorage(’,’) as
(EMPNO:int,ENAME:chararray,JOB:chararray,MGR:int,HIREDATE:datetime,SAL:int,COMM:int,DEPTNO:int);
grunt> emp_join = JOIN e1 BY EMPNO, e2 BY MGR;
grunt> dump emp_join;
(7566,JONES,MANAGER,7839,,2975,,20,7788,SCOTT,ANALYST,7566,,3000,,20)
(7566,JONES,MANAGER,7839,,2975,,20,7902,FORD,ANALYST,7566,,3000,,20)
(7698,BLAKE,MANAGER,7839,,2850,,30,7654,MARTIN,SALESMAN,7698,,1250,1400,30)
(7698,BLAKE,MANAGER,7839,,2850,,30,7900,JAMES,CLERK,7698,,950,,30)
(7698,BLAKE,MANAGER,7839,,2850,,30,7521,WARD,SALESMAN,7698,,1250,500,30)
```

Now we filter the data:

```
grunt> filter_emp = FILTER emp BY JOB == ’CLERK’;
grunt> dump filter_emp;
(7369,SMITH,CLERK,7902,,800,,20)
(7876,ADAMS,CLERK,7788,,1100,,20)
(7900,JAMES,CLERK,7698,,950,,30)
(7934,MILLER,CLERK,7782,,1300,,10)
```

We reorder data:

```
grunt> order_by_data = ORDER dept BY loc ASC;
grunt> dump order_by_data
(40,OPERATIONS,BOSTON)
(30,SALES,CHICAGO)
(20,RESEARCH,DALLAS)
(10,ACCOUNTING,NEW YORK)
```

We store the data:

```
grunt> store dept into ’/dump.txt’ using PigStorage(’,’);
```

Pig also comes with a set of User Defined Functions (UDFs)

AVG()	Computes average of numeric values
PigStorage()	Load and store data as structured text files
TextLoader()	Load unstructured data in UTF-8 format
BinStorage()	Load and store the data in machine readable format
TOBAG()	Convert expressions to individual tuples (placed in a bag)
TOTUPLE()	Convert one or more expressions to tuple
TOMAP()	Convert the key-value pairs into a Map

Note

Pig Latin is a complete langage! And mastering Pig Latin is not the objective of this course, if you want to know more about it, you can go to this website: http://pig.apache.org/docs/r0.17.0/

Hadoop Pros and Cons

Pros

- **Cost**: Hadoop is **open-source and uses cost-effective commodity hardware which provides a cost-efficient model**
- **Scability**: Hadoop is a **highly scalable model**. A large amount of data is divided into multiple inexpensive machines in a cluster which is processed parallelly
- **Flexibility**: Hadoop is designed in such a way that it can **deal with any kind of dataset** like **structured** (MySQL Data), **Semi-Structured** (XML, JSON), **Un-structured** (Images and Videos) efficiently
- **Speed**: With Hadoop you **can easily access TBs of data** in just a few minutes
- **Fault tolerance**: Hadoop uses commodity hardware which can be crashed at any moment: **data is then replicated** on various DataNodes

Cons

- **Problem with small files**: Hadoop stores the file in the form of file blocks which are from 128MB in size (by default) to 256MB. This **so many small files surcharge the Namenode and make it difficult to work**
- **Lots of disk access**: In Hadoop, **intermediary data results are often read or write to/from the disk** leading to some slowlessness and inefficiency
- **Vulnerability**: Although is everything for an organization, by default **the security feature in Hadoop is made un-available**
- **Difficulty**: Hadoop requires you **learn to write specific Java MapReduce routines or Pig Latin commands**

1. What is Big Data?

2. Hadoop
   - HDFS: designed for massive storage
   - Hadoop MapReduce: towards massive parallel computation
   - Pig and Pig Latin
   - Examples of Pig commands
   - Hadoop: pros and cons

3. Spark
   - What is Spark?
   - Comparison with Hadoop MapReduce
   - Spark's Resilient Distributed Dataset (RDDs)
   - PySpark: Spark with Python!

What is Spark?

- Spark is a **unified analytics engine for large-scale data processing**
- Which is **10 to 100x faster than Hadoop MapReduce**

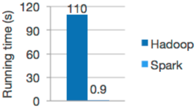
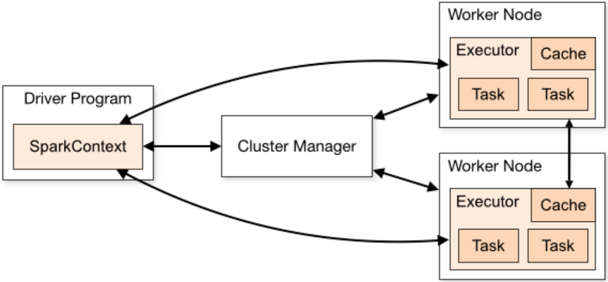


Figure: Logistic regression comparison Hadoop MapReduce VS Spark

- Spark is **also uses a cluster of nodes for computation** (here worker=DataNote, master node=Namenode)
- Spark applications run as independent sets of processes on a cluster **coordinated by the SparkContext object** in your main program (called the driver program)



1. The SparkContext **can connect to several types of cluster managers** to allocate resources (Spark's own cluster manager, Mesos or even YARN)
2. Once connected, Spark acquires executors on nodes in the cluster, **running computations and storing data for your application**. Once acquired, **application code** (defined by JAR or Python files passed to SparkContext) is **sent to the executors**
3. Finally, the **SparkContext sends tasks to the executors to run**...

Spark overcomes some of the biggest limitations of Hadoop:

- Spark is potentially up **10 to 100 times faster** than Hadoop MapReduce
- Spark **utilizes RAM, intermediary data is not stored on disk**
    - Hadoop stores a lot of intermediary data on disks for computation
- Spark **works well for smaller data-sets that can all fit into the RAM**
    - Hadoop is **more effective for truly massive datasets**
- Spark is more popular than Hadoop MapReduce
    - Spark is **easier to learn** and you can use **Python and Scala with it**

Reasons why Spark is s much faster than Hadoop MapReduce:

- Spark provides **in-memory computation** (designed to **transform data in-memory and hence reduces computation time**. While MapReduce writes/read a lot of intermediate results back and forth to/from Disk
- Spark utilizes **Direct Acyclic Graph** that **helps to do all the optimization and computation in a single stage** rather than multiple stages in the MapReduce model
- **Spark core is developed using the SCALA programming language** which is faster than JAVA

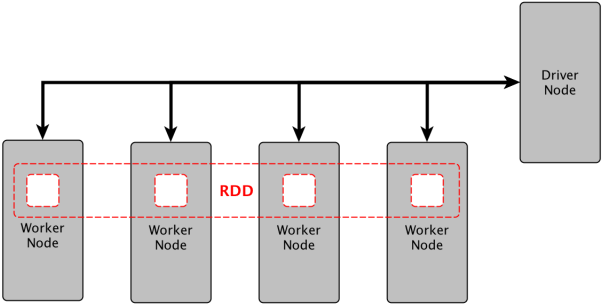
RDD (Resilient Distributed Dataset) is the **fundamental data structure of Spark** which are an **immutable**[2] collection of objects which computes on the different node of the cluster.

Decomposing the name RDD:

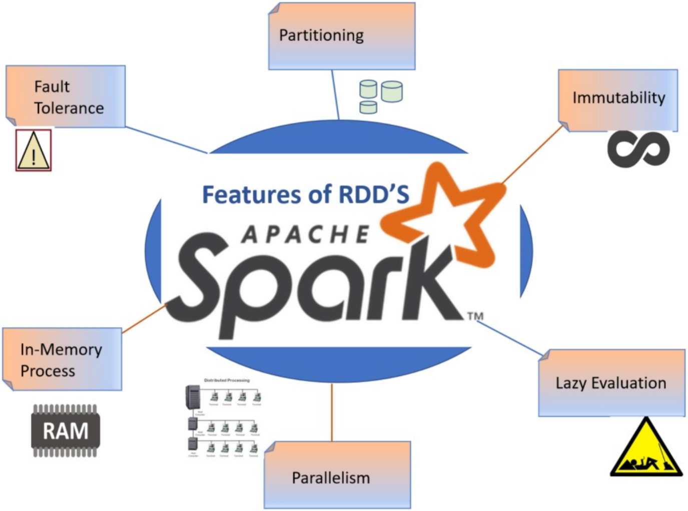
- **Resilient**: **fault-tolerant with the help of RDD lineage graph (DAG)** and so able to recompute missing or damaged partitions due to node failures
- **Distributed**: Data **resides on multiple nodes** (as partitions of RDD)
- **Dataset**: Records of the data you work with. The user can load the data set externally which can be either **JSON file, CSV file, text file or database via JDBC with no specific data structure**

[2] An immutable object is an object whose internal state remains constant after it has been entirely created

As mentioned above, **each and every dataset in Spark RDD is logically partitioned across many servers** so that they can be computed on different nodes of the cluster.



Spark's Resilient Distributed Dataset (RDDs)



Features of RDD'S

Fault Tolerance

Partitioning

Immutability

In-Memory Process

Parallelism

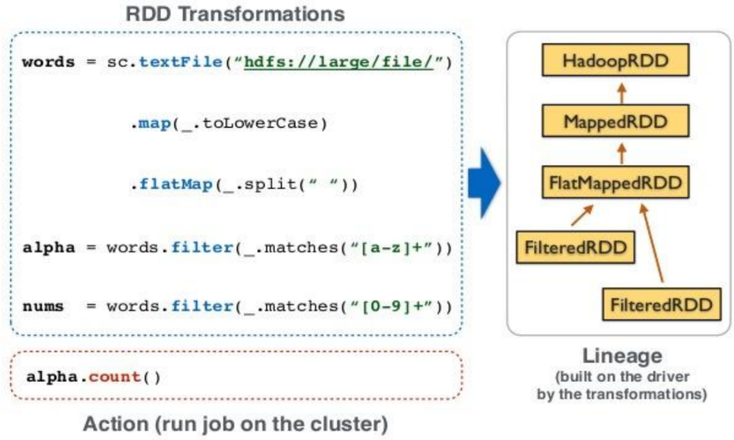
Lazy Evaluation

Main features of RDD:

- **In-memory Computation**: Intermediate results are stored in distributed memory (RAM) instead of stable storage (disk)
- **Lazy Evaluations**: All **transformations in Spark are lazy** the results are not calculated right away. Instead, they **just remember the transformations to apply**. Spark **only computes the transformations when an action requires a result for the driver program**
- **Fault Tolerance**: Spark RDDs are fault tolerant as they **track data lineage information to rebuild lost data** automatically on failure
- **Partitioning**: Partition is **the fundamental unit of parallelism in Spark RDD**. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions
- **Immutability**: Data is **safe to share across processes**. It can also be created or retrieved anytime which makes caching, sharing and replication easy

More information about the Fault Tolerance feature of RDD:

The **RDD data lineage is stored in the master node of the cluster** and allow to **restart the processing from everywhere** in case of problem:



```
words = sc.textFile(“hdfs://large/file/”)

            .map(_.toLowerCase)

            .flatMap(_.split(“ ”))

alpha = words.filter(_.matches(“[a-z]+”))

nums  = words.filter(_.matches(“[0-9]+”))
```

```
alpha.count()
```

Here is a very brief introduction to PySpark... We will start with a very simple example with this piece of code:

```
from pyspark import SparkConf, SparkContext
nomappli = "name1"
config = SparkConf().setAppName(nameppli)
config.setMaster("spark://master:7077")
sc = SparkContext(conf=config)
```

- "sc" represents the SparkContext
- It is an object that has several methods including those that create RDDs
- The commented line config.setMaster() allows to define the URL of the Spark Master

Now that the SparkContext is ready, we can create a very simple RDD:

```
nums=sc.parallelize([1,2,3,4])
```

Now we apply a transformation to the data with a lambda function (to compute the square of nums), it is a map transformation:

```
squared = nums.map(lambda x: x*x).collect()
for num in squared:
    print(’%i ’ % (num))
1
4
9
16
```

A more convenient way to process the data with PySpark is to use a DataFrame and use **Spark SQL** (SQLContext is used to initiate the functionalities of Spark SQL):

```
sqlContext = SQLContext(sc)
list_p=[(’John’,19),(’Smith’,29),(’Adam’,35),(’Henry’,50)]
rdd = sc.parallelize(list_p)
rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
sqlContext.createDataFrame(ppl)
list_p = [(’John’,19),(’Smith’,29),(’Adam’,35),(’Henry’,50)]
rdd = sc.parallelize(list_p)
ppl = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
DF_ppl = sqlContext.createDataFrame(ppl)
DF_ppl.printSchema()
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

PySpark: Spark with Python!

Here is the counting words example with PySpark:

```
import sys
from pyspark import SparkContext

sc = SparkContext()
lines = sc.textFile(sys.argv[1])
word_counts = lines.flatMap(lambda line: line.split(’ ’)) \
                   .map(lambda word: (word, 1)) \
                   .reduceByKey(lambda count1, count2: count1 + count2) \
                   .collect()

for (word, count) in word_counts:
    print(word, count)
```

Easier than writing a Java Hadoop MapReduce routine, right?

During next TD:

- We will practice Hadoop Pig
  - Make sure you have pulled the corresponding Docker image (apache-pig)!
- And we will practice PySpark
  - Make sure the Docker Compose application with Spark is working
  - Spend some time to get familiar with the PySpark documentation

PySpark official documentation

The official documentation
http://spark.apache.org/docs/latest/api/python/getting_started/index.html