Large-scale machine-learning

Lesson 3 — Parallelized computing on distributed data

Arthur Katossky & Rémi Pépin

ENSAI (Rennes, France)

January 2020

Forewords

Course outline

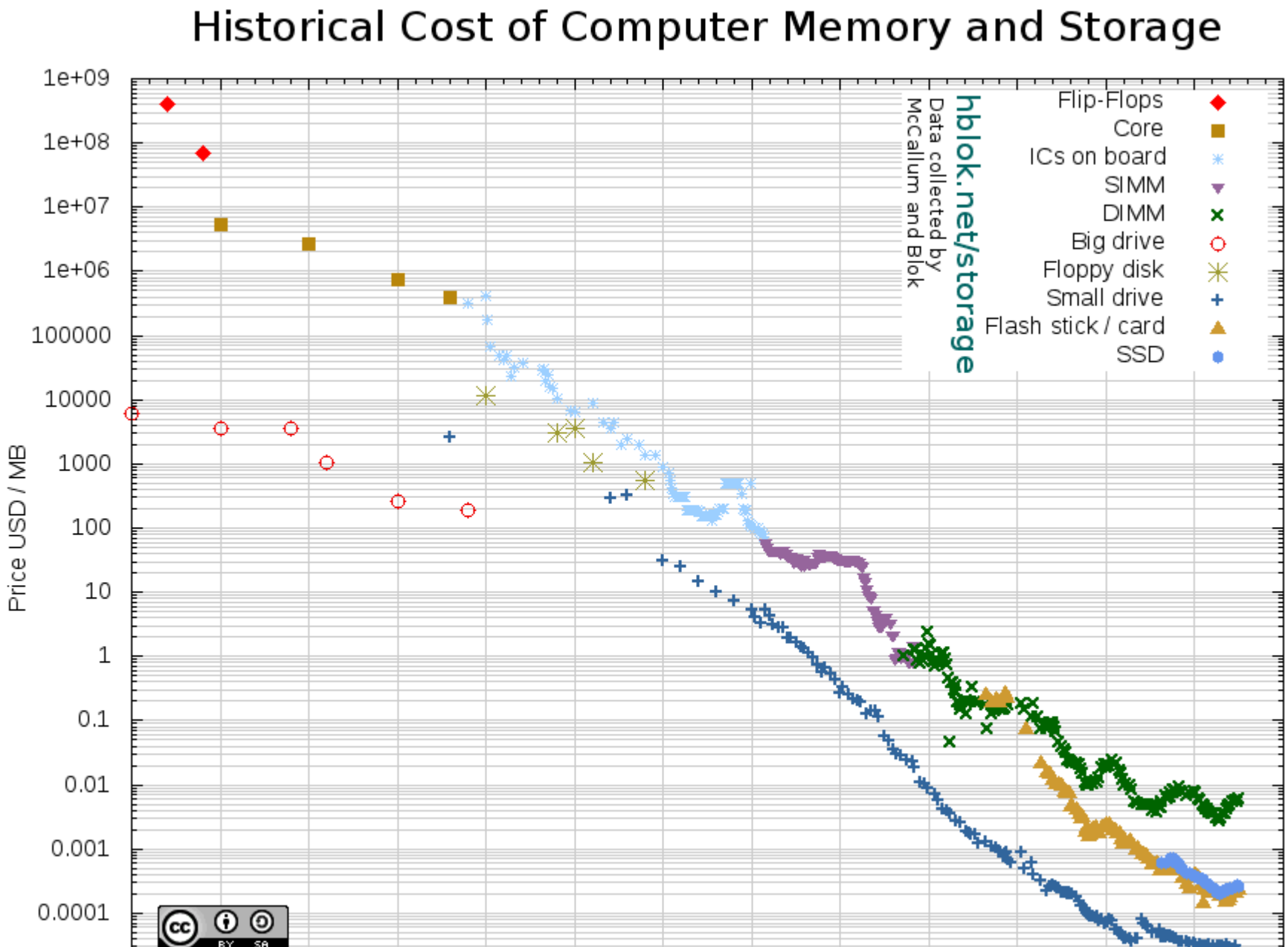
1. Refresher `<= Today`
2. File systems vs. databases
3. The fundamental problems of distribution
4. Distributing file systems
5. Distributing databases
6. Distributing tasks `<= Next week`
7. Parallelizing computation on distributed data
8. Statistical applications
9. Conclusions and perspectives

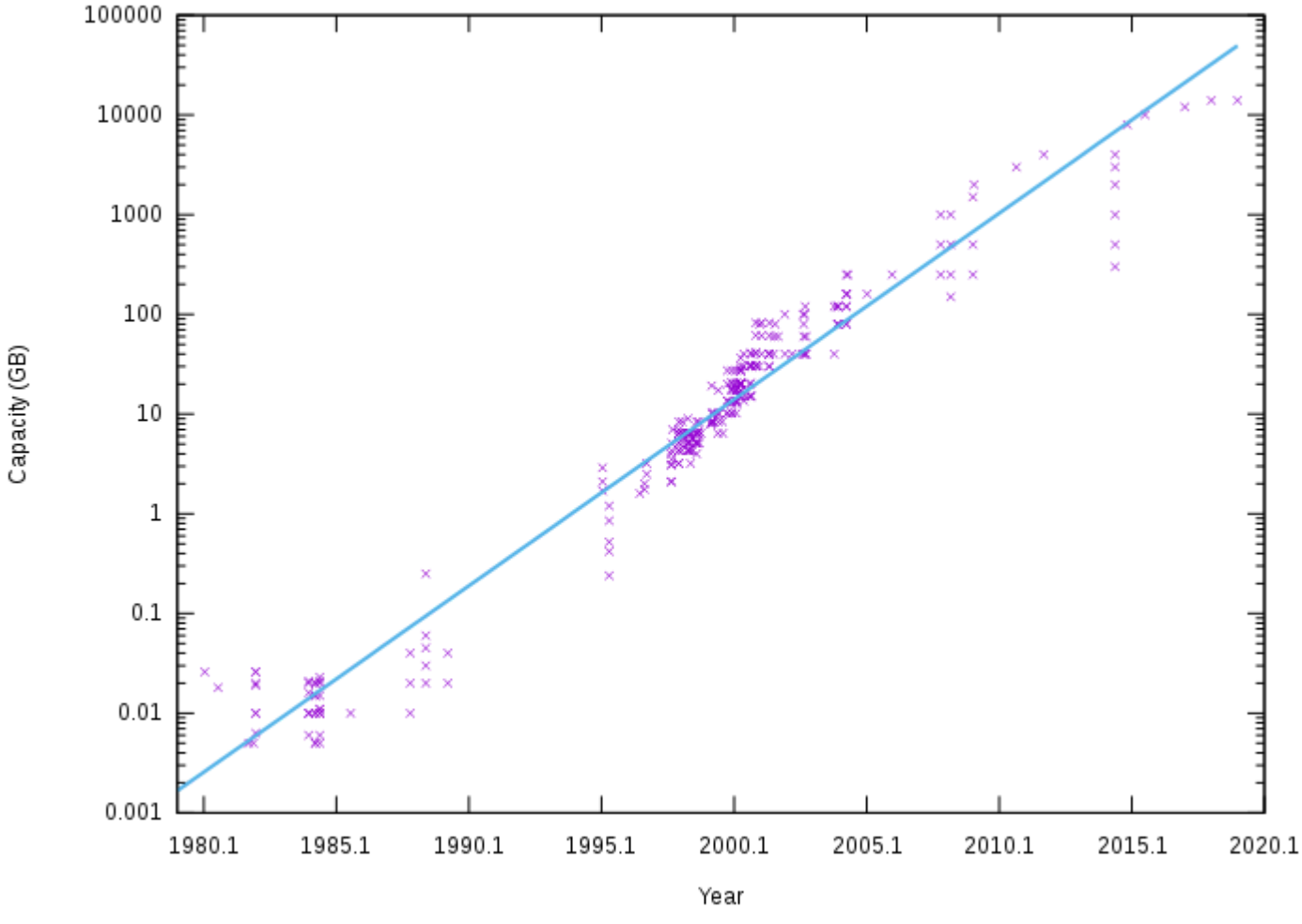
QCM

Go on Moodle. You have 5 mins.

1. Refresher

Storage improvement and limits


Historical Cost of Computer Memory and Storage
Price USD / MB
1e+09
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0.1
0.01
0.001
0.0001
hblok.net/storage
Data collected by
McCallum and Blok
Flip-Flops
Core
ICs on board
SIMM
DIMM
Big drive
Floppy disk
Small drive
Flash stick / card
SSD


100000
10000
1000
100
10
1
0.1
0.01
0.001
Capacity (GB)
1980.1
1985.1
1990.1
1995.1
2000.1
2005.1
2010.1
2015.1
2020.1
Year

Parallelisation

Cloud computing

2. File systems vs. databases

File systems vs. databases

Goal: what do we do when data exceeds the physical limits of our storage?

-> We first have to understand how things are stored in the normal case.

File systems

- Manages files on storage space (persistant memory, i.e. most often hard disk)
  - Makes the "physical file" (bits) and the "logical file" (fragments) transparent for the user
  - Does not care about file formats
  - Can handle *heterogeneous*, *unstructured* data (image, sound, text, application)
- Principal use cases :
  - Read
  - Write
  - Security (access rights)

File systems

Traditionnal file processing

File Systems

Key features

- Is responsible for the correspondance between the physical and logical file
- Maintains the **index** of where data is situated on disk
- Maintains the **namespace**, a virtual hierarchy of files and directory making it possible to identify one single file by a string, called a **path**
- Doesn't undestand file contents, only understands meta data (e.g. size, owner, time stamp of last change)
- Responsible of the integrity of files
- Responsible of the redundancy of files (*not needed on your personal computer*)
- Security (access rights)

Provides an useful abstraction for the clients (=programmes). It's not the programm responsability to know how files are stored (physical location, redundancy, access right). But it's the programm responsability to know how to read files.

Data bases

- Manages files on storage space (persistant memory, i.e. most often hard disk)
  - Makes the "physical file" (bits) and the "logical file" (fragments) transparent for the user
  - Possibly care about data format (numbers, texts, dates...)
  - Can handle *homogeneous*, *structured* data
- Principal use cases :
  - Unique entry point to access data
  - Knows how to read/write data
  - Exposes new tools to manipulate data
    - Specific lanquage (SQL for instance)
    - Transactions and concurrency management
    - Models to organize data

Data bases

Why use a data base ?

- Integrated tool to process data
- "Centralize" the data

Data bases

Data base processsing

Data bases

Data base transaction

- Data base responsibility to guarentee the coherence and validity of data
- **Transaction** : unit of work
  - **A**tomicity : complete entirely or not at all
  - **C**onsistency : changes affected data only in allowed ways
  - **I**solation : must no affect other transcations
  - **D**urability : changes must be written on persistant storage

3. The fundamental problems of distribution

Are the problems fundamentally different for file systems and databases?

The distinction becomes blur, especially when you start giving up on the *relational* aspect of databases.

> The difference between a distributed file system and a distributed [database] is that a distributed file system allows files to be accessed using the same interfaces and semantics as local files.

Such semantics include using folder-like hierarchy, attributing permissions to individual files, copying files to different locations, etc.

We will thus start by the *common* problems.

The fundamental problems of distribution

- **Availability:** you want your data available 24/7
  - **Latency**: you don't want to wait for hours to get answered (read and write)
  - **Throughput:** maybe you want to read ALL your data or write GB at once
  - **Fault tolerance:** a single node failure shouldn't make your data unavailable
- **Coherence:** the same query with the same data return the same result
  - **Atomicity*:** complete entirely or not at all
  - **Durability*:** your data mustn't get corrupted over time
  - **Inter-node consistency:** at the same time, all the node see the same data
- **Schema consistency*:** changes affected data only in allowed ways
- **Isolation*:** transactions mustn't overlap
- **Elasticity/Scalability:**(under constraint of constant / acceptably-increasing request time)
  - can it store more or bigger files ?
  - can it process more requests ? (reads / writes)
  - can I add more nodes ? (*horizontal scaling*)
  - does scaling require a lot of *ad hoc* work? can the scaling happen automatically? (*elasticity*)

The fundamental problems of distribution

Plus the usual questions of large-scale systems :

- **Confidentiality:** only authorized person can access your data
- **Data gouvernance:** under which law are your data ?
- **Environment:** is your system not too big ? Does your system use too much energy ?
- **Economy:** isn't it too expensive to manage?
- ...

The fundamental problems of distribution

Some solutions

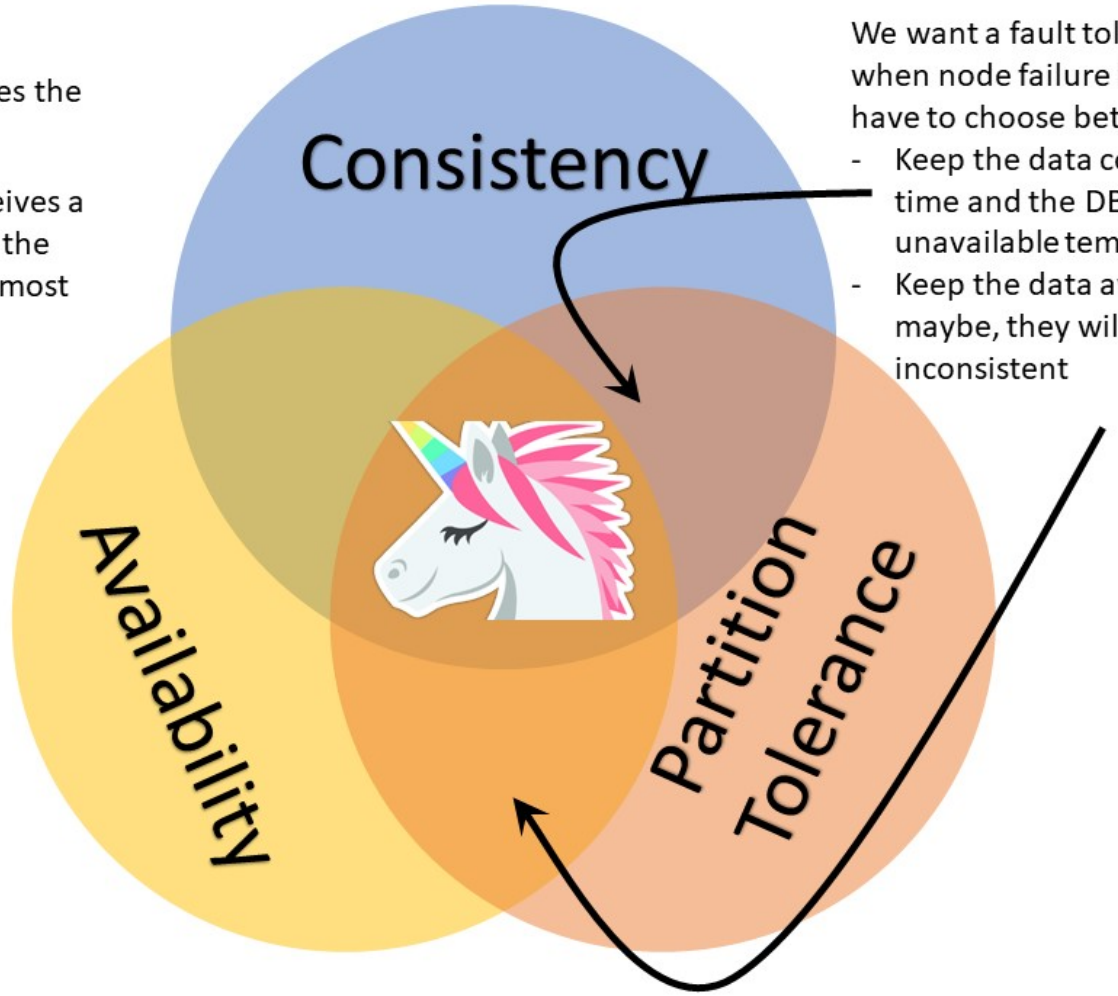
- **Redundancy / replication:** keep copies of the data in far away nodes, so that you don't lose information under hardware failure (++ availability, ++ fault-tolerance, ++ durability, -- inter-node consistency, -- schema-consistency, --cost, --environment)
- **Balancing\rebalancing:** use all your node fairly (--availability now, ++ availability later, ++ scalability)
- **Timestamp-based concurrency control:** use timestamp to resolve conflict (first in first out) (++isolation, --availability)
- **Get the closest data to the client:** if the data are close to the client, there is less network time (++ availability, - inter-node consistency, - governance)
- **Have a master:** it organizes the work to avoid conflict (+ consistency, - availability, - fault-tolerance)
- **Asynchronous processing:** nodes can accept change locally, and consolidate the transactions only in a second phase (++ availability, --inter-node consistency)
- **First-class actions:** you may chose to priviledge reads over writes, or to completely prevent modifying files, for instance (++ availability, ++ consistency)

The CAP theorem

Consistency: Every read receives the most recent write or an error

Availability: Every request receives a (non-error) response, without the guarantee that it contains the most recent write

Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes



We want a fault tolerant system, when node failure happen, we have to choose between

- Keep the data consistent. Takes time and the DB will be unavailable temporary
- Keep the data available. But maybe, they will be inconsistent

Transparency requirements

- **Access transparency:** clients are unaware that files/data are distributed and can access them in the same way as local files are accessed

- **Location transparency:** the way to refer to a file/data does not dependon its location
- **Concurrency transparency:** all clients should have the same view of the state of the file system / database
- **Failure transparency:** clients should not notice a single node failure
- **Heterogeneity and scale transparency:** it should not matter on which specific machines or on how many machines the file system / the database is distributed
- **Replication transparency:** clients should be unaware of the file replication performed across multiple servers to support scalability
- **Migration transparency:** files should be able to move between different servers without the client's knowledge.
- ...

Souce: https://en.wikipedia.org/wiki/Clustered_file_system#Distributed_file_systems

4. Distributing file systems

Distributing file systems

Distributed file systems are also known historically as **network file systems**

Dates back from the beginning of "the network" in the 1960s.

In 1985, Sun Microsystems creates "Network File System" (NFS), still in use today.

An exemple of distributed file system: HDFS

HDFS stands for "Hadoop Distributed File System."

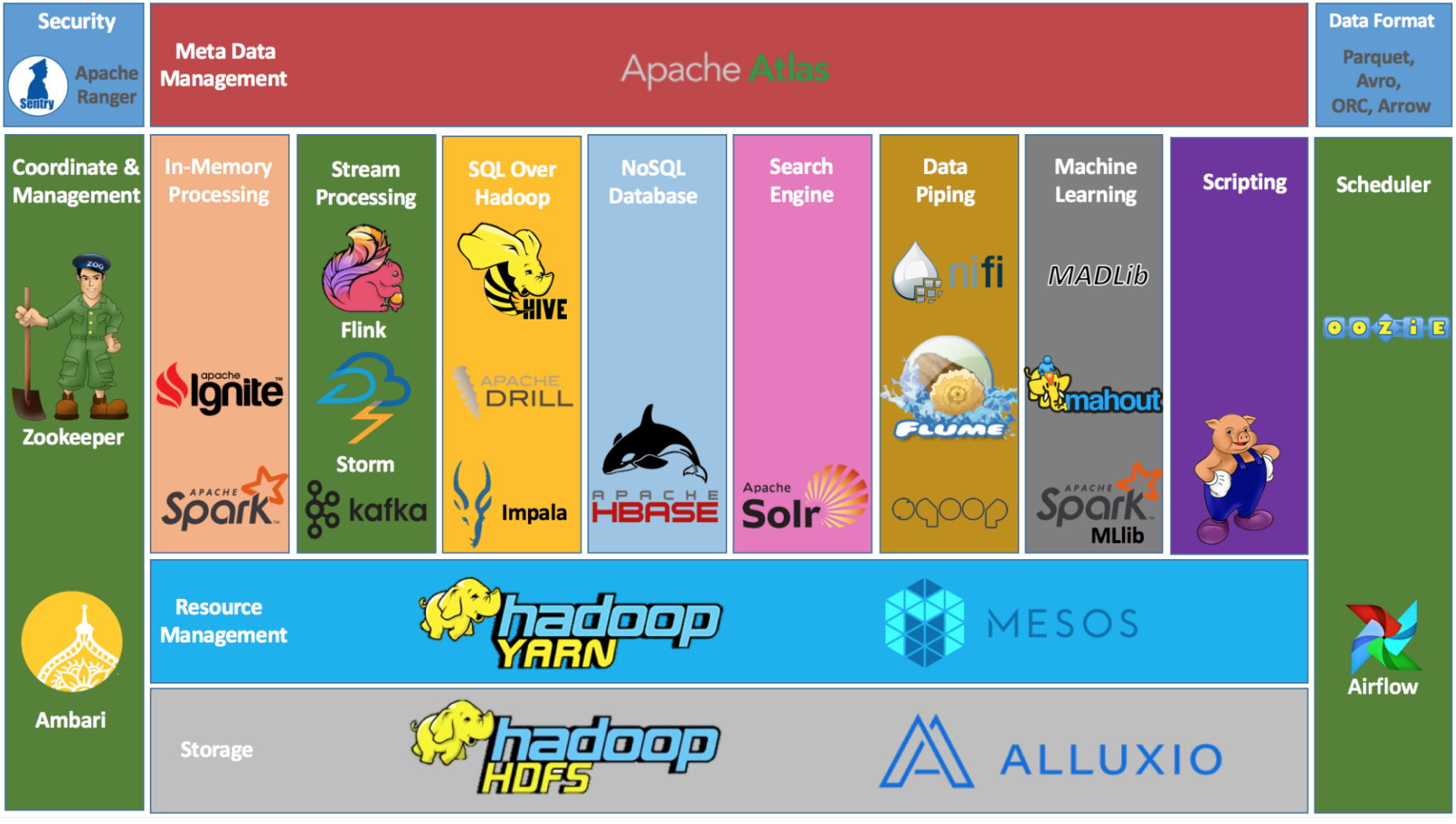
It is an open-source project financed by the Apache Fundation.

This section is heavily inspired from HDFS documentation pages (link). Asterisks (*) denote (almost) exact citations.

An exemple of distributed file system: HDFS

Hadoop is actually a complete suite of *modules*, of which HDFS and YARN are the basic components.

But "*Hadoop*" in a broader sense refers to a complete software ecosystem, most of which is also supported by the Apache fundation. This ecosystem encompasses the *Hodoop* modules MapReduce, Ozone and Submarine, and the libraries Ambari, Avro, Cassandra, Chukwa, HBase, Hive, Mahout, Pig, Spark, Tez and ZooKeeper. More information on Hadoop's website.



Source: https://www.oreilly.com/library/view/apache-hive-essentials/9781788995092/e846ea02-6894-45c9-983a-03875076bb5b.xhtml

An exemple of distributed file system: HDFS

Hadoop was first released in 2006, and has evolved a lot since.

HDFS's developement was inspired by the publication of *Google File System*, a now deprecated file system developped by Google, to serve as infrastructure for the *Apache Nutch* web search engine project.

We will here focus on the latest version, Hadoop 3.

Hadoop is mostly coded in *Java*.

An exemple of distributed file system: HDFS

Why HDFS?

- open-source
- well-documented
- well-spread

An exemple of distributed file system: HDFS

The architecture

HDFS has a master/slave architecture.

NameNode: a master server that manages the file system namespace and regulates access to files by clients*

DataNodes: slaves which manage storage attached to the nodes that they run on*

An exemple of distributed file system: HDFS

Key ideas

When given a new file, the **NameNode** splits it into one or more **blocks** and gives these blocks to be stored into a set of **DataNodes**. Each block is stored several times, on different DataNodes, and this number is the **replication factor** of that file.*

Files in HDFS cannot be modified, except for appends and truncates. Emphasis is indeed on the reading part, a scheme HDFS calls "write-once-read-many".

By default, the replication factor is 3, and the NameNode tries to allocate the replicas intelligently: one on a given node, that then sends a copy to a close node (faster but less fault-tolerant) and an other to a further away node (slower but more fault-tolerant).

An exemple of distributed file system: HDFS

Key ideas

If a client wants to write a file, they ask the NameNode.

1. The NameNode splits the file into blocks.
2. For each block, it selects a number of DataNode to write onto (typically 3), based on:

   a. disk space (more space is better) b. proximity to the client (closer is better) c. proximity to each other (one replication close, one far) d. distribution of blocks (blocks of the same file should be on different nodes)

3. It then passes the block split and the lists to the client.
4. The client writes each block on the first block on the list.
5. The DataNode passes the block and the list on to the next DataNode on the list. (This minimizes the use of the network in between the cluster and the client, likely to be slower than the network inside the cluster.)

The "client" is often not directly the user, but some other module asking for read / write access.

An exemple of distributed file system: HDFS

Key ideas

If a client wants to read a file, they ask the NameNode.

1. The NameNode looks into the index to associate the file path to blocks.
2. The NameNode locates DataNodes containing the blocks, closest to the client.
3. It passes the information to the client, who in turn reads directly from the specified DataNodes.

The "client" is often not directly the user, but some other module asking for read / write access.

An exemple of distributed file system: HDFS

Role of the master (nodename)

The **NodeName** :

- is the entry point for a client's requests
- decides how to split files into blocks and on which DataNodes to store these blocks
- never handles actual files, only stores metadata
- knows at any time the correspondance between a file's name and the block's identifiers (aka **namespace**, stored on disk) and the the physical location of blocks (aka **block index**, kept in memory)
- detects DataNode failure by listenning to their **heartbeat** and demands block replication as necessary

An exemple of distributed file system: HDFS

Role of the master (nodename)

The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.*

The **namespace** is persisted to disk at regular intervals (every X seconds or every Y changes). In between, a record of the changes is also written onto disk, in a file know as the edit log, so that at all time, the entire file system is preserved. In a case of a failure of the master, the data on disk is restored.

Where exactly the blocks are stored, however, is not stored on disk, but kept in memory.

An exemple of distributed file system: HDFS

Role of the slaves

The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.*

DataNodes:

- emit a regular **heartbeat** containing the list of all the blocks stored locally
- send copies directly to each other when the NameNode requires a copy to be made
- give access (in read or write) directly to the client

An exemple of distributed file system: HDFS

Properties

Fault-tolerance

> The primary objective of HDFS is to store data reliably even in the presence of failures. The three common types of failures are NameNode failures, DataNode failures and network partitions.

An exemple of distributed file system: HDFS

Properties

Fault-tolerance

1. Hadoop modules have **location awareness**, i.e. they use the locations of the nodes relative to each other. This enables HDFS to obtain **safe redundancy** by replicating data in different locations. By default, there is one copy in vicinity (typically in a server room, the same rack) and one copy in further away (typically, an other rack).

An exemple of distributed file system: HDFS

Properties

Fault-tolerance

1. Failure is explicitly taken into account in the design of the file syste. There may be 3 kinds of failure:
   - **NameNode failure:** when the NameNode fails, it is restarted, and:
     - restores the latest index saved on disk,
     - applies all the changes that are recorded in the edit log, also saved on disk
     - reconstitute the location of blocks in memory from the DataNodes' heartbeat

   Running several NameNodes with a distritbuted edit log is also possible.

   - **Network failure:** when a subset of DataNodes lose connectivity with the NameNode (aka *network partition*), the NameNode detects the absence of *heartbeat* and immediately demands replication of blocks below the replication factor. Since replication happens in distinct locations, it is unlikely that all of the replicated blocks become unavailable. (By default, a DataNode is considered dead after 10 min of silence.)
   - **DataNode failure:** this is a special case of *network partition* with only one node disconnected from the rest

An exemple of distributed file system: HDFS

Properties

Availability

1. **Latency / throughput** is achieved by delegating to the slave all the communication intensive work. Yet, the NameNode is a bottleneck, since all communication goes through it. **High throughput** is obtained at the expense of a (relatively) **low latency**. HDFS is not conceived for interactive use.

2. HDFS supports natively **balancing**: it will use first the least used resources.

3. HDFS is also compatible with *rebalancing*.

An exemple of distributed file system: HDFS

Properties

Scalability

1. By splitting files into blocks, HDFS does not limit the size of a single file, nor the number of files.
2. You can add slave servers at any time.
3. Users interact only briefly with the NameNode, and most of the interaction with the cluster is decentralized, hence enabling a large amount of simultaneous users.

However, HDFS is not meant for frequent writes, and does not scale in this regard. Indeed, even though data does not go through the DataNode, editting the index — which is, please remember, copied onto disk — and demanding copies to be made to the DataNodes take time. Multiple NameNode can cope with this but is not the standard installation. HDFS calls it "write-once-read-many".

An exemple of distributed file system: HDFS

Properties

Integrity

HDFS insures integrity by storing the ***checksum*** of a block alongside the block itself. If the block gets corrupted, the computed checksum does not match the stored one. (It is possible, but much less likely, that the checksum itselft gets corrupted.)

Clients checksum the block they download and inform the DataNode of the result. DataNodes also run regular checsksum of all unchecked blocks.

DataNodes store information about the checks already made, and thus can narrow the date about which data has become corrupted.

In case of checksum mismatch, clients flag the incriminated block as corrupted to the NameNode, in which case a new copy is demanded.

Other distributed file systems

- Proprietary private file systems such Amazon S3, Azure Storage, Google Cloud Storage, etc.
- **NFS** (Linux), **SMB** (Windows) or **AFP** (Apple), typically used in Network-attached storages (NAS's)
- **Lustre**, typically used in clusters of super computers
- ...