Data intensive Computing

Introduction :

The main trends:

- Computers not getting any faster
- Internet connections getting faster
- More people connected to the Internet

Conclusion: move the computation and storage of big data to the cloud!

Cloud computing :

Cloud Computing refers to both:

1. The applications delivered as services over the Internet
2. The hardware and systems software in the datacenters that provide those services

A computer system is divided into two categories: Hardware and Software. Hardware refers to the physical and visible components of the system such as a monitor, CPU, keyboard and mouse. Software, on the other hand, refers to a set of instructions which enable the hardware to perform a specific set of tasks.

The services: called Software as a Service (SaaS)

The datacenter hardware and software is called cloud

cloud has Five characteristics, Three service models ,Four deployment models

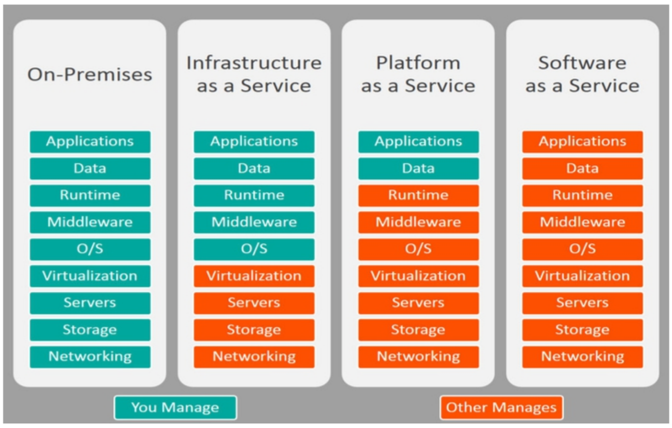
characteristics : on-demand (self-service), ubiquitous (omniprésent) network access, rapid elasticity (can grow if needed), mesured service pay per use (service depend de l'utilisateur), location transparent ressource pooling (Mutualisation des ressources).

Service models : Infrastructure as a Service (IaaS): similar to building a new house.

Platform as a Service (PaaS): similar to buying an empty house.

Vendor provides hardware and development environment

Software as a Service (SaaS): similar to living in a hotel.



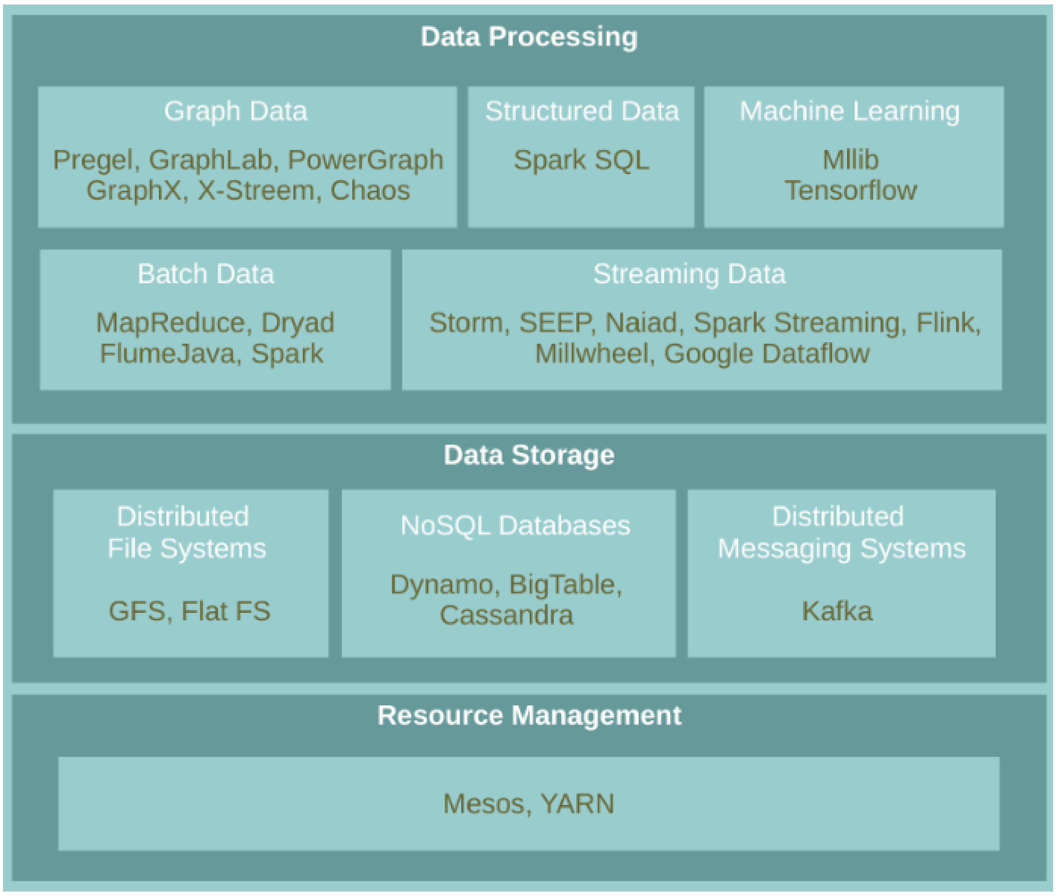
Deployement models : 1) Public Cloud, 2) Private Cloud, 3) Community Cloud, and 4) Hybrid Cloud (part private, part public)

Big data is the data characterized by 4 key attributes: volume data size, Velocity: data generation rate, Variety: data heterogeneity and Value.

Scale up or scale vertically: adding resources to a single node in a system, more expensive.

Scale out or scale horizontally: adding more nodes to a system, more challenging for fault tolerance and software development

The Big Data stack (all different part/object/component)



- Resource Management :

  Manage resources of a cluster (group), Share them among the platforms

- Data Storage :

  Distributed File Systems : Store and retrieve files on/from distributed disks, manage the storage across a network of machines

  NoSQL Databases : (BASE database) The difference between ACID and BASE database models is the way they deal with this limitation. **The ACID model provides a consistent system. The BASE model provides high availability**.

Messaging Systems : Store streaming data

– Data Processing (manipulation of data by a computer) :

Streaming Data : data that is generated continuously by thousands of data sources

Linked Data (Graph) : linked data is structured data which is interlinked with other data so it becomes more useful through semantic queries, Graph-parallel processing model (traitement parallele) , Vertex-centric and Edge-centric programming model (centré sur arrete et sommets)

Structured Data : Take advantage of schemas in data to process

Machine Learning :

Lecture 2 : Data storage : Distributed systeme file

a File System Controls how data is stored in and retrieved from storage device.

When data outgrows the storage capacity of a single machine: partition it across a number of separate machines.

Google File System (GFS)

Motivation and Assumptions : Huge files (multi-GB)

Most files are modified by appending to the end (Random writes (and overwrites) are practically non-existent)

Optimise for streaming access

Node failures happen frequently (**hardware failures)**

Files are split into chunks.

Chunk: single unit of storage.

- Immutable and globally unique chunk handle
- Transparent to user
- Each chunk is stored as a plain(simple) Linux file

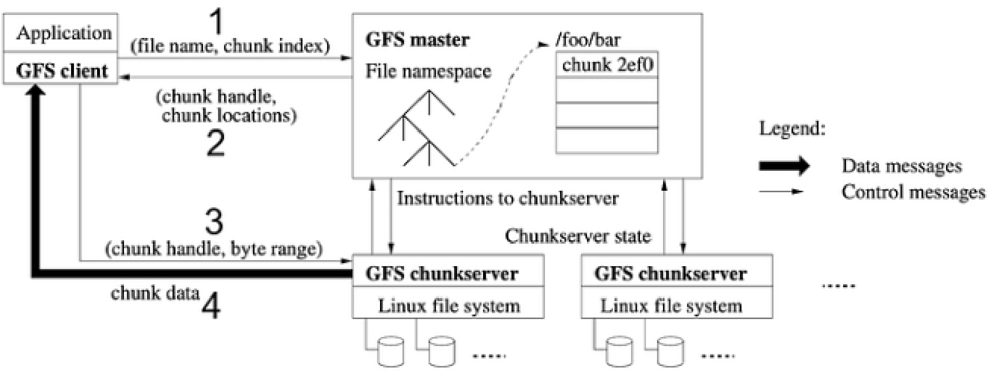


Figure 1: GFS Architecture

to read : The 4 steps illustrate a simple read operation. In step 1, the client asks the master for **metadata**: where are the replicas and locations where my data is stored? The metadata is returned in step 2, and then in step 3, the client asks for one of the replicas (usually the closest one) for the actual **data**, which is transferred in step 4.

Every client has to interact with the master. If the master started forwarding data to clients, bandwidth would be used up quickly, and the system couldn't serve many requests. By returning metadata, interactions with the master are quick, while interactions with chunkservers are longer (which is fine, since clients are probably talking to different chunkservers).

GFS master :

Responsible for all system-wide activities (activité à l'échelle du systeme)

kept in memory : Namespaces, ACLs (access control lists), mappings from files to chunks, and current locations of chunks and his replicas in operation log (used when a failure append to recover the state)

Periodically communicates with each chunkserver

- Determines chunk locations
- Assesses state of the overall system

While it may seem unsafe to have a single master, the decision was a practical one: it simplified the design and was much faster to build and roll out to production than a **distributed master**.

GFS Client : Issues control requests to master server

Issues data requests directly to chunkservers.

Caches metadata. Or Does not cache data.

GFS Chunkserver : Manages chunks, Tells master what chunks it has, Stores chunks as files, Maintains data consistency of chunks

Large Chunks :

advantages : Reduces the size of the metadata stored in master and clients' need to interact with master

disadvantages : Wasted space due to internal fragmentation

possibility of the interface : create, delete, open, close, read, and write (not poxis-compliant (norm))

snapshot: creates a copy of a file or a directory tree at low cost

append: allow multiple clients to append data to the same file concurrently

Consistency means that replicas will end up with the same version of the data and not diverge.

To ensure the consistency : For this reason, for each chunk, one replica is designated as the primary. The other replicas are designated as secondaries. Primary defines the update order. All secondaries follow this order.

Master selects a chunkserver and grants it lease for a chunk. At any time, at most one server is primary for each chunk. If master does not hear from primary chunkserver for a period, it gives the lease to someone else. (each chunk has only one chunkserver)

To write : 1. Application originates the request. 2. The GFS client translates request and sends it to the master. 3. The master responds with chunk handle and replica locations. 4. The client pushes write data to all locations. Data is stored in chunkserver's internal buffers. 5. The client sends write command to the primary. 6. The primary determines serial order for data instances in its buffer and writes the instances in that order to the chunk. 7. The primary sends the serial order to the secondaries and tells them to perform the write.

Write Consistency : Primary enforces one update order across all replicas for concurrent writes. It also waits until a write finishes at the other replicas before it replies.

Therefore:

- We will have identical replicas.
- But, file region may end up containing mingled fragments from different clients: e.g., writes to different chunks may be ordered differently by their different primary chunkservers
- Thus, writes are consistent but undefined state in GFS.

To append : 1. Application originates record append request. 2. The client translates request and sends it to the master. 3. The master responds with chunk handle and replica locations. 4. The client pushes write data to all locations. 5. The primary checks if record fits in specified chunk.

6. If record does not fit, then the primary:

- Pads the chunk (rempli ce qui est possible),
- Tells secondaries to do the same,
- And informs the client.
- The client then retries the append with the next chunk.

7. If record fits, then the primary:

  - Appends the record,
  - Tells secondaries to do the same,
  - Receives responses from secondaries,
  - And sends final response to the client

The Master operations

Namespace Management and Locking (1/2) : Represents its namespace as a lookup table mapping pathnames to metadata. Read lock on internal nodes, and read/write lock on the leaf.

(when you write a file in a repository you can read all the repository when write only on your file)

Replica placement :Maximize data reliability, availability and bandwidth utilization.

Replicas spread across machines and racks, for example:

- 1st replica on the local rack.
- 2nd replica on the local rack but different machine.
- 3rd replica on a different rack.

The master determines replica placement.

Replica Creation

- Place new replicas on chunkservers with below-average disk usage.
- Limit number of recent creations on each chunkserver.

Re-replication

- When number of available replicas falls below a user-specified goal.

Rebalancing (rééquilibrage)

- Periodically, for better disk utilization and load balancing.
- Distribution of replicas is analyzed

Garbage Collection :File deletion logged by master, renamed to a hidden name with deletion timestamp. When a hidden file is removed, its in-memory metadata is erased.

Stale Replica Detection : Need to distinguish between up-to-date and stale replicas. To do so, we use Chunk version number:

- Increased when master grants new lease on the chunk.
- Not increased if replica is unavailable.

Fault Tolerance:

for chunks : chunks replications , checksum checked every time an application reads the data , (checksum is **a small-sized block of data derived from another block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage**.)

for chunkserver : chunk has Version number updated when a new lease is granted, old versions are not served and are deleted

for master : Master state replicated for reliability on multiple machines.

When master fails:

- It can restart almost instantly.
- A new master process is started elsewhere.

Shadow (not mirror) master provides only read-only access to file system when primary master is down.

Lecture 3

Atomicity • All included statements in a transaction are either executed or the whole transaction is aborted without affecting the database.

Consistency : A database is in a consistent state before and after a transaction.

Isolation : Transactions can not see uncommitted changes in the database.

Durability : Changes are written to a disk before a database commits a transaction so that committed data cannot be lost through a power failure.

SQL database were not designed to be distributed, Internet-scale data size, High read-write rates

NoSQL Avoids:

- Overhead of ACID properties
- Complexity of SQL query

Provides:

Scalablity (évolutilité en fonction des données)

Easy and frequent changes to DB

- Large data volumes

Availability : Replicating data to improve the availability of data, Storing data in more than one site

Strong consistency • After an update completes, any subsequent access will return the updated value.

Eventual consistency : Does not guarantee that subsequent accesses will return the updated value.

If no new updates are made to the object, eventually all accesses will return the last updated value. (so it can take more time)

Partition tolerance means that **the cluster must continue to work despite any number of communication breakdowns between nodes in the system**.

BASE properties : Base Availability : posibility of fault but not fault of the whole system, Soft-state : copy of data may be inconsistent, Enventual consistency

NoSQL database :

key value : Collection of key/value pairs and Ordered Key-Value: processing over key ranges.

Column-Oriented Data Model : Similar to a key/value store, but the value can have multiple attributes (Columns) Store and process data by column instead of row (more efficient)

Document Data Model Similar to a column-oriented store, but values can have complex documents.

Graph Data Model : Uses graph structures with nodes, edges, and properties to represent and store data.

Big Table : NoSQL database Column-Oriented Data Model, CAP: strong consistency and partition tolerance

Data Model :

Distributed multi-dimensional sparse map (several table are used when they become too large)

Each tablet is served by exactly one tablet server.

Rows sorted in lexicographical order.

Column families: group of (the same type) column keys.

Each column value may contain multiple versions.

System structure :

It has 3 main components : master, tablet server, client library (a bit like GFS)

Master : Assigns tablets to tablet server, Balances tablet server load, Garbage collection of unneeded files in GFS, Handles schema changes, e.g., table and column family creations

Tablet Server : Can be added or removed dynamically, Each manages a set of tablets (typically 10-1000 tablets/server), Handles read/write requests to tablets and Splits tablets when too large.

Client Library : Library that is linked into every client, Client data does not move though the master, Clients communicate directly with tablet servers for reads/writes.

To save data, big table uses GFS , Big Table is a overlay that provides use structure table, we can then use query to get the data.

Chubby (building blocks for bigtable) : Ensure there is only one active master, Store BigTable schema information and access control lists, Store bootstrap location (adresse of table) of BigTable data, know which tablets are available (memory of the master)

Sstable : Each SSTable is stored in a GFS file, one tablets is cut in several sstable

Utilisation : The master executes the following steps at startup:

Grabs a unique master lock in Chubby, which prevents concurrent master instantiations.

Scans the servers directory in Chubby to find the live servers.

Communicates with every live tablet server to discover what tablets are already assigned to each server.

Scans the METADATA table to learn the set of tablets. (position)

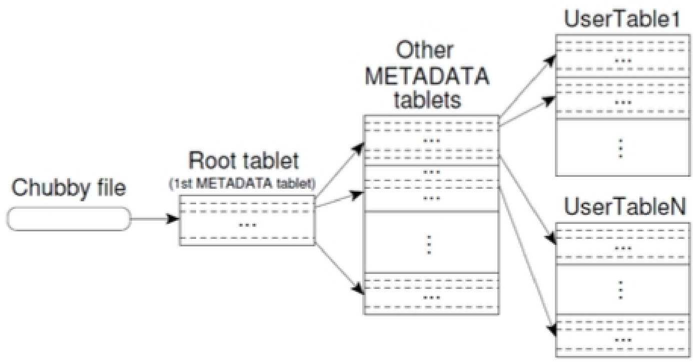
- ## Tablet Assignment

1 tablet → 1 tablet server.

Master uses Chubby to keep tracks of live tablet serves and unassigned tablets.

Master detects the status of the lock of each tablet server by checking periodically, Master is responsible for finding when tablet server is no longer serving its tablets and reassigning those tablets as soon as possible.

- ## Finding a Tablet



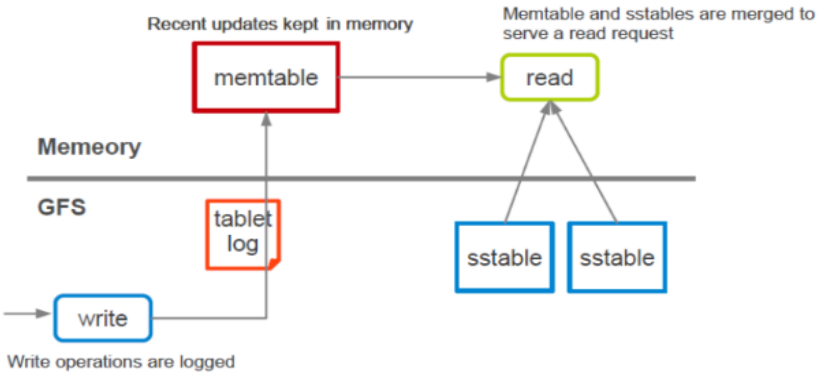
The first level is a file stored in Chubby that contains the location of the root tablet.

Root tablet contains location of all tablets in a special METADATA table.

METADATA table contains location of each tablet under a row.

The client library caches tablet locations.

- Tablet Serving

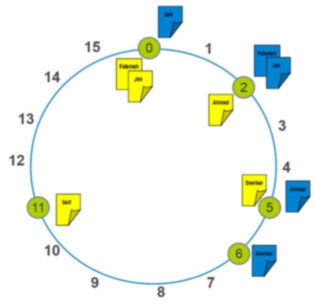


To write, if autorize the tablet log stores it in the memtable. When a user asks to read the sstable, the memtable and sstables are merged to serve a read request.

- Strong consistency : Only one tablet server is responsible for a given piece of data. Replication is handled on the GFS layer.
- Trade-off with availability
  - If a tablet server fails, its portion of data is temporarily unavailable until a new server is assigned.

Cassandra : A column-oriented database, created for Facebook and was later open sourced, who borrows from bigtable has sstable but no master, CAP: availability and partition tolerance

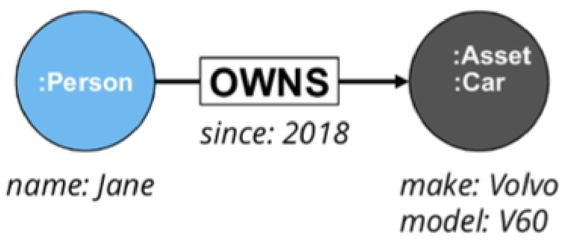
use Consistent hashing to partion data, it means using a hash function who with his Ids give a number who represent his position on a circle, the position of a given server, then to know where is store a data or a replicated data (to achieve high availability and durability) you use the same hash function and to take the closest node clockwise.



Gossip-based mechanism: periodically, each node contacts an other randomly selected node.

Neo4j :A graph database, The relationships between data is equally important as the data itself, CAP: strong consistency and availability, Cypher: a declarative query language similar to SQL, but optimized for graphs

Data Model : Node (Vertex) A waypoint along a traversal route Relationship (Edge), Label : Define node category (optional), Can have more than one Properties :Enrich a node or relationship

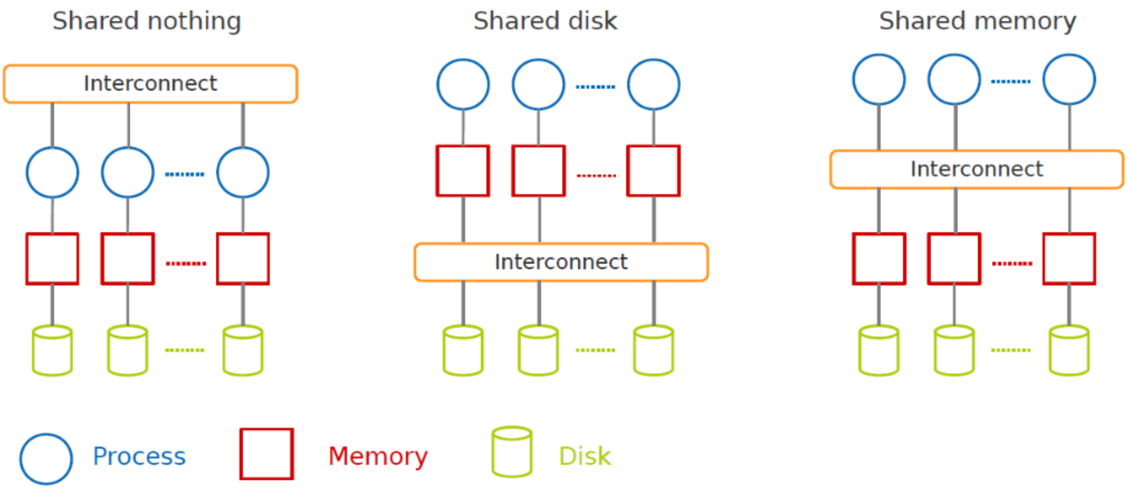


Lecture 5

Operating system : what knows where the data is store

what do we do when there is too much data ? ---> scale up or scale out

Parallel Architectures



MapReduce is a shared nothing architecture for processing large data sets with a parallel/distributed algorithm on clusters of commodity hardware.

Challenges : How to distribute computation?

How can we make it easy to write distributed programs?

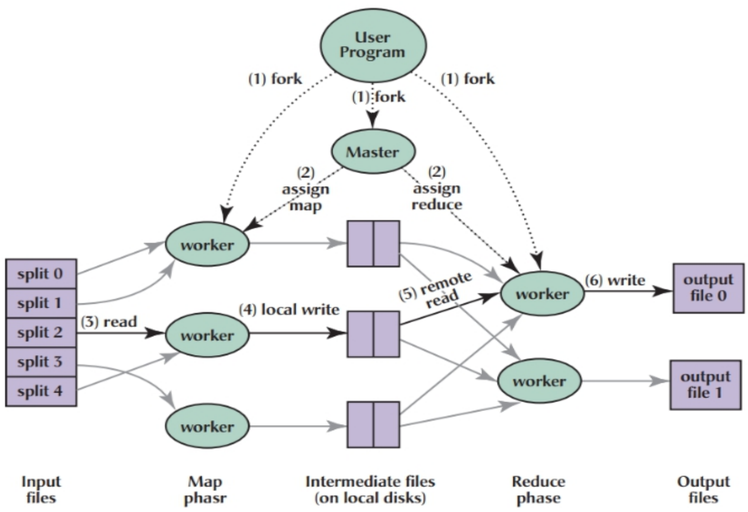
Machines failure.

MapReduce takes care of parallelization, fault tolerance, and data distribution.

Hide system-level details from programmers.

An execution framework: to run parallel algorithms on clusters of commodity hardware.

A programming model: to batch process large data sets (inspired by functional programming).



- The user program divides the input files into M splits.
  - A typical size of a split is the size of a HDFS block (64 MB).
  - Converts them to key/value pairs.

- It starts up many copies of the program on a cluster of machines.
- One of the copies of the program is master, and the rest are workers.
- The master assigns works to the workers.

  It picks idle workers and assigns each one a map task or a reduce task.
- A map worker reads the contents of the corresponding input splits.
- It parses key/value pairs out of the in put data and passes each pair to the user defined map function.
- The key/value pairs produced by the map function are buffered in memory.
- The buffered pairs are periodically written to local disk.
  - They are partitioned into R regions (hash(key) mod R).
- The locations of the buffered pairs on the local disk are passed back to the master.
- The master forwards these locations to the reduce workers.
- A reduce worker reads the buffered data from the local disks of the map workers.
- When a reduce worker has read all intermediate data, it sorts it by the intermediate keys.
- The reduce worker iterates over the intermediate data.
- For each unique intermediate key, it passes the key and the corresponding set of intermediate values to the user defined reduce function.
- The output of the reduce function is appended to a final output file for this reduce partition.
- When all map tasks and reduce tasks have been completed, the master wakes up the user program.

Fault Tolerance - Worker

? Detect failure via periodic heartbeats.

? Re-execute in-progress map and reduce tasks.

? Re-execute completed map tasks: their output is stored on the local disk of the failed machine and is therefore inaccessible.

? Completed reduce tasks do not need to be re-executed since their output is stored in a global filesystem.

Fault Tolerance - Master

? State is periodically checkpointed: a new copy of master starts from the last checkpoint state.

Reduce-side join

- Repartition join
- When joining two or more large datasets together

Map-side join

- Replication join
- When one of the datasets is small enough to cache

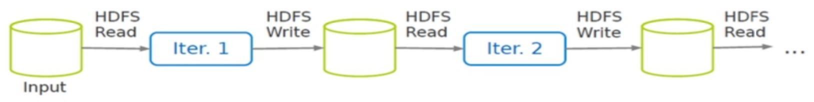
Lecture 6

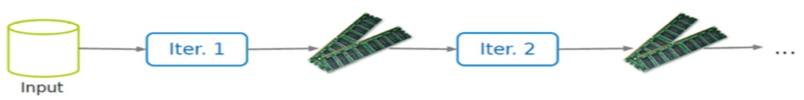
for cyclic data flow from stable storage to stable storage, Mapreduce is good, but if there is more layer there is better library (acyclic data flow)

MapReduce is expensive (slow), i.e., always goes to disk and HDFS.

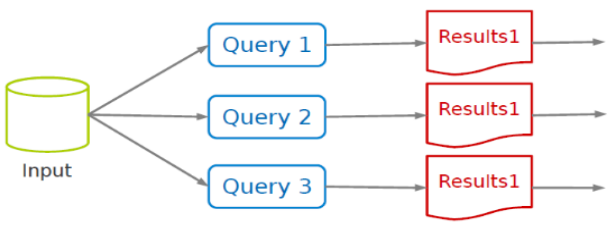
===> Spark

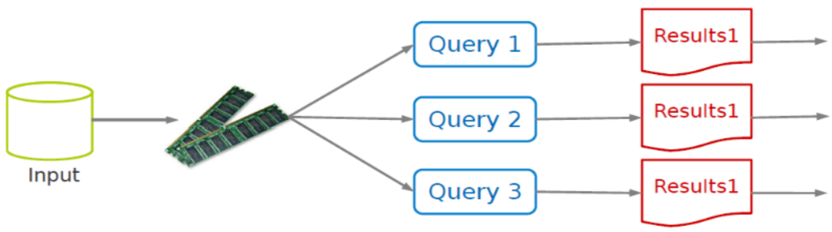
instead of writing into disk each time spark use the memory.



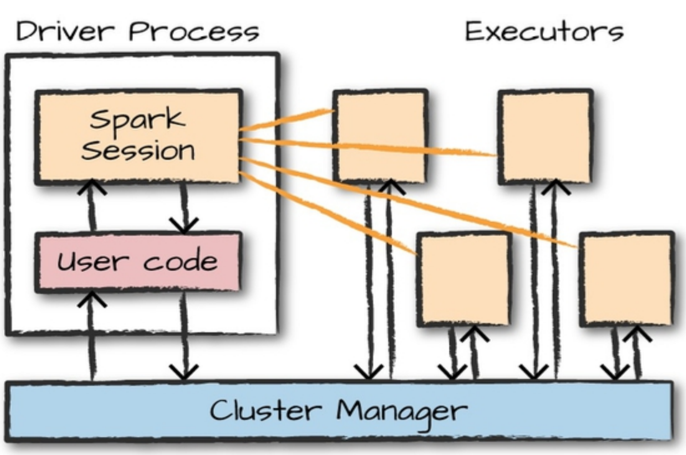


Instead of reading to the file each time we can use the memory (just one read)





Spark architecture



The heart of a Spark application, Runs the main() function

Responsible for three things:

- Maintaining information about the Spark application
- Responding to a user's program or input
- Analyzing, distributing, and scheduling work across the executors

Executor : Executing code assigned to it by the driver and Reporting the state of the computation on that executor back to the driver

To use spark application you need a sparksession. If you start from scrach you need to create it :

`SparkSession.builder.master(master).appName(appName).getOrCreate()` , if not you can use « spark » (predefini)

For low-level API functionality, you can use SparkContext, to create it :

```
val conf = new SparkConf().setMaster(master).setAppName(appName)
new SparkContext(conf)
```

if not you can use « sc » (predefini)

sparksession can do what sparkcontext does

The primary datatype : RDD (Resilient Distributed Datasets)

it is an Immutable collections (list) of objects spread across a cluster

An RDD is divided into a number of partitions, which are atomic pieces of information.

Partitions of an RDD can be stored on different nodes of a cluster.

Two types of RDDs:

- Generic RDD
- Key-value RDD (table)

To create a RDD, you use the use the parallelize method on a SparkContext.

Ex : `val numsCollection = Array(1, 2, 3)`

```
val nums = sc.parallelize(numsCollection)
```

ex from a textfile doc : `val myFile1 = sc.textFile("file.txt")`

RDDs support two types of operations:

- Transformations: allow us to build the logical plan (lazy operation, receive rdd as input and create rdd as output)
- Actions: allow us to trigger (declancher) the computation

exemple of transformations : filter returns the RDD records that match some predicate function.

distinct removes duplicates from the RDD.

map and flatMap apply a given function on each RDD record independently.

sortBy sorts an RDD records.

To make a key-value RDD:

- map over your current RDD to a basic key-value structure.

Ex :`val words = sc.parallelize("take it easy, this is a test".split(" "))`

`val keyword1 = words.map(word => (word, 1)) // (take,1), (it,1), (easy,,1), (this,1), (is,1), (a,1), (test,1)`

- Use the zip to zip together two RDD.

Ex : `val numRange = sc.parallelize(0 to 6)`

`val keyword3 = words.zip(numRange) // (take,0), (it,1), (easy,,2), (this,3), (is,4), (a,5), (test,6)`

- keys and values extract keys and values, respectively.

```
val words = sc.parallelize("take it easy, this is a test".split(" ")) val keyword = words.keyBy(word => word.toLowerCase.toSeq(0).toString)
// (t,take), (i,it), (e,easy,), (t,this), (i,is), (a,a), (t,test)
```

```
val k = keyword.keys
val v = keyword.values
```

- mapValues maps over values.

`val mapV = keyword.mapValues(word => word.toUpperCase) // (t,TAKE), (i,IT), (e,EASY,), (t,THIS), (i,IS), (a,A), (t,TEST)`

Aggregate the values associated with each key : groupByKey or reduceByKey and join

groupBykey create a new RDD but each key is only present ones and has a list of the previous values.

In groupByKey, each executor must hold all values for a given key in memory before applying the function to them. But, in reducebykey do it while ..

ex join : `val joinedChars = kvChars.join(keyedChars) // (t,(1,4)), (t,(1,4)), (t,(1,4)), (t,(1,4)), (t,(1,4)), (h,(1,6)), (,,(1,9)), (e,(1,8)), ...`

Exemple of action : collect returns all the elements of the RDD as an array at the driver.

first returns the first value in the RDD.

take returns an array with the first n elements of the RDD(Variations:takeOrdered and takeSample)

count returns the number of elements in the dataset.

countByValue counts the number of values in a given RDD.

max and min return the maximum and minimum values, respectively.

reduce aggregates the elements of the dataset using a given function. (the given function should be commutative and associative so that it can be computed correctly in parallel.)

saveAsTextFile writes the elements of an RDD as a text file. • Local filesystem, HDFS or any other Hadoop-supported file system.

saveAsObjectFile explicitly writes key-value pairs.

An RDD that is not cached is re-evaluated each time an action is invoked on that RDD.

There are two functions for caching an RDD:

- cache caches the RDD into memory
- persist(level) can cache in memory, on disk, or off-heap memory

Checkpoint saves an RDD to disk.

Ex : `val words = sc.parallelize("take it easy, this is a test".split(" "))`

```
sc.setCheckpointDir("/path/checkpointing")
words.checkpoint()
```

Two types of transformations :

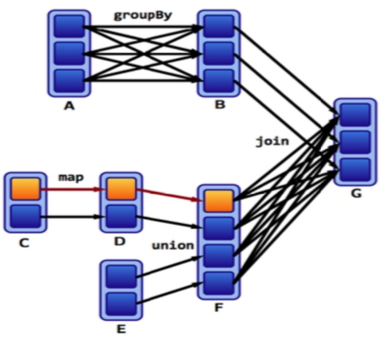
Narrow transformations (dependencies)

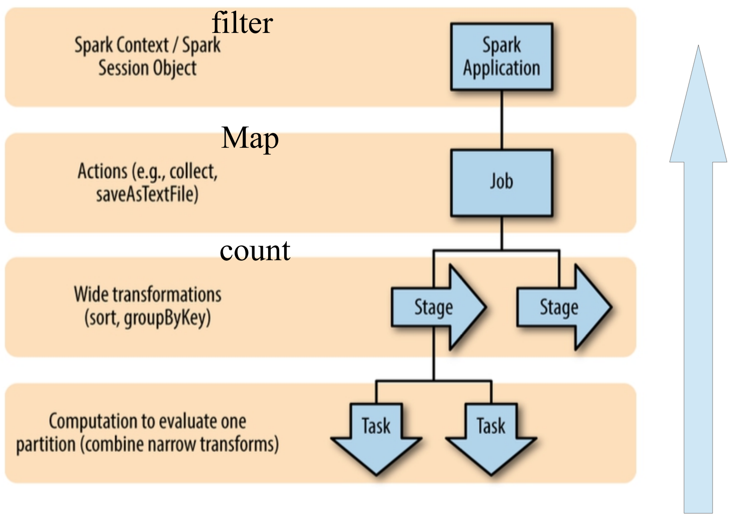
- Each input partition will contribute to only one output partition. Ex : C to D

Wide transformations (dependencies)

- Each input partition will contribute to many output partition. Ex : A to B

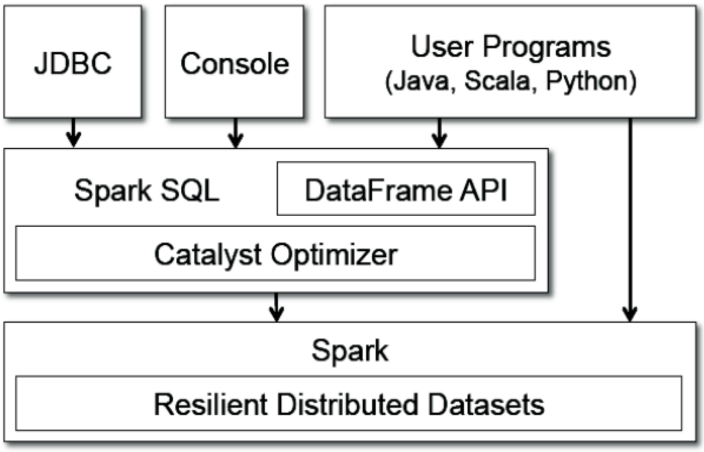
In case of faul tolerance : we Recompute only the lost partitions of an RDD.





Lecture 7 :Spark sql

Difference spark/saprk sql : spark sql use a structured data, is also using spark



a structured data know about the schema of the data it's dealing with contrary to rdds.

DataFrame: a strutured collections,equivalent to a table, allows us to do query on a data structured

It is seen as different column, exemple of query you can make :

col returns a reference to a column, expr performs transformations on a column

columns returns all columns on a DataFrame

row is record data, type Row, it doesn't have schema like column

To create a dataFrame : from a RDDs ex : `val tupleRDD = sc.parallelize(Array(("seif", 65, 0), ("amir", 40, 1))`

`val tupleDF = tupleRDD.toDF("name", "age", "id")` or

```
case class Person(name: String, age: Int, id: Int)
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleDF.toDF
```

from data source ex : `val peopleJson = spark.read.format("json").load("people.json")`

```
val peopleCsv = spark.read.format("csv")
    .option("sep", ";")
    .option("inferSchema", "true")
    .option("header", "true")
    .load("people.csv")
```

DataFrame transformations :

select and selectExpr (create a new table)

Ex :`people.select("name", "age", "id").show(2)`
`people.selectExpr("avg(age)", "count(distinct(name))", "sum(id)").show()`

filter and where both filter rows. distinct can be used to extract unique rows. withColumn adds a new column to a DataFrame. withColumnRenamed renames a column. drop removes a column.

You can use udf to define new column-based functions.

Ex :import org.apache.spark.sql.functions.{col, udf}

```
val df = spark.createDataFrame(Seq((0, "hello"), (1, "world"))).toDF("id", "text")
```

```
val upper: String => String = _.toUpperCase
val upperUDF = spark.udf.register("upper", upper)
```

```
df.withColumn("upper", upperUDF(col("text"))).show
```

Any change we apply create a new dataframe (it is immutable)

DataFrame Actions

? Like RDDs, DataFrames also have their own set of actions. ? collect: returns an array that contains all of rows in this DataFrame. ? count: returns the number of rows in this DataFrame. ? first and head: returns the first row of the DataFrame. ? show: displays the top 20 rows of the DataFrame in a tabular form. ? take: returns the first n rows of the DataFrame.

In an aggregation you specify :A key or grouping or An aggregation function , there is 3 ways of doing it : Summarizing a Complete DataFrame (with count, countDistinct, first and last return the first and last value of a DataFrame,min, max, sum,avg)

Group by : Grouping with expressions with agg ex : `people.groupBy("name").agg(count("age").alias("ageagg")).show()`
Grouping with Maps ex : `people.groupBy("name").agg("age" -> "count", "age" -> "avg", "id" -> "max").show()`

Windowing : The window determines which rows will be passed in to this function ex :

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions.col
val people = spark.read.format("json").load("people.json")
val windowSpec = Window.rowsBetween(-1, 1)
val avgAge = avg(col("age")).over(windowSpec)
people.select(col("name"), col("age"), avgAge.alias("avg_age")).show
```

```
+---+---+-------+
|age| id|   name|
+---+---+-------+
| 15| 12|Michael|
| 30| 15|   Andy|
| 19| 20|   Andy|
+---+---+-------+
Option 1
+-------+---+--------+
|   name|age| avg_age|
+-------+---+--------+
|Michael| 15|    22.5|
|   Andy| 30|   21.33|
|   Andy| 19|    24.5|
+-------+---+--------+
```

Joins Example - Inner : classique join , -outer ex : `val joinExpression = person.col("group_id") === group.col("id")`

```
var joinType = "outer"
person.join(group, joinExpression, joinType).show()
```

```
+----+-------+--------+---+----------+
|  id|   name|group_id| id|department|
+----+-------+--------+---+----------+
|   1|   Amir|       1|  1|       KTH|
|   2|Sarunas|       1|  1|       KTH|
|null|   null|    null|  2|      SICS|
|   0|   Seif|       0|  0|  SICS/KTH|
+----+-------+--------+---+----------+
```

join inner same way to use it

Two different communication ways during joins:

- Shuffle join: big table to big table, (like reducemap) divide the table in group with the same key and then join them with an executor
- Broadcast join: big table to small table, split the the big table give to an executo each partition and replicate the small table in each executor

You can run SQL queries on views/tables via the method sql on the SparkSession object, first you create a temporary view then you can

ex :
```
people.createOrReplaceTempView("people_view")
val teenagersDF = spark.sql("SELECT name, age FROM people_view WHERE age BETWEEN 13 AND 19")
```

dataframe limitation :
```
// people columns: ("name", "age", "id")
    val people = spark.read.format("json").load("people.json")
    people.filter("id_num < 20") // runtime exception
```

Because the dataframe doesn't know the type of the row is just row

solution : dataset same thing but actually no the class of the row

ex :
```
case class Person(name: String, age: BigInt, id: BigInt)
    val people = spark.read.format("json").load("people.json").as[Person]
    people.filter(x => x.age < 40).show()
    people.map(x => (x.name, x.age + 5, x.id)).show()
```

Structured APIs In Spark

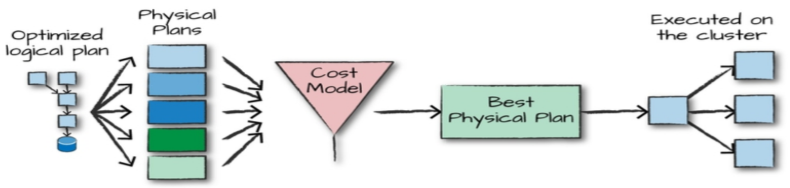
	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Structured Data Execution Steps : 0. Write DataFrame/Dataset/SQL Code.

1. If valid code, Spark converts this to a logical plan.

Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer. If the analyzer can resolve it, the result is passed through the Catalyst optimizer. It converts the user's set of expressions into the most optimized version.

2. Spark transforms this logical plan to a Physical Plan • Checking for optimizations along the way.



3. Spark then executes this physical plan (RDD manipulations) on the cluster.

Upon selecting a physical plan, Spark runs all of this code over RDDs. Spark performs further optimizations at runtime. ? Finally the result is returned to the user.

Lecture 8 : stream processing

Stream processing is the act of continuously incorporating new data to compute a result, no predetermined beginning or end.

Database Management Systems (DBMS): data-at-rest analytics

- Store and index data before processing it.
- Process data only when explicitly asked by the users.

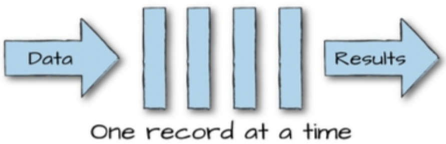
Stream Processing Systems (SPS): data-in-motion analytics : Processing information as it flows, without storing them persistently.

Data stream is unbound data, which is broken into a sequence of individual tuples. A data tuple is the atomic data item in a data stream. Can be structured, semi-structured, and unstructured.

Streaming Processing Patterns :

Micro-batch systems :Slicing up the unbounded data into a sets of bounded data, then process each batch.

Continuous processing-based systems : Each node in the system continually listens to messages from other nodes and outputs new updates to its child nodes.



Window: a buffer associated with an input port to retain previously received tuples. Different windowing management policies. :

- Count-based policy: the maximum number of tuples a window buffer can hold
- Time-based policy: based on processing or event time period

Tumbling window: supports batch operations: When the buffer fills up, all the tuples are evicted.

Sliding window: supports incremental operations: When the buffer fills up, older tuples are evicted.

Event time: the time at which events actually occurred. • Timestamps inserted into each record at the source.

Prcosseing time: the time when the record is received at the streaming application.

Ideally, event time and processing time should be equal. Skew (biais, difference) between event time and processing time.

Triggering determines when in processing time the results of groupings are emitted as panes.

Windowing determines where in event time data are grouped together for processing.

Each can be time-based or count-based

During Time-based Windowing, we use Watermarking to deal with lateness. Watermarks flow as part of the data stream and carry a timestamp t. Normaly, W(t) declares that event time has reached time t in that stream, there should be no more elements from the stream with a timestamp $t' \leq t$. But often it happen, it gets used to update a query.

We can use windowing and triggering at the same time.

Between source and processing (uses info), we need something to store shortly the info because if the processing part (consumer part) fail or is not connected or is too slow all the information are lost.

That's why we use a **Message Broker** who decouples the producer-consumer interaction.

It runs as a server, with producers and consumers connecting to it as clients. Producers write messages to the broker, and consumers receive them by reading them from the broker. Consumers are generally asynchronous.

In typical message brokers, once a message is consumed, it is deleted. Log-based message brokers durably store all events in a sequential log. A log is an append-only sequence of records on disk. A producer sends a message by appending it to the end of the log. A consumer receives messages by reading the log sequentially.

Exemple of a message brokers : Kafla a distributed(several brokers), topic oriented(not all consumer read all data), partitioned (the data are split), replicated commit log service.

Topics are queues: a stream of messages of a particular type. Topics are logical collections of partitions (the physical files), Ordered, Append only, Immutable. Ordering is **only** guaranteed within a partition for a topic. Messages sent by a producer to a particular topic partition will be appended in the order they are sent and a consumer instance sees messages in the order they are stored in the log.

Replicated : One broker is the leader of a partition: all writes and reads must go to the leader.

Kafka uses Zookeeper (Coordination) for the following tasks: Detecting the addition and the removal of brokers and consumers. And keeping track of the consumed offset of each partition.

Consumers are responsible for keeping track of offsets. Messages in queues expire based on pre-configured time periods (e.g., once a day).

Kafka only guarantees at-least-once delivery.

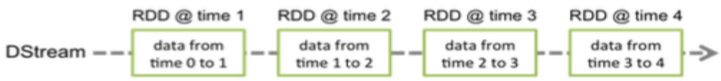
Lecture 9 : Scalable Stream Processing - Spark Streaming and Beam

Spark Streaming (batch processing)

Run a streaming computation as a series of very small, deterministic batch jobs.

Discretized Stream Processing (DStream) is a list of RDD (one RDD = the little green thing)

Zoom




```
val conf = new SparkConf().setAppName(appName).setMaster(master)
        val ssc = new StreamingContext(conf, Seconds(1))
```

Seconds(1), represents the time interval at which streaming data will be divided into batches.

For Basic Sources

Socket connection • Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

File stream • Reads data from files.

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
streamingContext.textFileStream(dataDirectory)
```

for Advanced Sources :Connectors with external sources ex : Twitter, Kafka, Flume, Kinesis, ...

```
        TwitterUtils.createStream(ssc, None)
        KafkaUtils.createStream(ssc, [ZK quorum], [consumer group id], [number of partitions])
```

Example - Word Count

```
// Create a local StreamingContext with two working threads and batch interval of 1 second.
        val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
        val ssc = new StreamingContext(conf, Seconds(1))
        //Create a DStream that represents streaming data from a TCP source
        val lines = ssc.socketTextStream("localhost", 9999)
        val words = lines.flatMap(_.split(" "))
        val pairs = words.map(word => (word, 1))
        val wordCounts = pairs.reduceByKey(_ + _)
        wordCounts.print()
        ssc.start()
        ssc.awaitTermination()
```

Spark provides a set of transformations that apply to a over a sliding window of data. A window is defined by two parameters: window length and slide interval.

Tumbling window => window length =slide interval window(windowLength, slideInterval) • Returns a new DStream which is computed based on windowed batches.

reduceByWindow(func, windowLength, slideInterval) • Returns a new single-element DStream, created by aggregating elements in the stream over a sliding interval using func.

reduceByKeyAndWindow(func, windowLength, slideInterval) 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 function func over batches in a sliding window.

Ex :
```
val pairs = words.map(word => (word, 1))
val windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _, Seconds(30), Seconds(10))
windowedWordCounts.print()
```

To Accumulate and aggregate the results from the start of the streaming job.

Ex :
```
val ssc = new StreamingContext(conf, Seconds(1))
ssc.checkpoint(".")          //provide a checkpointing directory for stateful streams.
val lines = ssc.socketTextStream(IP, Port)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val stateWordCount = pairs.mapWithState(StateSpec.function(updateFunc))
val updateFunc = (key: String, value: Option[Int], state: State[Int]) => {
    val newCount = value.getOrElse(0)     //get the value or give 0
    val oldCount = state.getOption.getOrElse(0)
    val sum = newCount + oldCount
    state.update(sum)
    (key, sum) }
```

Structured Streaming (a more accurate way of doing it)

Treating a live data stream as a table that is being continuously appended (at the end). Defines a query on the input table, as a static table, Spark automatically converts this batch-like query to a streaming execution plan. Specify triggers to control when to update the results. Three output modes:
1. Append: only the new rows appended to the result table since the last trigger will be written to the external storage.

2. Complete: the entire updated result table will be written to external storage.

3. Update: only the rows that were updated in the result table since the last trigger will be changed in the external storage.

Ex :
```
val spark = SparkSession...
val lines = spark.readStream.format("socket").option("host", "localhost").option("port", 9999).load()
import org.apache.spark.sql.functions._
val words = lines.select(split(col("value"), "\\s").as("word"))
val counts = words.groupBy("word").count()
val writer = counts.writeStream.format("console").outputMode("complete")
\\ word count details
import org.apache.spark.sql.streaming._
val checkpointDir = "... »
val writer2 = writer.trigger(Trigger.ProcessingTime("1 second")).option("checkpointLocation", checkpointDir)
val streamingQuery = writer2.start()
```

you can also read or write data from a file or from kafla : (see slide 42 , 9diaporama)

Most of operations on DataFrame/Dataset are supported for streaming.

```
case class Call(action: String, time: Timestamp, id: Int)
    val df: DataFrame = spark.readStream.json("s3://logs")
    val ds: Dataset[Call] = df.as[Call]
```

ex : Selection and projection , Aggregation , SQL commands

```
df.select("action").where("id > 10") // using untyped APIs
    ds.filter(_.id > 10).map(_.action) // using typed APIs
```

```
df.groupBy("action") // using untyped API ds.groupByKey(_.action) // using typed API
```

```
df.createOrReplaceTempView("dfView") spark.sql("select count(*) from dfView") // returns another streaming DF
```

Window operation (on the event time : time at the source)

you Aggregations over a sliding event-time window or window sliding (an element can be twice)

```
// The sensorReadings DataFrame has the generation timestamp as a column named eventTime
    sensorReadings.groupBy("sensorId", window("eventTime", "5 minute")).count()
```

```
import org.apache.spark.sql.functions.*
    sensorReadings.groupBy("sensorId", window("eventTime", "10 minute", "5 minute")).count()
```

The trailing gap (watermark delay) defines how long the engine will wait for late data to arrive.

```
  sensorReadings.withWatermark("eventTime", "10 minutes")
    .groupBy("sensorId", window("eventTime", "10 minutes", "5 minute"))
    .mean("value")
```

you can also do some stateful operations like this : (see example slide 53)

```
def arbitraryStateUpdateFunction(
    key: K,
    newDataForKey: Iterator[V],
    previousStateForKey: GroupState[S] ): U
```

```
val inputDataset: Dataset[V] = ...// input streaming Dataset
```

```
inputDataset.groupByKey(keyFunction) // keyFunction() generates key from input
    .mapGroupsWithState(arbitraryStateUpdateFunction)
```

Input data (V): case class UserAction(userId: String, action: String)
Keys (K): String (that is the userId)
State (S): case class UserStatus(userId: String, active: Boolean)
Output (U): UserStatus, as we want to output the latest user status.

Lecture 10 : graph data

Difficult to extract parallelism based on partitioning of the data. Difficult to express parallelism based on partitioning of computation. In a graph, we want to limit the connection of data from different machine 2 types of graph partionning : minimize the of data are separate (edge-cut) (good for opposite) , replicate data to keep the connections et minimize this replicat (vetex-cut) (good if few nodes have lots connections, lots nodes haves few connections)

Pregel : Large-scale graph-parallel processing platform developed at Google : Applications run in sequence of iterations, called supersteps. A vertex (node) in superstep S can:reads messages sent to it in superstep S-1, sends messages to other vertices: receiving at superstep S+1 (Vertices communicate directly with one another by sending messages) , modifies its state.

Superstep 0: all vertices are in the active state. A vertex deactivates itself by voting to halt: no further work to do. A halted vertex can be active if it receives a message. The whole algorithm terminates when: All vertices are simultaneously inactive. There are no messages in transit.

One master, Coordinates workers, Assigns one or more partitions to each worker, Instructs each worker to perform a superstep.

Each worker, Executes the local computation method on its vertices, Maintains the state of its partitions. Manages messages to and from other workers.

To prevent from fault tolerance : At start of each superstep, master tells workers to save their state: • Vertex values, edge values, incoming messages, Master saves aggregator values (if any) and When master detects one or more worker failures all workers revert to last checkpoint.

Pregel Limitations : Inefficient if different regions of the graph converge at different speed and Runtime of each phase is determined by the slowest machine.

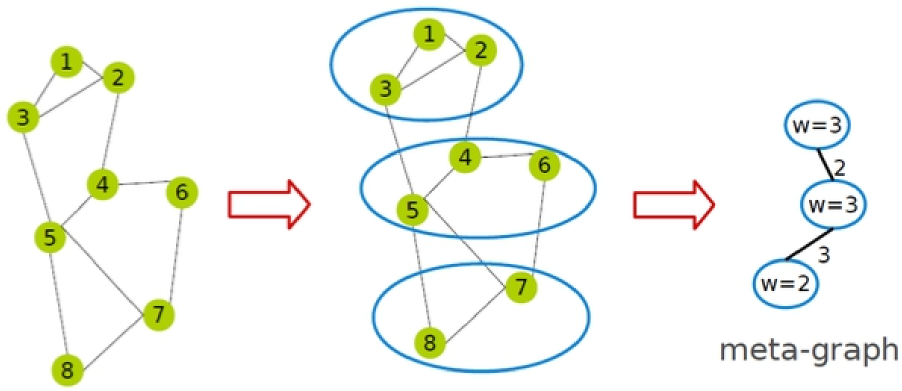
GraphLab : allows asynchronous iterative computation. A vertex can read and modify any of the data in its scope (shared memory). Some problem of consistency can appired, Overlapped scopes: race-condition in simultaneous execution of two update functions. To avoid that :

Full consistency: during the execution f(v), no other function reads or modifies data within the v scope (me and my neighbor)

Edge consistency: during the execution f(v), no other function reads or modifies any of the data on v or any of the edges adjacent to v. (no one can read or write my value, but we can read the value of my neighbor)

Vertex consistency: during the execution f(v), no other function will be applied to v. (no one can read or write on my value)

higher the level of consistency is lower the level of parallelism egdge partitioning



Fault Tolerance

Synchronous :The systems periodically signals all computation activity to halt. Then synchronizes all caches, and saves to disk all data which has been modified since the last snapshot.

Asynchronous : The snapshot function is implemented as a function in vertices (start be him then if it hasn't been save his two neighbors). It takes priority over all other update functions.

PowerGraph

Factorizes the local vertices functions into the Gather : accumulate information from neighborhood Apply: apply the accumulated value to center vertex. Scatter: update adjacent edges and vertices, 3 phases.

Synchronous scheduling like Pregel. : Executing the gather, apply, and scatter in order,
Changes made to the vertex/edge data are committed at the end of each step.
Asynchronous scheduling like GraphLab : Changes made to the vertex/edge data during the apply and scatter functions are immediately committed to the graph.
Visible to subsequent computation on neighboring vertices.

Graph Partitioning : Vertx-cut partitioning can be done randomly but high number of replicat or Greedy vertex-cuts : A(v): set of machines that vertex v spans.

Case 1: If $A(u) \cap A(v) \neq \infty$, then the edge (u, v) should be assigned to a machine in the intersection.

Case 2: If $A(u) \cap A(v) = \infty$, then the edge (u, v) should be assigned to one of the machines from the vertex with the most unassigned edges.

Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

Case 4: If$A(u)=A(v)=\infty$,then assign the edge (u,v) to the least loaded machine.

Powergraph, pregel et graphlab has a pb : if you have to use graphs processing and table proces..

GraphX : Unifies data-parallel and graph-parallel systems

Spark (graphX) represent graph structured data as a property graph

```
val users: RDD[(VertexId, (String, String))] = sc.parallelize(Array((3L, ("rxin", "student")),
        (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
        val relationships: RDD[Edge[String]] = sc.parallelize(Array(Edge(3L, 7L, "collab"),
        Edge(5L, 3L, "advisor"), Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"), Edge(5L, 1L, "-")))
        val defaultUser = ("John Doe", "Missing")
        val graph: Graph[(String, String), String] = Graph(users, relationships, defaultUser)
```

you can use information of the graphe, use function (filtere, transformation, reverse returns a new graph with all the edge directions reversed, subgraph takes vertex/edge predicates and returns the graph containing only the vertices/edges that satisfy the given predicate...), join see ex : , we can also do some sort of map function with aggregateMessages (regrouper des message)

On graphX, pregel takes two argument lists: graph.pregel(list1)(list2). The first list contains configuration parameters : The initial message, the maximum number of iterations, and the edge direction in which to send messages. The second The second list contains the user defined functions :Gather : accumulate information from neighborhood Apply: apply the accumulated value to center vertex. Scatter: update adjacent edges and vertices

graphX uses a vertex-cut and Routing table: a logical map who says in which server is replicate a vertex.

Lecture 11 : ressource management (it is a OS for all computers with this you can have acess to all ressource of all computer)

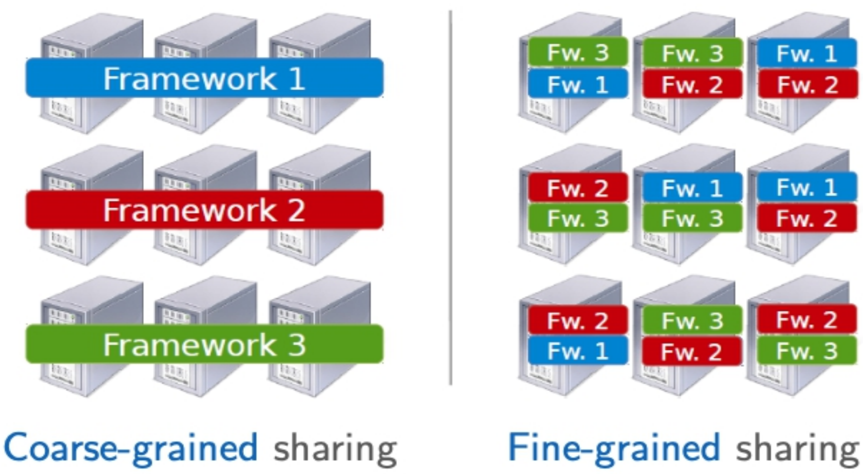
No single framework optimal for all applications, thus running each framework on its dedicated cluster expensive, hard to share data between framework. Running multiple frameworks on a single cluster, maximize utilization and share data between frameworks. (frameworks are all the tools see precedently (mapreduce,Spark ..)

Three resource management systems: Mesos, YARN, Borg

Mesos : a common resource sharing layer, over which diverse frameworks can run.

A framework (e.g., Hadoop, Spark) manages and runs one or more jobs, consists of one or more tasks, (e.g., map, reduce) consists of one or more processes running on same machine.

Mesos uses Fine-graned sharing, an allocation at the level of tasks within a job. It improves utilization, latency, and data locality.



Two types of scheduler : Global(borg) monolithic scheduler and Distributed (mesos,YARN) two level scheduler

Global takes all informations : Job requirements, Job execution plan, Estimates, global policies and availabality then create the task scheduler. It is optimal , but complex and Hard to anticipate future frameworks requirements.

Distributed (mesos) : global policies and availabality (1) then offer to frameworks (2) and

Frameworks select which offers to accept and which tasks to run , inform mesos (3) who inform the exucutor(4). (mesos knows the ressources needed per task by framework (0)). Advantages : Simple: easier to scale and make resilient and Easy to port existing frameworks, support new ones. Disadvantages : Distributed scheduling decision: not optimal.

Allocation of ressource : fair sharing, for 1 ressource (ex:CPU) everybody gets 1/n, Generalized by max-min fairness (Handles if a user wants less than its fair share. E.g., user 1 wants no more than 20%) Generalized by weighted max-min fairness (Give weights to users according to importance)

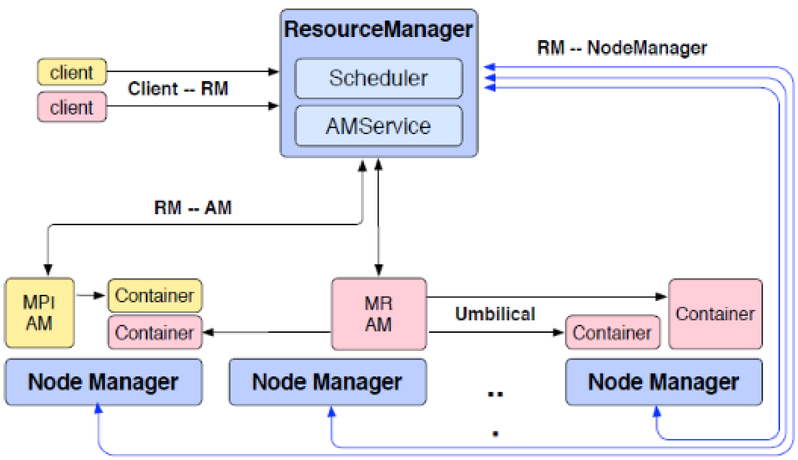
for 2 ressources (CPU, RAM) : Asset fairness: give weights to resources (e.g., 1 CPU = 1 GB) and equalize total value given to each user.
Dominant resource of a user: the resource that user has the biggest share of (Total resources ⟨8CPU, 5GB⟩ User 1 allocation: ⟨2CPU, 1GB⟩: 2/8 CPU and 1/5 RAM Dominant resource of User 1 is CPU, then apply max-min fairness to dominant shares: give every user an equal share of her dominant resource.

YARN (Global scheduler)

1 - a client submit a job to the ressource manager (RM) 2 - The RM provides an Application Id

3 - The client provides a CLC (Container Launch Context, the command necessary to create the process, environment variables, security tokens, etc)

4-The RM asks a NM (node managers (NM) tell to the RM which ressource (container) is available) to launch an application manager (AM, responsible for a job, Request resources from RM) 5- The selected NM launches an AM. 6- The AM registers with the RM. The RM shares resource capabilities with the AM. 7- The AM requests containers (logical bundle of resources (CPU/memory)), and the RM assigns containers based on policies and available resources.



Borg

To use, you ask him to do a job giving : cell (cluster) to run in, program to run, parameters, …
Cell(cluster) : a set of machines managed by Borg as one unit. Allocs can be used to set resources aside for future tasks, to retain resources between stopping a task and starting it again, and to

gather tasks from different jobs onto the same machine

Architecture : BorgMaster : The central brain of the system, Holds the cluster state, Replicated for reliability (using paxos), Scheduling: where to place tasks
Borglet : Manage and monitor tasks and ressource, Borgmaster polls Borglet every few seconds

Main difference with Yarn is that it find ressources to a job but also assign directly a task to one machine, to do so it also take care of preferencia and built-in creteria. It is a Monolithic schedulers: use a single, centralized scheduling algorithm for all jobs.

Contrary to Yarn and mesos who are : Two-level schedulers: separate concerns of resource allocation and task placement.

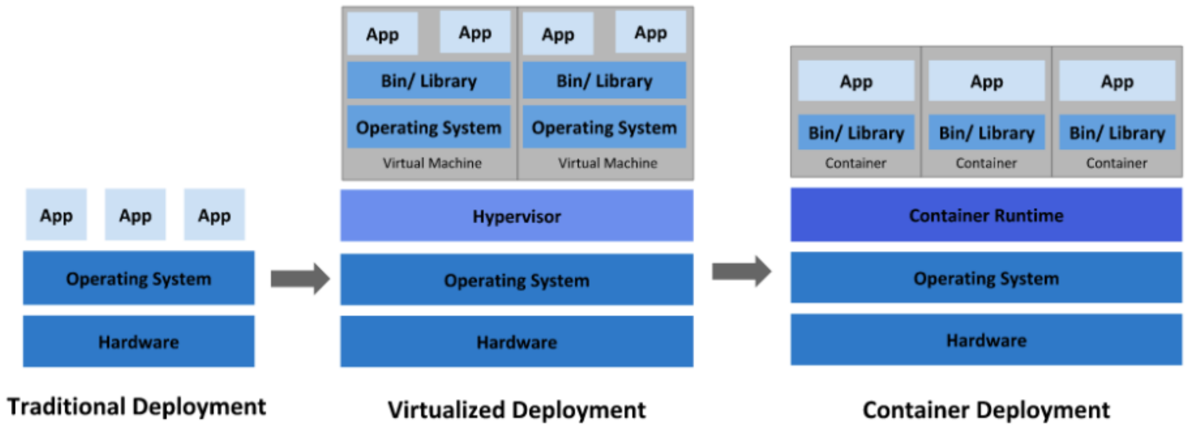
Application deployement :

Traditional deployement (Running applications on physical servers) :No resource boundaries for applications in a physical server. Resource allocation issues, e.g.,one application would take up most of the resources, so the other applications would underperform.

Virtual Machines (VMs): a full machine running all the components, including its own operating system (OS), on top of the virtualized hardware. Virtualization allows to run multiple VMs on a single physical server's CPU : Allows applications to be isolated between Vms,

Secure, as the information of one application cannot be, freely accessed by another application,
Utilizes the resources of a physical server better.
Better scalability as applications can be added/updated easily.

Containers are similar to Vms, but they have relaxed isolation properties to share the OS among the applications. A container packages applications as images that contain everything needed to run them: code, runtime environment, libraries, and configuration. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.



Docker is a virtualization software that creates container from a copy of a template called docker image (We can have multiple containers (copies) of the same image). Docker daemon represents the server that you contact with Docker client, the command line tool to get (the docker image). Docker stores the images in registries (public and private), Docker registries (Docker hub is a public registry of Docker images)

Container orchestration can help to manage the life cycles of containers, especially in large and dynamic environments. His tasks : Provisioning and deployment of containers. Redundancy and availability of containers. Scaling up or removing containers to spread application load evenly across host infrastructure Movement of containers from one host to an other, if there is a shortage of resources in a host, or if a host dies, Allocation of resources between containers, Load balancing of service discovery between containers, Health monitoring of containers and hosts, Configuration of an application in relation to the containers running it.

You describe the configuration of your application in a YAML or JSON file where you said how you want him to handle the task below.

Ex of Container orchestration : Kubernetes (based on Borg but improve), Marathon (runs on Mesos)

Lecture 12 : datalake

Data pb : 60% reported data quality as top challenge. 86% of analysts had to use stale data, with 41% using data that is > 2 months old. 90% regularly had unreliable data sources over the last 12 months. 75% of the time of data analyst is to analyze and understand data

Getting high-quality, timely data is hard!

Data Management in 1980 : Data warehouses ETL (Extract, Transform, Load) data directly from operational database systems. Purpose-built for SQL analytics and BI but (2010) Could not support rapidly growing unstructured and semi-structured data: time series, logs, images, documents, etc. High cost to store large datasets. No support for data science and ML.

Data lakes (2010) : Low-cost storage to hold all raw data, does EL and the transormation is done by the users themself. Data reliability suffers: Multiple storage systems with different semantics,

Data Lake is ideal for those who want in-depth analysis whereas Data Warehouse is ideal for operational users (business case)

Data Lake defines the schema after data is stored whereas Data Warehouse defines the schema before data is stored.

Lakehouse combine the two, the system implement Data Warehouse management and performance features on top of directly-accessible data in open formats.

Metadata layers for Data Lakes enable us to access to different form of the same data, versioning, at high speed Delta Lake is Metadata layer

Uses New Query Engine Designs, Great SQL performance on Data Lake storage systems and file formats.

ML frameworks already support reading Parquet, ORC, etc.

Delta Lake is an open source storage layer that brings reliability to Data Lakes. Provides ACID transactions. Provides scalable metadata handling. Provides time travel and versioning. Unifies streaming and batch data processing.

Delta Lake maintains information about which objects are part of a Delta table. A Delta Lake table is a directory (e.g., mytable) that holds data objects and a log of transaction operations (in a sub-repository DeltaLog, one file per commit can be several actions, automatically created when a Delta Lake table is created). Before apply any operation on a Delta Lake table, Spark checks the table DeltaLog to see what new transactions have posted to the table. Example of actions : Change metadata: name, schema, partitioning, etc. Add/remove file: adds/removes a file, Protocol evolution: upgrades the version of the transaction protocol, Set transaction: records an idempotent transaction id, Commit info: information around commit for auditing.

Every 10 commit files (json) it concatenate them into one parquet file it's easier to read and quicker

If several people are modifying, a repository simultaneous it tried Optimistic Concurrency Control, means if it is possible we see the result as if it has be done one after the other. In the vast majority of cases, this reconciliation happens silently and successfully. However, in the event that there is an irreconcilable problem, it throws an error.

We can use this json file to time travel or data versioning, it can also help to debug

Schema enforcement: prevents users from accidentally polluting their tables with mistakes or garbage data. To do soit has 3 rules : Rule 1: cannot contain any additional columns that are not present in the target table’s schema. Rule 2: cannot have column data types that differ from the column data types in the target table. Rule 3: Can not contain column names that differ only by case (letter).

Schema evolution: enables automatic addition of columns when desired. With .option(’mergeSchema’, ’true’), you can Adding new columns (this is the most common scenario). Changing of data types from non-nullable to nullable. Upcasts from ByteType → ShortType → IntegerType. With .option(’overwriteSchema’, ’true’), you can Dropping a column. Changing an existing column’s data type (in place). Renaming column names that differ only by case (e.g., Foo and foo).

In the end, there is some exemple on how to use delta lake with spark, you can : Loading Data into a Delta Lake Table, Loading Data Streams into a Delta Lake Table, Schema Enforcement, Schema Evolution, Transforming Existing Data - Updating Data – deleting data - Upserting Data to a Table, Auditing Data Changes with Operation History, Querying Previous Snapshots of a Table with Time Travel