TEORIA SPARK del ppt “Intro to Spark”

Unified analytics engine for large-scale data processing
Parallel data processing on computer clusters

- Processing data at scale: Terabytes
- Use common resources (local or cloud)
- From a Pandas-like framework (high level abstractions)

Common data processing pattern:
Data Flow:

- Read/ aggregate raw data from many sources
- Clean/ normalize
- Transform
- Analyze
- Store

Spark as an unified platform for writing big data application:

- Data loading
- SQL queries
- machine learning
- streaming computation
- composable APIs

Spark computing engine

- **Move computation to data, not data to computing cores**.
- **Spark handles loading data and mix of storage systems.**

Vaig a montar un cluster i dividir les dades en blocs petits. On cada ordinador fa el mateix comput però amb les seves dades. Execució local. Moure la computació a les dades. Puc tenir les dades en múltiples llocs i no em preocupo de sincronitzar ni distribuir.

Big data explosion!!!:
Why didn’t Spark exist 10 years ago? because of all the changes of having more data.

- **Speed:** Applications need to add parallelism to run faster. From 2005: no faster CPUS, but an increase in CPU cores.
- **Cost:** cost drop of storage. 1TB storage cost cuts in half every 14 months.
- **Cost.2**: cost drop of collecting data technology.

Why Spark?

- Collecting data is inexpensive
- Need of large, parallel computations
- Difficult to scale large software solutions and traditional models (SQL
- Spark proposal of cluster computing with working sets.

En conclusió: **Speed, Ease of use, Generality, Runs everywhere**

CORE CONCEPTS

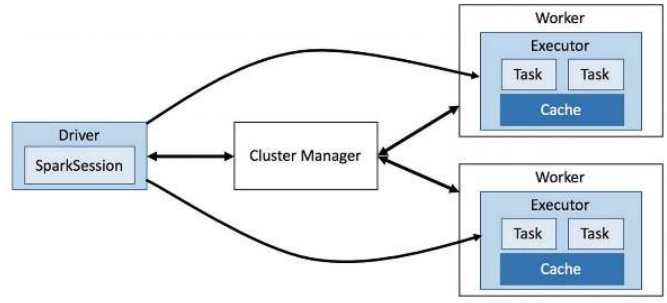
1. **Resilient Distributed Datasets: RDDs**
   - Fault-tolerant because of parallel data structures
   - Read-only, partitioned collection of records ((si només em deixa llegir i calculo una mitjana, on la guardo? doncs escriure al final. Només puc llegir i afegir al final. No puc modificar, operació molt lenta))
   - Operated through transformations and actions

2. **Spark is a distributed system to process very large volumes of data.** The world's largest Spark cluster has more than 8000 machines. **Resource management systems:** master and worker concepts.

1 DRIVER process per Spark App

N Executors per Driver — **Master/Worker architecture**

Driver/Master: El procés principal que gestiona l'execució de Spark. S'encarrega d'enviar les tasks als executors per al seu processament.
Executor/Worker: Els executors són els workers que realment executen les tasks. Cada executor rep les tasks del Driver, llegeix les particions de dades que li toquen, i les processa utilitzant els cores de CPU disponibles.



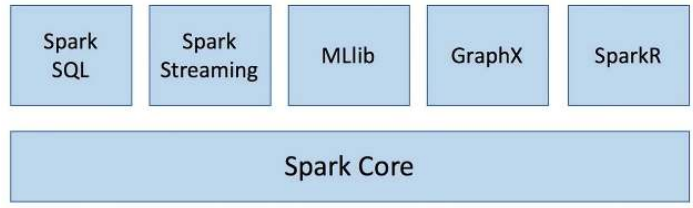
##Pregunta profe a classe, pot entrar: Quina és la relació entre CPU core i Task ?
1 CPU core per Task: En Spark una Task és la unitat bàsica de treball i s'executa en un únic core de CPU. Llavors la quantitat de CPU cores disponibles en un clúster determina la quantitat de tasks que es poden executar simultàniament. Per exemple, si tens un clúster amb 8 cores de CPU, pots executar fins a 8 tasks en paral·lel.

##Pregunta profe a classe, pot entrar: Quina és la relació entre Task i la partició de dades
1 Task per partició de Dades: Cada Task s'encarrega de processar una partició de dades específica. Quan es distribueixen les dades, Spark les divideix en particions, i cada partició és assignada a una task per ser processada. Això implica que el nombre total de tasks en una aplicació Spark serà igual al nombre de particions de dades.

##Pregunta profe a classe, pot entrar: Com es distribueixen les dades?
Les dades es divideixen en particions, les quals són assignades a task. Cada task té una partició especifica. Aquestes tasks són dividides en CPU cores, quantes més CPU cores i

particions hi hagi, més tasks es poden executar simultàneament, el que generalment condueix a un processament més ràpid de les dades.

SPARK ARCHITECTURE:



- Unified stack built on top of Spark Core
- Separate libraries targeting specific data processing workloads
- Data flows through the APIs with no need of intermediate storage

Spark Core:

- Fault tolerance
- In-memory computation: cache() and persist()
- Scheduling and monitoring
- Interacting with storage systems

Spark SQL:
Introduces the df high-level API; blurs the line between RDDs and relational tables; read/write JSON, CSV; catalyst optimizer

MLib:
Based on the df API; +50 common MLalgorithms; “Featurization”, Hyperparameter tuning, model persistence

Spark Streaming, GraphX, SparkR:
Fault tolerant streaming apps; graph-parallel computation; large-scale data analysis using R

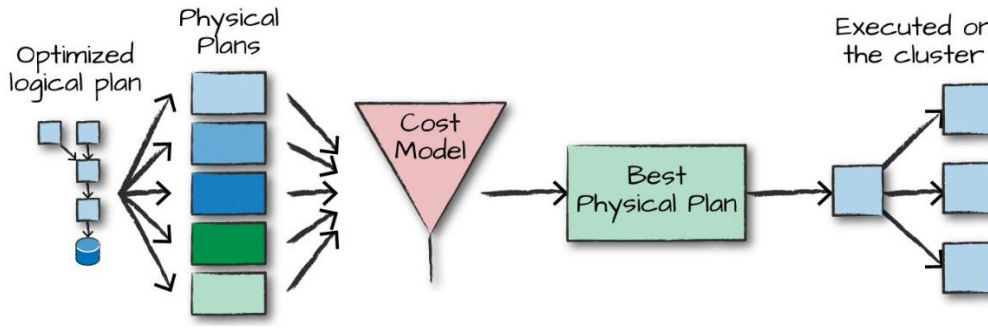
Spark Dataframes:

Distributed table-like collections of well-defined rows and columns. Each column has the same type data and the same number of rows.

Action on DataFrames: Spark plans on how to manipulate rows and columns to compute the result

Structured API Execution:

1. Write DataFrame code
2. If the code is valid it’s converted to a **Logical plan**
3. Transform logical plan to a **physical plan: Catalyst optimizer**
4. Spark executes physical plan on the cluster.



Select: manipulate columns: c.select(“customer”,”product”).show
SelectExpr: manipulate columns and manipulate names + build complex expressions: c.selectExpr(“customers as customer_id”).show ;; c.selectExpr(“*”,”(quantity = price) as equal_price”) ;; c.selectExpr(“avg(price)”, “count(customer)”)

```
+----------+--------+--------+-------+--------+-----+-----------+
|      date|    time|customer|product|quantity|price|equal_price|
+----------+--------+--------+-------+--------+-----+-----------+
|01/10/2018| 2:20 PM|     100|      1|       6|   86|      false|
|04/08/2018|11:38 AM|     200|      2|       8|   79|      false|
```

```
+----------+---------------+
|avg(price)|count(customer)|
+----------+---------------+
|    73.875|              9|
+----------+---------------+
```

Where: filtering rows. Used to filter rows with an expression of T/F:
c.where(expr(“product=8”)).show ;;
c.where(expr(“quantity<9”)).where(expr(“customer!=100”)).show
Orderby: sorting rows: c.orderBy(“price”).show ;;
c.orderBy(desc(”customer”),asc(“price”)).show

Aggreations:

- **Aggregate:** collect data together
- **Summarize** numerical data by custom grouping
- **Key** of grouping**:** column to focus on
- **Aggregation function:** how to transform column values
    - count
    - countDistinct
    - approx_count_distinct
    - last
    - min/max
    - sum, sumDistinct
    - avg
    - var_pop/stdev_pop

c.groupBy(“customer”).count.show
c.groupBy(“customer”).agg(sum(“quantity”)).show() = for those customers with the same value, add all the products bought by customer.