NOVA IMS

BIG DATA ANALYTICS

NOTES

Anastasia Nica

2021/2022

INDEX

INTRODUCTION TO BIG DATA

Increasing number of data sources Huge volume of new data Difficult for institutions to manage data and researchers to make use of it **Data Deluge**

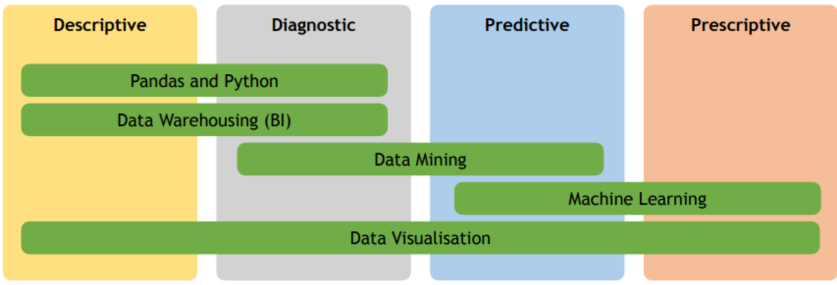
How was data stored before Data Deluge?

Data was stored in **RDBMS** – Relational Database Management Systems -, and the tasks were processed by a **single mainframe**, in a **sequential** fashion.

The increasing demand for storage was dealt through **vertical growth** of CPU, memory, and storage capacity. However, this setup started to **fail** since the early 2000's, with the increasing availability of **new data sources**.

What is Big Data?

1) **Collecting large amounts of data**
    - Need to collect huge amounts of data | **Volume**
    - Need to collect incoming data in real time | **Velocity**
    - Need to collect many different types of data | **Variety**
    - Need to ensure data quality | **Veracity**
2) **Make sense of the data**
    - What happened? | **Descriptive**
    - Why did it happen? | **Diagnostic**
    - What will happen? | **Predictive**
    - How can we make it happen? | **Prescriptive**



Big Data offers:

- Instant knowledge of errors
- Implementation of new strategies on a larger scale
- Improvements in service
- Fraud detection in real time
- Cost savings
- Better sales insights
- Keep up with the customer trends
- Enables data driven and real time decision making
- **A new way to do data science!**

Sources of Big Data

- Large Hadron Collider (CERN Data Centre)
- Astronomy
- Gene sequencing
- Medical imaging
- IoT
- Reality mining
- Wearable computing
- Social media

Applications of Big Data

- Understanding human behavior
- Understanding and targeting customers
- Security and law enforcement applications
- Segmentation and forecast of markets

Big Data can be defined by its two main challenges:

1) How can we store the big volumes of data generated?
2) How can we analyze the data in useful time?

NOSQL DATABASES

Before: SQL (RDBMS) and Flat Files → **Scaling problems** (costly and difficult).

Now: NoSQL Databases and Distributed File Systems → Hadoop HDFS, Google FileSystem.

Distributed Computing

Distributed Computing: Uses multiple computing devices to process tasks. Processing tasks "go" to the data. **Data → CPU**

Parallel Computing: A task is replicated and shared by multiple processing units in the same computer. Two or more calculations happen simultaneously. **Data → CPU**

Concurrent Computing: Several computations are executed concurrently, during overlapping time periods, contrarily to **Sequential Computing**, where one is completed before the next starts.

Grid Computing: Not very efficient. Usually used for academic purposes. Network of **interconnected computers** running special grid computing network software, where one is used as a **server** to handle all administrative duties of the system (mother node).

Big Data Ecosystem

Collection of interconnected solutions that have been developed along the years by many organizations to **tackle problems** that arise in the Big Data Environment. These are **ready-to-use** solutions that **lower the costs and time** of deployment.

Primitive Storage Solutions

- Flat Files
- Hierarchical (parent-child dependencies; helped limit redundant data)
- Network (many-to-many relationships; further improved redundancy)

Relational Databases

- Information is stored in tables
- Relationships between tables represent the relationships between the data
- Storing and retrieving data
- **Transactions:** Collection of actions that make **consistent** transformations of system states
- They are ACID compliant
- ACID = **A**tomicity (a transaction either works or fails as a whole) + **C**onsistency + **I**solation (if transactions are being executed concurrently, the final state should be the same as if they were executed sequentially) + **D**urability (if a transaction is committed, it should remain committed even in case of system failure).

Relational DBMS **can't cope** with the large volumes of data and extremely large number of users.
Big Data requires DBMS that provide:

- Large volumes of Read and Write operations
- Low latency response times (nearly real-time)
- High availability

Big Data Storage Solutions (Distributed Storage Systems)

- **Distributed File Systems**
- **Distributed Databases** (NoSQL): Cluster of computers storing data in a distributed manner, instead of a single database in one machine. No standardized query language

Four characteristics of NoSQL databases:

1) Scalability
2) Cost
3) Flexibility
4) Availability

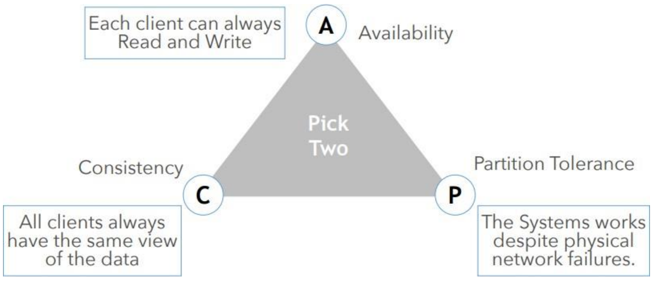
Scaling-up/Vertical Scalability: Moving to a powerful machine or increasing the capacity of the machine. Used by RDBMS/SQL

Scaling-out/Horizontal Scalability: Multiplying the number of machines that are storing the data Used by distributed DBMS/NoSQL

Although **scaling-out is cheaper** than scaling-up because the multiple machines can be cheaper, they are also **less reliable**.

CAP Theorem

All DBMS aim to store data persistently, maintain data availability and ensure data consistency. However, according to the **CAP Theorem**, **only 2 of these are possible at the same time**.



If the system is **always available** and **can tolerate failures**, then **it's not consistent**, because when a user connects to it, it might not receive the latest version of the information it requested.

If the system is **consistent** and **tolerates failure**, then it **can't be available** all the time, because the user has to wait for the last version of the information it requested.

If the system is **consistent** and is **available** all the time (servers are connected), then it **won't tolerate any failures**.

In relational databases, we **don't have a partition problem** because they rely on a **single machine**. With distributed databases, we need to **tradeoff** availability and consistency for partition tolerance.

4 main families of data models in Big Data:

- **Document stores** (*mongoDB*; json documents)
- **Graph databases** (*neo4j*; networks and relationships between entities)
- **Key-value stores** (*redis*; simplest databases; very fast)
- **Wide-column/Multikey Value Maps** (*apache hbase*; one single big table; data can be sparse; keeps track of changes over time; each cell is identified by a row id, a column id and a timestamp)

pH of a database: Ranges from **ACID** (traditional relational DB) to **BASE** (NoSQL databases)

BASE = Basic **A**vailability **+ S**oft-State (stores don't have to be write-consistent nor do replicas have to be mutually consistent all the time) **+ E**ventual Consistency (stores are consistent at some later point, for example at read time)

HADOOP

Hadoop: Distributed File System solution for Big Data, suited for **Analytics**, **open-source** project by Apache Software Foundation. Based on two key Google technologies. Companies that contributed to the development of Hadoop: cloudera, facebook, yahoo!, linkedin.

Hadoop has 3 main components:

- **Hadoop Distributed File System** (HDFS; for storage)
- **Map Reduce** (distributed processing framework; analyze data from the HDFS)
- **YARN** (manage the access to resources of the cluster by the users)

Hadoop Ecosystem

Composed by many more components apart from the 3 mentioned above. The base is the HDFS, for storage, the layer above is for resource management (YARN) and on top we have the tools for analytics.

Hadoop runs on a **cluster** with individual machines represented as **nodes**.

HDFS

- Essentially tries to emulate the behavior of a file system in a **distributed** environment
- Written in Java and based on Google File System
- Sits on top of native filesystems
- Responsible for **storing data** on the cluster
- Data is **split into blocks** and distributed across multiple **nodes** of the cluster (each bl takes up 64MB or 128MB)
- Each block is **replicated** multiple times and stored in **different nodes**, to ensure **tolerance to failures** and **availability**
- Because we're working with a cluster, if we need to **increase capacity**, we can just **add more machines** to the cluster

DataNodes: Where the blocks are stored as standard files, in a set of directories specified in Hadoop's configuration files.

NameNode: Tells us where each block that makes up a file is stored in the cluster. Without it, there's no way to access the files.

When a client application wants to read a file:

1) Communicates with the *NameNode* to determine which blocks make up the file and which *DataNodes* have those blocks.
2) Then, communicates directly with the DataNodes needed.
3) This way, the *NameNode* is never a bottleneck (client retrieves the data directly from the *DataNodes*).

Good for:

- Very large files (but few)
- Streaming data access (read-once, read-many-times)
- Commodity hardware (cheap). HDFS is designed to work around failure, having copies of everything

Bad for:

- Low-latency data access (HDFS is optimized for high throughput of data, at the expense of latency)
- Lots of small files (because data is split into blocks and that requires a lot of memory by itself)
- Multiple writers, arbitrary file modifications (writes can only be made in the end of the file, by a single writer)

Interacting with the cluster can be done using the command line, using Spark (by URI) or using a Java API (e.g., MapReduce, Impala, etc.)

The NameNode must be running at all times. If it stops, the cluster becomes **inaccessible**.

High availability mode:

- Two *NameNodes*
- One Active and one Standby

Classic Mode:

- One *NameNode*
- One "helper" node: *SecondaryNameNode*

MapReduce

Framework to **distribute data processing tasks** across multiple nodes (processing units).

3 steps:

1) Map
2) Shuffle and Sort
3) Reduce

Mapper

- Applies a function to each block of a file
- Read, parse the data and return an object to be aggregated
- The objective is to **prepare the data**, transforming it from the raw format to a format that can be aggregated

- Returns a list of key-value pairs
- Each Map task is executed on the *DataNode* where the block is located

Shuffle and Sort

- Optimization step for the reduce task
- Outputs of the maps are sorted, consolidated and shuffled if necessary

Reducer

- Takes the outputs from the maps and aggregates them as one
- Returns the final output

The various Map tasks are performed in **different machines**, **reducing the processing time**, and then sent to one single machine where all the outputs will be aggregated, producing the final result.

YARN

YARN stands for **Y**et **A**nother **R**esource **M**anager. Tries to assign different machines to different users, so they don't overlap, and manages queues. It contains a **resource manager** and a **job scheduler**. YARN allows multiple data processing engines (top layer) to run on a single Hadoop cluster.

Resource Manager

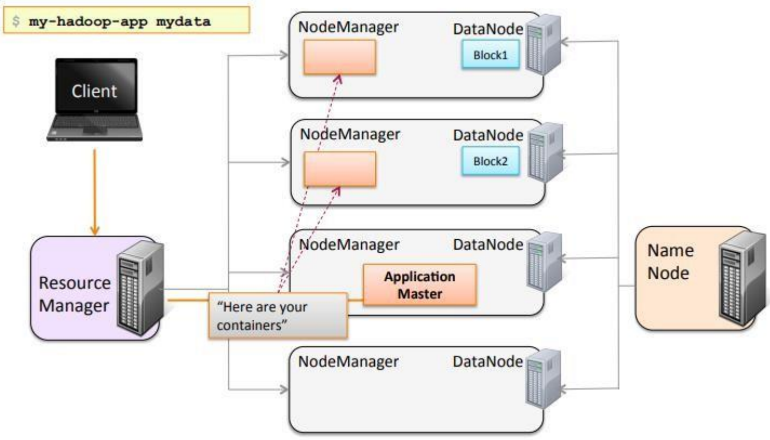
- Runs on a master node
- Global resource scheduler
- Arbitrates system resources between competing applications
- Has a pluggable scheduler to support different algorithms

Node Manager

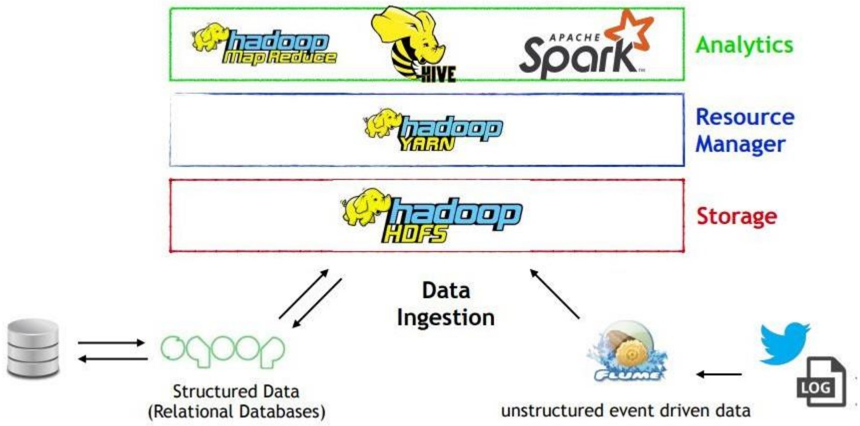
- Runs in slave nodes
- Communicates with the Resource Manager

Client wants to run an application on the cluster:

1) Client communicates with the resource manager
2) Resource Manager allocates a container for the **Application Master** in the cluster, which will run the client application
3) Application Master asks the *NameNode* for the location of the data files.
4) Application Master requests access to those data blocks to the Resource Manager
5) Application Master runs containers on the machines that have the blocks (map and reduce stages, if the reduce stage doesn't happen elsewhere)
6) When it's done, the Application Master contacts the Resource Manager to inform that the process is over.



Hadoop Ecosystem (simplified)



MAP REDUCE AND HIVE

HADOOP APPLICATION EXAMPLES

- **Offload** the stress and minimize the costs of maintaining **traditional pipelines**
  - Data Processing/ETL
  - Data Warehouses
- Being able to **store a large variety of data**
  - Enterprise Data Hub
  - Telemetry
  - 360-degree customer view

Data Processing/ETL offload

- ETL pipelines are complex and not very resilient to changes
- Takes months to make any changes
- Large maintenance costs

- A lot of data ends up being discarded (consumes unnecessary storage and resources)
- Majority of data in DW is dormant
- Difficult to scale (same as RDBMS)

Before: Data sources (OLTP) → **ETL** → **DW** (ELT + Analytic Query & Reporting) → **BI** (OLAP)

After: Data sources (OLTP) → ETL/ELT (**hadoop cluster**) → **DW** (ELT + Analytic Query & Reporting) → **BI** (OLAP)

Advantages of Hadoop:

- **Simplifies ETL pipelines**
- Provides **scalability** for storing data that might be **dormant** right now
- **Reduces the costs** of storing and managing data
- Relies on open-source technologies for storage
- Releases pressure from the DW
- ETL is done just as data needs to be retrieved to the DW
- Does all of this without requiring changes at the input and end-user layers

Data Warehouse Offload

- DW should only store data that has a high value right now (everything else is kept on Hadoop and can be retrieved to the DW when needed)
- DW can't keep up with growing data (multiple storage solutions and data sources)
- Difficult to maintain consistency and combine data from multiple sources

Advantages of Hadoop:

- **Centralized** point where all the data is stored (Hadoop system). From that point, data propagates to other systems
- No conflicts between versions of data
- No more complexity when it comes to collecting data from a variety of sources
- A way of storing data **persistently**

Is MapReduce dead?

- Requires very low-level programming (few people can write efficient MapReduce scripts)
- Nowadays there are more robust and faster solutions (Spark, Hive, Pig)
- There are still companies using this solution, but it's practically dead
- Pig and Hive have an interpreter that translates and optimizes operations into MapReduce tasks

Is Hadoop dead?

- Peaked in 2015/2016
- Apache Hadoop played a big role in the emergence of Big Data
- Not dead but there are many more solutions nowadays, especially in the cloud

- Hadoop is designed to address a specific need: handling large datasets efficiently using commodity hardware
- Other technologies offer more flexible and efficient options
- It's up to you to choose the right technology for your needs

Alternatives to Hadoop:

- Redshift, BigQuery and snowflake are alternatives to Hive
- Spark is good for batch processing
- Flink is great for real-time processing
- Kafka is an ecosystem with its own tools that allows the staging of data coming in real time

Hadoop MapReduce

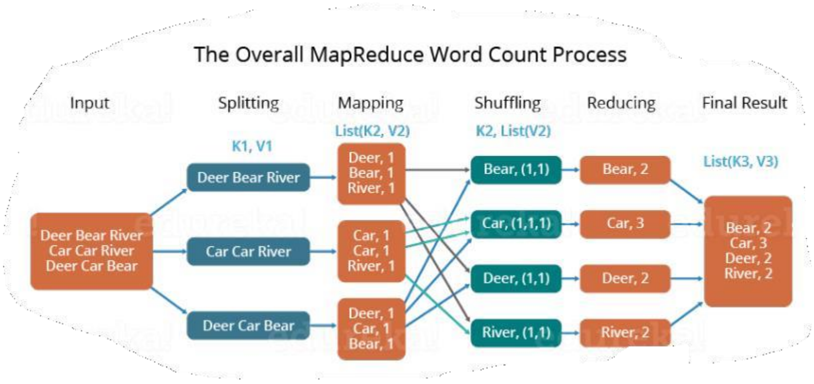
Word Count Problem

Map: each instance of Map works with a block of text. The input is a list of elements, where each element is a line of text. Output is a list of **key-value pairs**, where the key is the element on which we do the aggregation.

- First step is splitting the line into a list of words
- Pre-processing to remove ambiguities, weird characters, punctuation, etc. and make the data as clean as possible

Shuffle & Sort: Data is shuffled, sorted and **grouped by key**. We have a list of values corresponding to singular counts of each word (key).

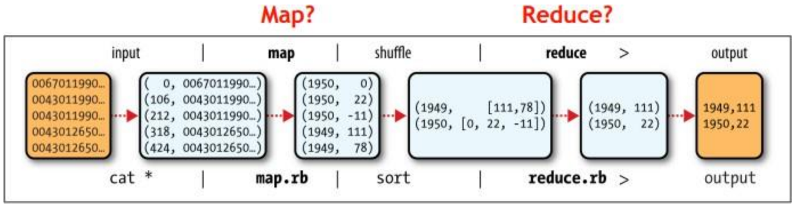
Reduce: Take each key and **apply the function** (in this case, sum).

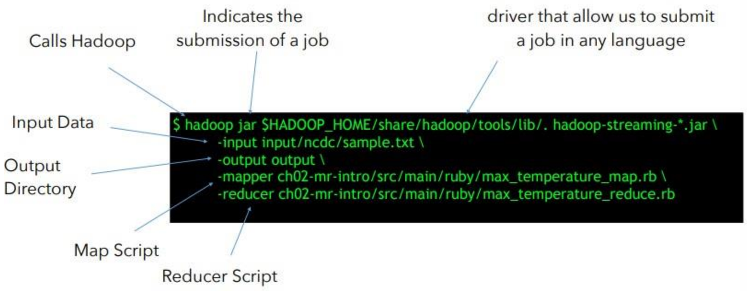


Why do we care about counting words?

- Simple problem to explain
- It's challenging when working with massive amounts of data (number of unique words can easily exceed available memory)
- The statistics used are simple aggregate functions
- MapReduce breaks complex tasks down into smaller elements which can be executed in parallel
- Many common tasks are similar to word count

Anatomy of a Map Reduce Job





Clustering with K-Means and Hadoop MapReduce

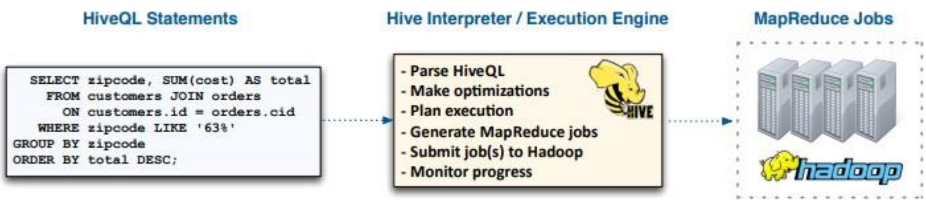
1) Data is distributed across the machines
2) Choose **k initial centroids** at random from the **set X**
3) Apply **k-meansMap** and **k-meansReduce** to X
4) Compute the **new means** (centroids) from the results of k-meansReduce
5) **Broadcast** the new means to each machine on the cluster (in the form of a table with the locations of the centroids)
6) Repeat 2-4 until the means have **converged**

k-meansMap: Computes the distance of each observation to each centroid and keeps track of what is the closest one. The **output** for each observation is a key-value pair, where the key is the closest centroid and the value is the data point and a 1 – (X,1).

k-meansReduce: Computes the **average** of all observations associated with a specific centroid (key) and then **update** all the centroids. Overall, we do two sums.

HIVE

- Alternative analytical framework to work with the data stored in HDFS
- Used to perform SQL queries on data in HDFS
- Uses an SQL-like language called HiveQL
- Does **not** replace RDBMS
- Generates MapReduce jobs that run on the Hadoop cluster
- Originally developed by Facebook for data warehousing, now an open-source Apache project



Why use Apache Hive?

- More productive than writing MapReduce directly (less errors, less code)
- Brings large-scale data analysis to a broader audience (experience with SQL is enough)
- Offers interoperability with other systems (BI tools, extensible through Java and external scripts)

How Hive Loads and Stores data

- Hive Queries operate on **tables**, just like in an RDBMS
  - A table is an HDFS **directory** containing one or more files
  - Hive supports many formats of data storage/retrieval
- The structure and location of tables is specified at creation
- The metadata is stored in Hive's **metastore**
- Hive checks if the format is compatible on read (on write it doesn't complain)

Hive vs RDBMS

Hive doesn't turn your Hadoop cluster into an RDBMS, it simply produces MapReduce jobs from HiveQL queries, so **limitations** of HDFS and MapReduce still apply.

	RDBMS	Hive
Query Language	SQL	HiveQL
Update/Delete Individual Records	YES	NO
Transactions	YES	NO
Index Support	Extensive	Limited
Latency	Very Low	High
Data Size	Terabytes	Petabytes

Interacting with Hive

- **Command-line shell**
- **Web UI**
  - Hive Query Editor
  - Metastore manager
- APIs
  - Open Database Connectivity (**ODBC**)
  - Java Database Connectivity (**JDBC**)

Big Data File Formats

- Avro (good for doing column-level operations)
- Parquet
- Apache ORC (excellent compression)

Choosing the optimal file format provides multiple benefits:

- Faster read times
- Faster write times
- Splittable files
- Schema evolution support
- Advanced **compression** support

Hadoop HDFS vs RDBMS

	Hadoop HDFS	RDBMS
Scalability	Scale-Out	Scale-Up
Data Format	Key-Value pair	Record
Querying Framework	MapReduce/HiveQL	SQL
Schema	Denormalized	Normalized
Data Types	All varieties of data	Structured data
Operations	OLAP/Batch/Analytical queries	OLTP/Real-time/Point Queries

DATA INGESTION

Stages in a Big Data Pipeline

- **Collection**
- **Ingestion**
- **Preparation** (to meet requirements; combine data from multiple sources)
- **Computation** (exploring, analyzing, getting KPIs, training models, clustering, etc.)
- **Presentation** (report results to the team responsible for the production level)

Although Hadoop was a very important technology in the beginning, nowadays the cheapest and most flexible solutions for companies are **on the cloud**. They have many **tools** for each stage in the Big Data pipeline to satisfy the needs of the companies. It's very common for companies now to store their data fully on the cloud instead of on-premises, using distributed file systems.

Examples: AWS, Microsoft Azure, Google Cloud – fully integrated ecosystems with every tool needed for the Big Data pipeline – or the open-source way (using all open-source tools).

Some processing tasks are very expensive, time-consuming, and heavy **Batch processing**
Some things need to be computed on-the-fly, as data arrives **Real-time processing**

Batch processing:

- Series of tasks are sent to the cluster
- Every defined period of time (for example, every 2 days), the data is sent to the analytics engine to perform analytics
- Slower but more complex

Real-time processing:

- Even before being stored, data is used to perform analytics (it is sent to the analytics engine and the database at the same time)
- Just for tasks that require real-time feedback
- Fast and simple analytics

Lambda architecture: Allows for both **real-time** and **batch processing**, depending on the tasks. The decision between saving data for batch processing or using it for real-time processing is done in the **preparation & computation** stages.

It's **difficult** to build pipelines for **ingesting data** that are consistent, easy to maintain and have a low maintenance cost. Because they are really **susceptible to changes** in the data sources or in the endpoint of the pipeline.

Data Ingestion:

- Simply put in or get data from HDFS
- Get transactional data from relational DBs and import it to HDFS (also combine with other types of data)
- Export data from HDFS to relational databases

Structured Data: Sqoop

Open-source tool used to **import data stored in RDBMS to HDFS**, like lookup tables and legacy data. Originally written by Cloudera, but now an Apache project.

It's possible to **read directly** from a RDBMS in the **Mapper**, however, that is not recommended because it can lead to the equivalent of a **DDoS** (Distributed Denial of Service) **attack** on the RDBMS (because data is distributed across multiple machines). This is why Scoop exists.

Functionalities

- Import **tables** from RDBMS into HDFS
  - Just one table
  - All tables in the DB
  - Portions of a table
  - Supports WHERE clause
- Imports data as **delimited text files** or **SequenceFiles**
  - Default is comma-delimited text-file

- Can be used for **incremental data imports**
  - First import retrieves all rows in a table
  - Subsequent imports retrieve only the new rows created after last import
- Uses **MapReduce** to actually import the data
  - Controls the number of Mappers to avoid DDoS attacks
  - Uses 4 mappers by default
- Uses a **JDBC** (Java Database Connectivity) **interface**
- Import the whole database including **metadata**

Why do we care about Sqoop?

- **Streamlines** the process of importing and exporting structured data
- **Optimized** to use the **MapReduce** framework
- Can be used to perform **incremental** updates
- Can be combined with **UI tools** to allow data import/export to **non-technical users**

Unstructured Data: Apache Flume

Problem: Unstructured data comes from a **variety of sources**. Data can have **different specifications**. **Event driven data sources** only produce data when an event is occurring and can spend long periods of time without producing data.

Apache Flume

- **High-performance** system for data collection
- Developed by Cloudera and now an Apache project
- Captures data in **real-time** and stores it elsewhere (creates logs)
- Needs a **scalable enough system** to handle **peaks** of streaming data
- It's **horizontally-scalable**, **extensible** and **reliable**

How does it work?

- Flume **"agents"** collect data from many sources (including other agents)
- **Multiple agents** can be used in large-scale deployments (can be combined in chain)
- Supports **transactions** (once the data is captured, it stays in the channel for as long as it needs to be stored in the sink)
- Flume supports **inspection** and **modification** of in-flight data
- In the end, they **sink** the data (send it to the final destination)
- The **channel** is the element of each agent that **buffers** the data when the source is producing more data than the sink can ingest
- The **interceptors** are small scripts that **modify** the data in the channels

Examples of sources: Syslog, Netcat, Exec, Spooldir, HTTP Source.

Examples of sinks: Null, Logger, IRC, HDFS, HBaseSink.

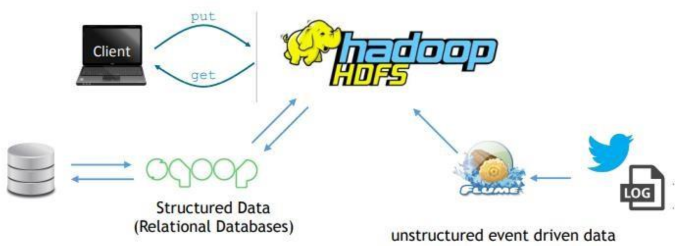
Built-in channels:

- **Memory**
  - Stores events in the RAM

  - Extremely fast, but not reliable (volatile memory)
- [illegible]
  - Stores event on the hard-drive
  - Slower but more reliable
- **JDBC**
  - Stores events in a relational database
  - Still slower than file

Flume Agent Configuration File

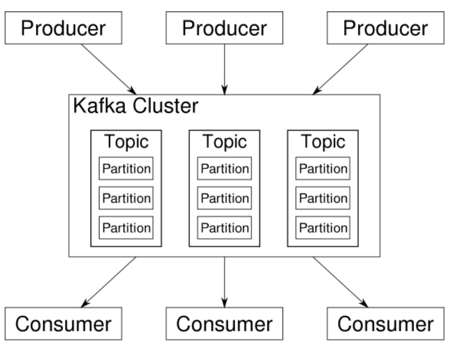
- **Simple** setup
- **Java properties file**, specifying:
  - Agent name (can be multiple agents)
  - Properties of source, channel, and sink
- Uses **hierarchical** references (though a **user-defined ID** for each component)



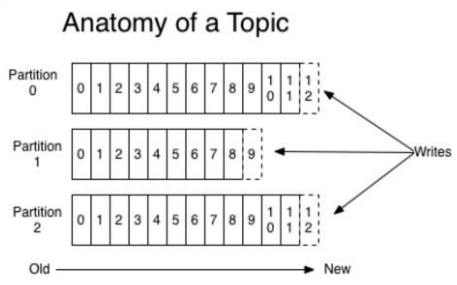
Apache Kafka

- Modern approach for data ingestion
- Originally developed by Linkedin
- **Publish/Subscribe** system (corresponds to Source/Sink, respectively)
- Better way to **buffer and store** data before it is consumed
- Works as **middleware** to ensure the persistence data that has a **limited lifetime**
- Stores data as **sequential logs** and allows **processing** of those logs as they are stored
- Kafka is a **cluster**, instead of an instance of an agent that is running on a machine

Kafka architecture



- **Producers** are the data sources
- **Topics** are a sequence of records distributed across multiple servers (**brokers**)
- Topics are divided into **partitions**, allowing for parallelization
- Partitions work as **logs**
- Each message within a partition has an **offset** that helps consumers keep track of their reading position



- Messages are identified by the topic, **partition** and **offset**
- Records are key-value pairs
- New messages are **appended** to the end of a partition
- **Consumers** read the data from the topics

Kafka has a family of solutions:

- **Kafka Streams** (API for processing streaming data)
- **Kafka KSQL** (real-time data processing of Kafka)
- **Kafka Connect** (provides drivers to connect to different sources and sinks)

Advantages of Kafka:

- Centralizes communication between producers and consumers of data
- Good for streaming data
- Allows a large number of consumers
- Highly available and resilient to failures and supports automatic recovery
- Ideal for large scale data systems

APACHE SPARK

Apache Spark is a **fast** and **general engine** for **large-scale data processing**. The way to interact with Spark is through the **Spark Shell** (python or Scala) or through **Spark applications** for large-scale data processing (python, Scala, R or Java).

Python Interface → **pySpark** (the one we will work with!)

Spark is the state-of-the-art solution for **analytics** in **Big Data**!

How to work with Spark

Spark Context (sc): identifies properties of the cluster being used; every spark program needs a SparkContext.

- **sc.stop** to terminate the program

Fundamental unit of data in Spark

RDDs (Resilient Distributed Datasets): datasets that are distributed in the cluster, so if they are lost in memory they **can be recreated.**

Three ways to create an RDD:

- From a file or set of files
- From data in memory
- From another RDD

sc.parallelize: create RDDs from collections/containers of data (e.g., lists).

sc.textFile: create RDDs from one or more files, using their location (URI).

Two types of RDD operations:

- Actions (return values) – *count(), take(n), collect(), saveAsTextFile()…*
- Transformations (define a new RDD based on the current one) – *map(function), filter(function)*

Properties of RDDs:

- RDDs are **immutable** and we **cannot access** specific elements, like in lists
- **Lazy execution:** Data in RDDs is not processed until an **action** is performed
- If possible, performs sequences of **transformations by row**, so no data is stored, we avoid overflowing the memory and can spot errors early on
- Spark relies heavily on **function** programming (many RDD operations take functions as arguments)
- **Anonymous functions** are defined in-line without an identifier (**lambda** functions)

Two types of transformations:

- Single-RDD transformations – *flatMap, distinct, sortBy*
- Multi-RDD transformations – *intersection, union, zip*

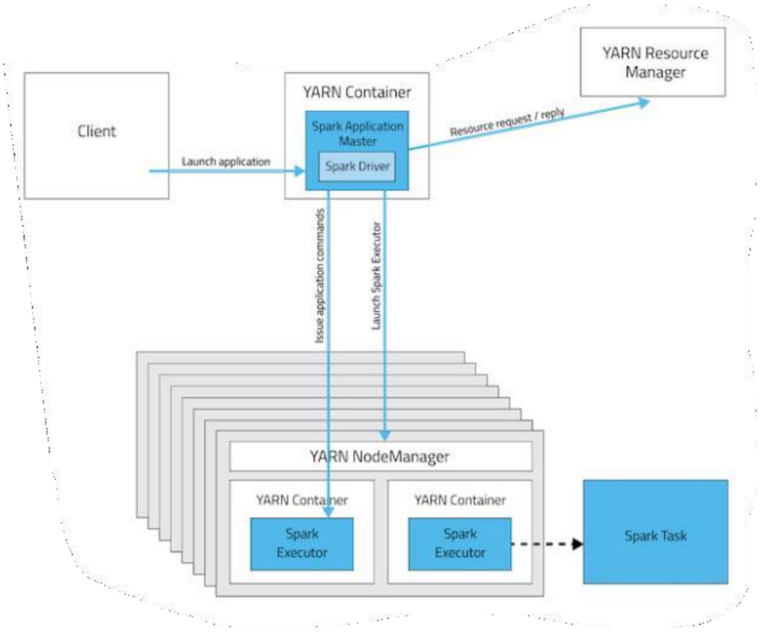
Other actions:

- Sampling operations – *sample, takeSample*
- Double RDD operations – *mean, sum, variance, stdev*

Running Spark Applications

Run a Spark Application: **spark-submit** (command line tool to submit the script to a specific cluster). Spark can run **locally** (for testing) or on a **cluster**.

Running Spark on YARN (Cluster Mode)

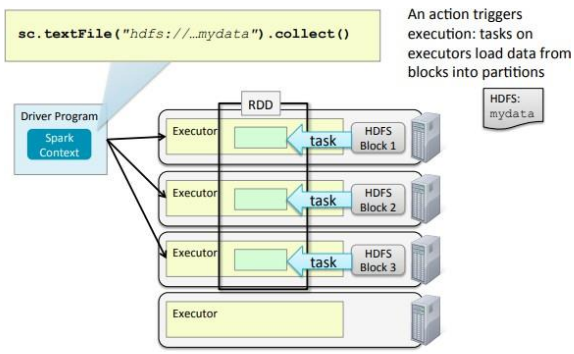


In **client mode**, the client has direct access to the **Spark Driver**, so it's more interactive.

Running Spark on YARN – HDFS Data Locality

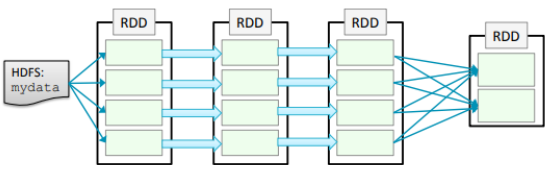
Supposing we have a Hadoop **cluster of 4 machines** and an **HDFS file** split into 3 blocks.

1) Driver Program (master node) controls all the executers (each executer runs locally on a single machine)
2) Run an operation to load the data into an RDD
3) Tasks on the executers loads the data **from the blocks into partitions** of the RDD
4) Since we ran an action, apart from loading the data, the **output is sent back** to the Driver Program



Sometimes, it's **not possible to keep** the partitions of the RDD on the **original** machines that we created the RDD on. So, we have to **shuffle** the RDD and **move the data** into other machines. This is a very expensive task.

```
> avglens = sc.textFile(file) \
     .flatMap(lambda line: line.split()) \
     .map(lambda word: (word[0],len(word)) \
     .groupByKey()
```



In the case above, we need to create a **new RDD** with only **two partitions** (data needs to be moved around the machines), because of the **groupBy**.

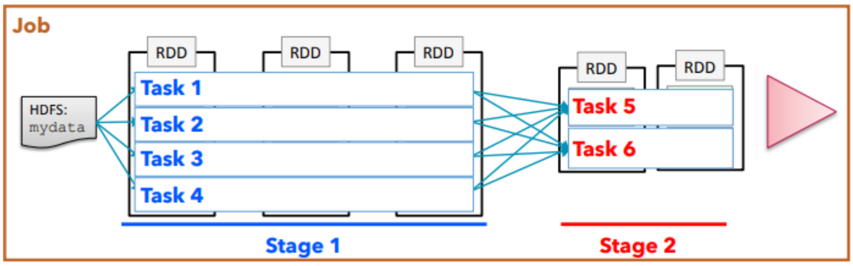
Anatomy of a Spark Job

Job: Set of tasks executed as a result of an action.

Stage: Set of tasks in a job that can be executed in parallel (using the same executers).

Task: Individual unit of work sent to one executer.

Application: Can contain any number of jobs managed by a single driver.



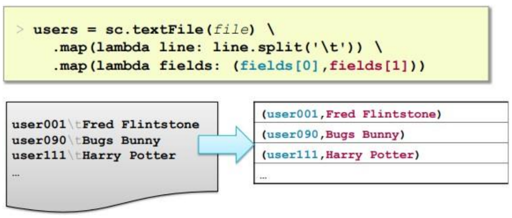
Two big families of transformations

- **Narrow Transformations**
    - Happen within the same stage
    - A new RDD can be mapped into the first RDD
    - Each partition of the RDD has a 1 to 1 map
    - Very fast
    - Don't require shuffling
- **Wide transformations**
    - Require a reshuffling of the data
    - 1 to N map of the RDD partitions
    - Slower than narrow transformations
    - Associated with reduce tasks (aggregations)

Pair RDDs

RDD where each **element** is a **key-value pair**. This is useful for **map-reduce** algorithms.

First step: Put the data in key-value format. This can be done through many transformations, such as: *map, flatMap, keyBy.*



Spark implements **map-reduce** with much **greater flexibility**, because map and reduce functions can be chained to other operations and the results can be stored in memory.

Map phase: Operations at record-level.

Reduce stage: Look at the map output and do some aggregations using multiple records.

Word Count Example

1) *flatMap*: One word in each element of the RDD
2) *Map:* Create key-value pairs
3) *reduceByKey:* Sums values (v1+v2) whenever it encounters matching keys

Other Pair RDD operations:

- countByKey
- groupByKey
- sortByKey
- join

Spark SQL

- Spark module for structured data processing, built on top of **core Spark**
- It provides a **DataFrame API** (data is represented in DataFrames instead of RDDs) and a set of **tools** to work against those DataFrames
- It runs a compiler – **Catalyst Optimizer** – that interprets the queries and **compiles** them into **RDD operations**
- Similarly to core Spark, it requires a context object – **SQL Context** or **HiveContext** (working with an Hadoop cluster running Hive)
- SQL Context is created based on the SparkContext
- **DataFrames can be created:**
  - From a structured data file
  - From an existing RDD
  - By performing an operation or query on another DataFrame
  - By defining a schema
  - From a database
- **Basic operations:** *schema, printSchema, cache/persist, columns, dtypes, explain*
- **Queries:** create a new DataFrame
  - DFs are immutable
  - Queries are similar to RDD transformations
  - Queries can be chained like transformations
- **Actions:** return data to the Driver
  - **Examples:** *collect, take(n), count, show(n)*
- Spark SQL supports **SQL queries**
  - First, register the DataFrame as a table
  - Then, use *sqlCtx.sql()* to write an SQL query
- DataFrames are **built on RDDs**
  - Base RDDs contain **row objects**
  - Use *.rdd* to get the underlying RDD
- **Row RDDs** have all the standard Spark **actions** and **transformations**
- Row RDDs can be transformed into **Pair RDDs** to use map-reduce methods

- Spark SQL is most useful for ETL or incorporating structured data into other applications

Spark SQL is especially useful when working with:

- Large amounts of data
- Distributed storage
- Intensive computations
- Distributed computing
- Iterative algorithms
- In-memory processing and pipelining

Spark GraphX

- Create a representation of data as a **graph**
    - Social networks
    - Web page hyperlinks
    - Roadmaps
- Work with graph data to extract measures
- Provides **graph creation**, **representation**, **analysis** and **post-analysis processing**

Spark MLib

- Provides a set of comprehensive **machine learning models**
    - Clustering
    - Classification
    - Regressions
    - Collaborative filtering (recommendations)

Databricks

Cloud provider that creates an **implementation of Spark** that is easy to connect to data stored on the cloud. It's a paid service. Comes included with Azure.

It runs on top of a file system – **Databricks File System (DBFS)**, instead of HDFS.

It connects to different storage layers and facilitates the processing of data.