

# BIG DATA

ANNO: 18/19  
RROK GJINAJ – LORIS CANGINI

//Mancano alcuni dettagli.

<b>Definizioni</b>	5
Big Data	5
Le 4 V dei Big Data	5
<b>Ciclo di vita dei Big Data</b>	6
Acquisizione	6
Estrazione	6
Integrazione	6
Analisi	6
Interpretazione	7
Decisione	7
<b>Sfide</b>	7
<b>Infrastruttura e architettura Big Data</b>	7
Computazione distribuita	8
Symmetric Multi Processing (SMP)	8
Massively Parallel Processing (MPP)	9
Cluster computing	9
Grid computing	9
High Performance computing	9
Architettura software	10
NIST	10
Lambda	10
Kappa	10
<b>Hadoop</b>	11
Architettura Hadoop	11
HDFS	11
Blocco	12
Namenode (NN)	12
Datanode (DN)	13
Data locality	13
I/O	13
Formati HDFS	13
File Standard	13
Formati specifici	13
YARN (Yet Another Resource Negotiator)	15
Walkthrough	16
MapReduce	16
Combinatori	17
Partizionatori	17
Quanti Task di Reduce usare?	17
Esecuzione MapReduce	18
Algoritmi MapReduce	18
MapReduce a 2 stadi	18

Altri Moduli Hadoop	19
Ambari	19
ZooKeeper	19
Oozie	19
Pig	19
Sqoop (SQL-to-Hadoop)	20
Hive	19
HBase	20
Phoenix	20
Spark	20
Storm	20
Giraph	20
Mahout	20
Kafka	20
Impala	20
Distribuzioni principali	20
<b>Spark</b>	21
RDD (Resilient Distributed Dataset)	22
Creazione RDD	22
Operazioni	22
Trasformazioni	22
Azioni	22
RDD Lineage	23
DAG (Directed Acyclic Graph)	23
Architettura Spark	24
Ottimizzazione di Spark	25
Partizionamento	25
Shuffling	25
Serializzazione	27
Configurazione del Cluster	27
CPU	27
Memoria	27
Variabili Condivise	28
<b>Spark SQL</b>	29
Features	29
Utilizzo	30
Ottimizzatore Catalyst	30
Alberi e regole	30
Analisi	30
Ottimizzazione logica	30
Pianificazione fisica	31
RDD Lineage	31
<b>Data Streaming</b>	32

Architettura	32
Collezione	33
Message Queuing	33
Analisi	33
Modelli di dati	34
Windowing	34
Algoritmi	35
Random Sampling	35
Contare gli elementi distinti	35
Membership	36
Frequenza	36
Storage dei dati	36
Accesso ai dati	36
WebHook	36
HTTP Long Polling	36
Server-sent events	36
WebSockets	37
<b>Database NoSQL (Not only SQL)</b>	37
Modelli di dati	38
Aggregati	38
Chiave-Valore	38
Documentale	38
Wide Column	38
Grafo	39
Consistenza	39
Sharding	40
Replication	41
Persistenza poliglotta	42
<b>Big Data Mining</b>	43
Partitioning, Communicating, Agglomerating, Mapping (PCAM)	43
Partizionamento	43
Comunicazione	43
Agglomerazione	43
Mappatura	43
Conclusioni	43
<b>Cloud Computing</b>	45
<b>Amazon Web Services (AWS)</b>	45
<b>Data Science</b>	49
<b>Data Scientist</b>	49
CRISP-DM	49
Comprensione del business	49
Comprensione dei dati	50

Preparazione dei dati	50
Modellazione	50
Valutazione	50
Distribuzione	50
<b>Machine Learning in Spark</b>	50
Valutazione	50
Matrice di confusione	51
Metriche per classificazione	51
Errore di regressione	51
<hr/>	
<b>Pratica</b>	51
<i>Web GUI</i>	51
<i>Comandi HDFS</i>	52
<i>MapReduce</i>	52
<i>Spark UI</i>	52
<i>Resource Manager</i>	52
<i>History Server</i>	52
<i>Spark Streaming</i>	52

# Definizioni

---

## Big Data

Quei dati che oltrepassano le capacità dell'hardware/software comune.

Dataset oltre le capacità di gestione dei programmi comuni.

### Le 4 V dei Big Data

- **Velocità:** prodotti ed analizzati ad alta velocità
- **Volume:** tanti dati
- **Varietà:** i dati arrivano da diverse fonti
- **Veracità:** incertezza sui dati (inconsistenza, incompletezza, ambiguità, ...)

Si possono considerare altre V:

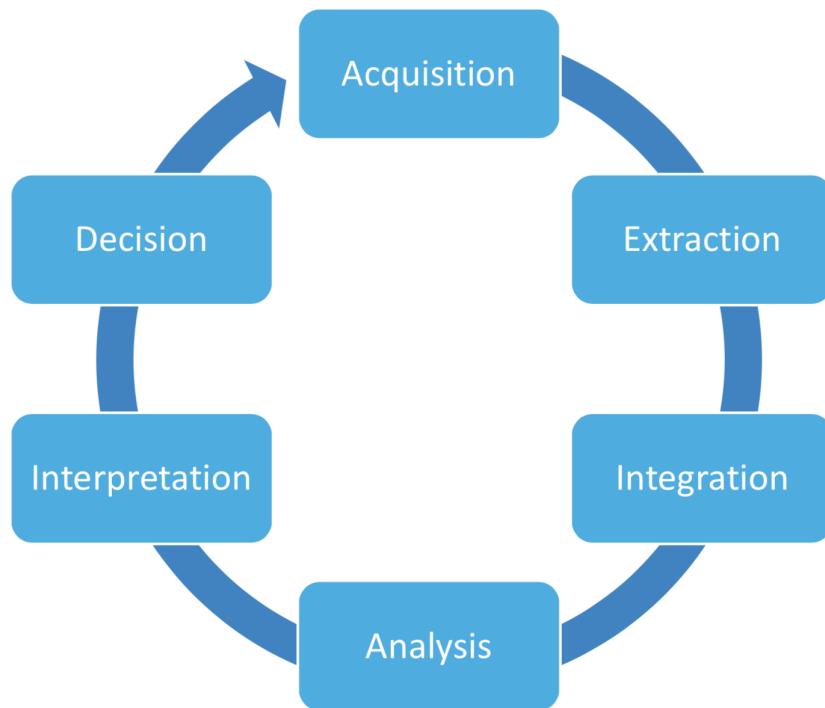
- Valore: possibilità di ritrovare il dato all'occorrenza
- Visualizzazione: si può dare senso al dato
- Viscosità
- Viralità

Data la varietà si possono applicare modelli + sofisticati.

- ✓ Algoritmi che fungono meglio
- ✓ Tolleranza errori
- ✓ Sfruttamento della *long tail* ⇒ la conoscenza è estratta principalmente dai dati insignificanti
- X Dati eterogenei ⇒ + errori
- X dati + veloci della potenza dei chip

## Ciclo di vita dei Big Data

---



Lo stesso dato attraversa tutti gli stadi.

### Acquisizione

- Selezione: capire quali dati sono utili
- Filtraggio e compressione
- Generazione di metadati

### Estrazione

- Trasformazione e normalizzazione
- Pulizia

### Integrazione

- Standardizzazione, risoluzione conflitti

### Analisi

- Esplorazione: capire i dati in tempo reale
- Data mining / Machine learning
- Visualizzazione

## Interpretazione

- Conoscenza del dominio
- Conoscenza della provenienza
- Identificazione di pattern

## Decisione

- Miglioramento continuo

## **Sfide**

---

- Performance
  - Distribuzione delle risorse e dei servizi
- Eterogeneità
- Efficacia
- Comprensione dei risultati
- Privacy
- Costi

## **Infrastruttura e architettura Big Data**

---

Scale-up: migliorare le risorse disponibili (processori, RAM, ...)

- ✓ Meno consumo
- ✓ Meno costo di raffreddamento
- ✓ Meno complesso da implementare
- ✓ Meno costi di licenze
- ✓ Meno equipaggiamento di rete
- X Elevati costi per i componenti
- X Maggior rischio di fallimenti hardware
- X lock-in sui vendor
- X Limitato nel tempo

Scale-out: aggiungere risorse a quelle esistenti

- ✓ Minor costo
- ✓ Fault-tolerance e monitoraggio più semplici
- ✓ Facile da migliorare
- X Maggior costo per licenze
- X Maggior consumo
- X Maggior equipaggiamento di rete

## Computazione distribuita

Idea ⇒ parallelizzare il lavoro da fare e farlo eseguire a diversi worker.

Problemi:

- come assegnare il lavoro ai worker?
- Che fare se c'è troppo lavoro?
- Che fare se i worker devono condividere dati?
- Come aggregare risultati?
- Come sapere se un worker ha finito?
- Che fare se dei worker muoiono?

Rischi:

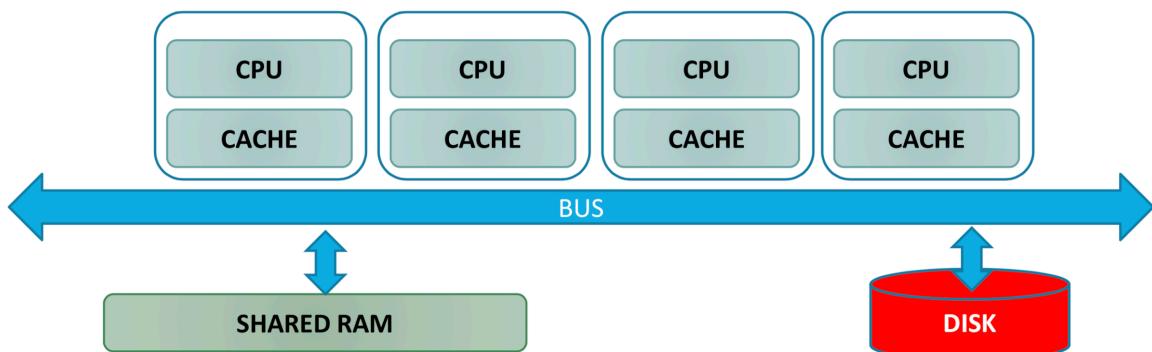
- Deadlock, Starvation, ...
  - Causati da comunicazione e accesso alle risorse condivise
- Serve sincronizzazione

Alcune soluzioni sono:

- Si nascondono dettagli agli sviluppatori
  - Si evitano race condition, ...
- Si separa il cosa dal come
  - Lo sviluppatore specifica la computazione da eseguire
  - Il framework decide come eseguirla

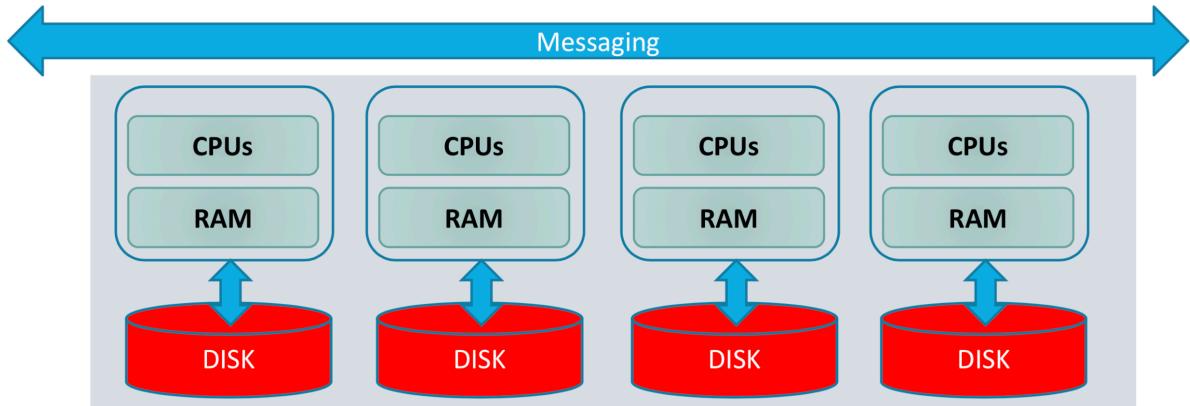
## Symmetric Multi Processing (SMP)

Molti processori con la stessa RAM, lo stesso bus di I/O e lo stesso disco.



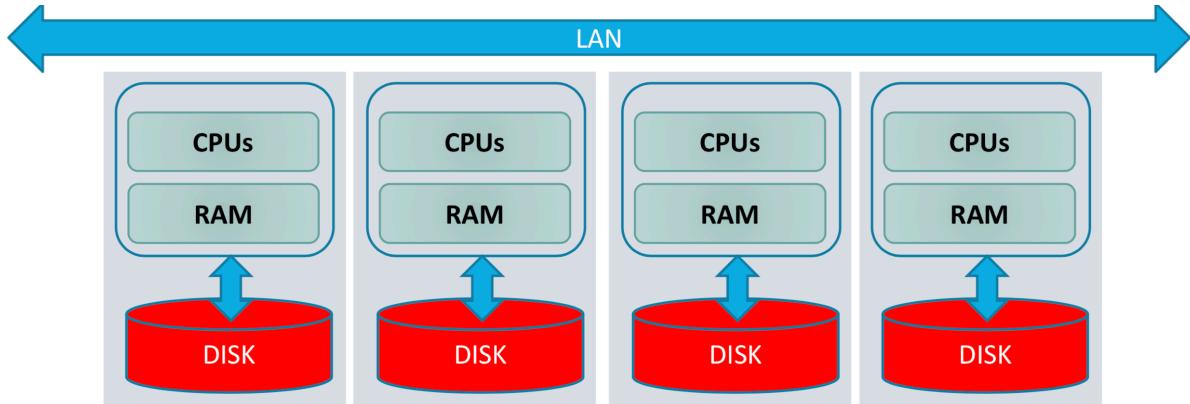
## Massively Parallel Processing (MPP)

Molti processori, ognuno con il proprio disco e RAM, comunicanti tramite messaggi (shared nothing architecture).



## Cluster computing

Gruppo di computer connessi tramite LAN.



## Grid computing

Collezione di computer eterogenei da diverse locazioni.

## High Performance computing

Sistemi massivamente paralleli creati per risolvere compiti cpu-intensivi.

## Architettura software

### NIST

5 ruoli:

- Orchestratore: definizione e integrazione delle attività
- Provider dei dati: fornitore di dati
- Application Provider: esecutore di operazioni
- Framework Provider: mantentore della struttura
- Consumatore: utente finale

### Lambda

Tutti i dati passano per 2 vie:

- **Hot**: dati in real-time, ma poco accurati
- **Cold**: dati non in tempo reale, ma più accurati

Svantaggi:

- logica duplicata
- complessità nel gestire le architetture delle due vie

### Kappa

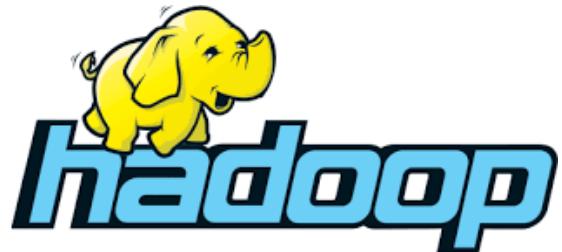
Tutti i dati fluiscono per una singola via, che funziona tramite stream.

## Hadoop

---

Framework open-source di Apache.

Hadoop è un framework che permette l'**elaborazione distribuita di grandi dataset** attraverso **cluster di computer** usando modelli di programmazione semplici.  
È progettato per **trovare e gestire fallimenti a livello applicativo**.



**Goal** ⇒ immagazzinamento e elaborazione di grandissimi dataset.

Centinaia di soluzioni, raccoglibili in 3 scenari:

- Analisi ⇒ Batch
- Streaming ⇒ Near real-time
- Interattivo ⇒ Real-time

Le principali tecniche usate per fare analisi sono:

- ETL (Extract, Transform, Load)
- Data fusion e data integration
- Cloud computing
- Data mining
- Machine learning
- ...

**Infrastruttura** ⇒ cluster di commodity hardware

**Moduli** ⇒ HDFS, MapReduce, YARN

## Architettura Hadoop

### HDFS

È il **DFS** (Distributed File System) di Hadoop.

È progettato per memorizzare file di grandi dimensioni (GB/TB), con accesso via stream.

Le applicazioni che funzionano su HDFS necessitano di scrivere i dati una volta e leggerli molte.

I fallimenti hardware sono la norma e non un'eccezione ⇒ I dati sono ridondanti nel cluster.

- Normalmente vengono replicati 3 volte
  - la prima copia nel nodo dove il client ha dato il comando di scrittura.
  - la seconda in un nodo in un rack diverso dalla prima copia.
  - la terza in un nodo diverso dalla seconda, ma nel suo stesso rack.

HDFS non è la soluzione a tutto.

- Le applicazioni che devono accedere ai dati molto rapidamente non funzionano bene su HDFS in quanto non garantisce bassa latenza.
- Non funziona bene quando si hanno molti file piccoli.

Blocco

Unità minima letta o scritta.

Normalmente variano da 64MB a 1GB.

Se un file è più piccolo di un blocco, non lo occupa tutto, ma solo il necessario.

L'utilizzo dei blocchi semplifica la gestione e la replicazione dei dati.

La loro grandezza è dovuta ad una questione di performance (meno seek).

Namenode (NN)

Il nodo padre, mantiene il filesystem e le locazioni dei blocchi per ogni file (sparsi nei vari datanode).

È uno SPoF (Single Point of Failure), in quanto senza di esso HDFS non va.

- Possibile soluzione è replicare lo stato del NN in diversi filesystem.
- Altra soluzione è usare un NN secondario che possa essere usato per far ripartire il primario in caso di fallimenti.

Per garantire **High-Availability** si configurano 2 Namenode, uno attivo e l'altro in attesa.

Nel caso in cui quello attivo fallisca, l'altro subentra.

Il NN in attesa si mantiene aggiornato.

La dimensione dello spazio dei blocchi è limitata dalla memoria del NN.

Per ovviare a questo problema si possono usare NN aggiuntivi, ognuno responsabile di una porzione del file system (namespace). Questo meccanismo viene detto **Federation**.

Questo garantisce maggiore **Scalabilità, Performance, Disponibilità, Manutenibilità, Sicurezza e Flessibilità**.

### Datanode (DN)

Memorizza e ritrova blocchi.

Dice periodicamente al Namenode quali blocchi sta mantenendo (heartbeats).

I nodi sono organizzati in rack, a loro volta organizzati in data centers.

### Data locality

Hadoop sfrutta la topologia del cluster per applicare principi di località dei dati: **muovi il codice dai dati invece del contrario**.

### I/O

Per leggere un file, un cliente deve:

1. **Chiedere al NN per determinare dove i dati voluti si trovano.**
2. Il NN risponde con gli id dei blocchi del file da leggere e il loro DN.
3. **Il client contatta i DN per rimediare i dati.**

**I dati non vengono mai spostati all'interno del NN**, ma solamente tra DN e client.

**Le comunicazioni con il NN implicano solamente trasferimenti di metadati.**

Per scrivere un file, i passaggi sono simili.

1. **Il client domanda al NN i blocchi nei quali scrivere.**
2. **Il NN risponde con gli ID dei blocchi e con i DN nei quali replicare il dato.**
3. Il client scrive

### Formati HDFS

#### File Standard

(*csv, xml, json, binari, ...*)

Quando serve compattarli o scompattarli bisogna necessariamente lavorare su tutto il file.

#### Formati specifici

Compressione per blocchi, quindi può essere fatta in parallelo.

#### **Sequence File (chiave-valore)**

Usato per storage intermedio o per memorizzare files piccoli

#### **Formati di serializzazione**

- Apache Thrift (Facebook) e Protocol Buffers (Google)
  - va definito a priori lo schema dei dati da serializzare.
- Apache Avro
  - serializza schema e dati assieme.

### Formati colonnari

Salvano i dati per colonne invece che per righe

- se serve leggere lo stesso dato da molti record, in questo modo è molto + efficiente.
- Miglior compressione.

### ORC (Optimized Record Columnar) Files

Maggior compressione di Parquet, ma solo per dati non innestati.

### Apache Parquet

Ogni tupla è un *message*

Internamente può avere dati primitivi (*string*, *int*, ...) o gruppi (*group*) (contenenti altri dati)

Possibili required, optional (0-1), repeated (0-N)

```
message M {
    repeated group G {
        required string S;
        optional int I;
    }
}
```

Sfrutta un metodo intelligente per memorizzare anche dati innestati (e.g. array) in formato flat.

- **livello di ripetizione:** quale livello della struttura si sta ripetendo.
  - Utile solamente nei livelli che possono ripetersi, altrimenti sarebbe sempre 0.
  - esempio: `[[a,b,c],[d,e,f,g]], [[h],[i,j]]`

Repetition level	Value
0	a
2	b
2	c
1	d
2	e
2	f
2	g
0	h
1	i
2	j

- **livello di definizione:** il livello di innestamento.
  - I livelli required non necessitano di livelli di definizione in quanto devono obbligatoriamente esserci.

- esempio:

```
message M {
    optional group a {
        required group b {
            optional string c;
        }
    }
}
```

Value	Definition Level
a: null	0
a: { b: null }	Impossible, as b is required
a: { b: { c: null } }	1
a: { b: { c: "foo" } }	2 (actually defined)

- Possibile comprimere
- I files non sono totalmente memorizzati per colonne
  - si prendono gruppi di righe e vengono memorizzati essi per colonne.
  - normalmente la dimensione dei gruppi è uguale alla dimensione dei blocchi HDFS.

## YARN (Yet Another Resource Negotiator)

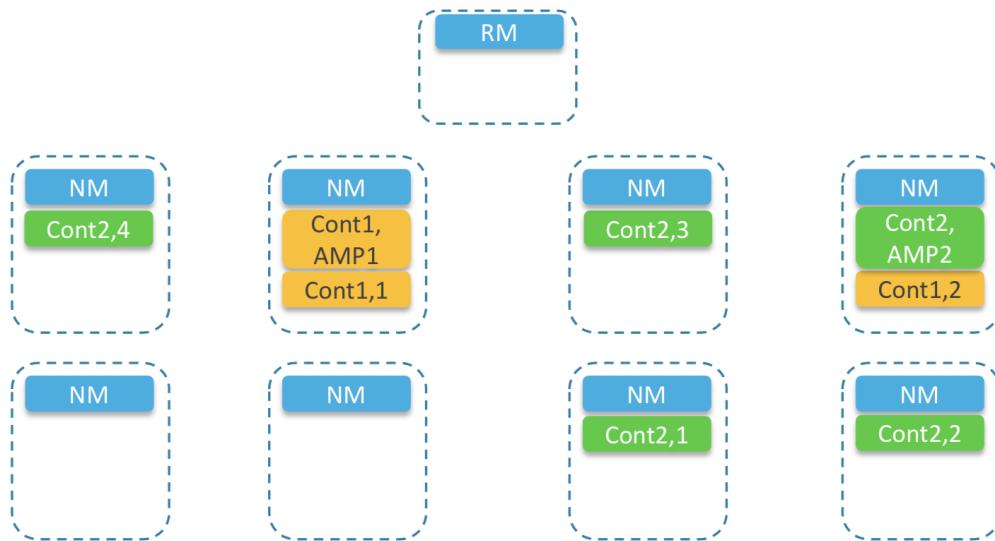
Framework per schedulazione dei job e gestione delle risorse.

**Non viene usato direttamente dal codice, ma dai framework** (MapReduce, Spark, ...).

È formato da 2 componenti principali:

- **Resource Manager (RM)**: decide globalmente come associare le risorse alle applicazioni.
  - **Scheduler**: responsabile per allocare risorse alle applicazioni.
    - 3 principali tipologie:
      - FIFO
      - Capacity: capacità fissa ad ogni job
      - Fair: bilanciamento dinamico delle risorse tra i job
  - **Application Manager**: accetta job, trovando il container nel quale eseguire l'AMP. Fornisce anche un servizio per far ripartire AMP in caso di fallimenti.
- **Node Manager (NM)**: responsabile a livello di container (entità astratta usata per eseguire applicazioni).

Su richiesta di un client, il RM trova un NM sul quale lanciare l'**Application Master Process (AMP)** in un container. L'AMP potrà poi richiedere altri container.



## Walkthrough

1. Il client sottmette l'applicazione al RM
2. Il RM lancia l'AMP in un container
3. L'AMP si registra nel RM
4. L'AMP domanda risorse, quando necessario, al RM
5. AMP lancia container nei NM
6. Il codice esegue nei container allocati nei NM fornendo informazioni all'AMP
7. Il client comunica con AMP per avere informazioni
8. Una volta conclusa l'esecuzione, AMP si cancella dal RM e si spegne, liberando il container.

## MapReduce

Sistema basato su YARN per la elaborazione parallela di grandi dataset.

Stesse operazioni della programmazione funzionale.

- **Map:** applico la stessa funzione a tutti i dati in input, trasformandoli
  - molto parallelizzabile
- **Reduce:** accumulo in un qualche modo
  - se la riduzione è parziale, allora è parallelizzabile

L'implementazione su Hadoop si chiama *Hadoop MapReduce*.

Si usa una struttura chiave-valore, un programma è la definizione di map, reduce ed alcuni parametri. Hadoop gestisce tutto il resto.

La coppia Map e Reduce forma un **Job**. Le operazioni Map e Reduce in un job vengono chiamati **Task**. Un programma può avere + job.

I Task vengono schedulati da YARN ed eseguiti.

Un processo MapReduce consiste in:

1. Input suddiviso in *split*
2. Viene istanziato un task di Map per ogni split.
3. Il risultato viene ordinato e redistribuito, mettendo insieme i vari gruppi in base alla chiave
4. Esecuzione Reduce a partire dai gruppi creati al passaggio precedente
5. Salvataggio in HDFS

Di default il numero di Task di Reduce è deciso in base alle risorse disponibili. Di norma un task di reduce processa più chiavi.

Un Task di Reduce è bene che non sia troppo breve per non sprecare risorse, ma non ci deve nemmeno mettere troppo a fare il suo lavoro.

I risultati dei mapper vengono distribuiti equamente, in base a delle funzioni hash applicate alle chiavi, tra i reducer.

### Combinatori

Aggregare solo con Reduce potrebbe essere poco efficiente.

Si può fare un pre-riduzione a livello di map usando un combinatore (**combiner**), ma solo se la funzione da usare è sia commutativa ed associativa, altrimenti si generano problemi dovuti all'ordine col quale vengono effettuate le operazioni e su quali dati (ad es. la concatenazione di stringhe  $(A+B \neq B+A)$  o la media (media delle medie  $\neq$  media degli elementi) causerebbero problemi).

### Partizionatori

Si può definire una logica con la quale suddividere il lavoro tra i reducer tramite dei **partizionatori**, per decidere quanti task di Reduce usare.

La funzione hash usata dal partizionatore per suddividere i dati può essere definita dall'utente, anche se normalmente quella di default funziona bene.

### Quanti Task di Reduce usare?

- Hadoop di default ne usa 1.
- 1 reduce per ogni chiave: normalmente non fattibile perché le chiavi da processare sono più dei nodi disponibili.
- Un reduce ogni nodo: i valori associati alle chiavi potrebbero non essere equamente distribuiti.
- Molti reduce ogni nodo

Scegliere il numero appropriato di task di reduce è un arte!

Di base, un task di reduce dovrebbe eseguire per ~5 minuti.

## Esecuzione MapReduce

1. Chiamata a *submit* o *waitForCompletion* da parte del programma
2. Contatto RM per ottenere ID
3. Copio tutti i dati che mi servono in HDFS
4. Contatto RM ed eseguo
5. Lo scheduler YARN decide dove lanciare
  - a. alloca un container in un Node Manager
  - b. AMP viene lanciato nel container
6. AMP tiene traccia del progresso del job
7. I dati vengono rimediati da HDFS
8. AMP chiede dei container (al RM) per i task
  - a. prima i task di map
    - i. RM tenta di mantenere il vincolo di data locality (task eseguito nello stesso nodo (*data locality*), o nello stesso rack (*rack locality*), dei dati)
    - b. i reduce vengono chiesti dopo al 5% di completamento dei map
9. AMP fa partire i container assegnati da RM contattando il Node Manager
  - a. il task avvia la JVM per eseguire il programma
10. il task carica tutto il necessario
11. il task viene eseguito

I Map creano un file per ogni Reduce nel disco del suo container e informa AMP.

I Reduce iniziano a ridistribuire i dati dopo una certa percentuale di completamento dei mapper, ma ordinamento e riduzione vengono eseguiti solo dopo aver collezionato tutti i dati.

I Reduce scrivono il loro output in un file.

Se ci sono problemi durante l'esecuzione i task vengono riallocati da AMP. Se AMP fallisce viene riallocato dal RM.

## Algoritmi MapReduce

- **Filtraggio:** trova le tuple con determinate caratteristiche.
- **Sommarizzazione:** trova indicatori riassuntivi dei dati (somma, media, ...).
- **Join:** combinare gli input basandosi su certi valori condivisi.
- **Ordinamento**
  - si sfrutta il fatto che tra map e reduce i dati vengono ordinati. Map e Reduce in questo caso non fanno nulla. Per avere un ordinamento globale bisogna che ci sia un unico reducer.

## MapReduce a 2 stadi

Si possono mettere in cascata diversi Job, però ad ogni fine Job Hadoop salva il risultato su disco, anche se non sarebbe necessario per eseguire il Job successivo.

## Altri Moduli Hadoop



Ambari

Creazione, Gestione e monitoraggio dei cluster.



Oozie

Scheduler dei jobs.



ZooKeeper

Coordinatore delle applicazioni distribuite.



Pig

Piattaforma per analizzare grandi dataset.  
Programmi espressi ad alto livello e compilati  
in job MapReduce.



Sqoop (SQL-to-Hadoop)

Trasferitore di dati tra Hadoop e RDBMS.

Offre la possibilità di importare tabelle da un database ad HDFS e viceversa.

Può lavorare su interi database, su singole tabelle o su porzioni di esse.

L'importazione viene fatta attraverso MapReduce (con solo task di map).

Ciò che viene importato è il risultato di una query SQL.



Hive

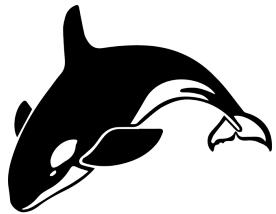
Permette di vedere dati presi da Hadoop sotto forma di tabelle e di scriverci query SQL sopra, che verranno compilate in job MapReduce.

**Database** ⇒ insiemi di tabelle

**Tabelle** ⇒ dati strutturati

**Partizioni** ⇒ parti di tabelle

**Cluster o Bucket** ⇒ parti di partizioni determinate tramite funzioni hash

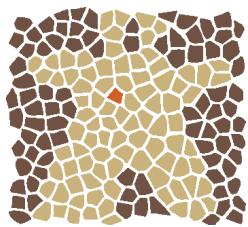


HBase

DBMS distribuito non relazionale.



[Spark](#)



Giraph

Elaborazione di grafi, progettato per essere altamente scalabile.



Kafka

Piattaforma di streaming distribuito. Fa streaming di dati tra applicazioni, oppure li trasforma.



Phoenix

Database relazionale costruito su HBase.



Storm

Computazione in tempo reale.



Mahout

Fornitore di algoritmi scalabili di machine learning.



Impala

Motore SQL.

## Distribuzioni principali

- Cloudera
- MapR
- InfoSphere BigInsights
- Amazon Web Services (AWS)

# Spark

---

Il paradigma resta MapReduce, ma + ottimizzato.

MapReduce (l'implementazione) ha un sacco di limitazioni.

Son cambiate tante cose dalla sua implementazione:

- Prima
  - Dischi come fonte di dati primaria
  - single core
  - OOP
  - nessun framework per storage/processing distribuito
  - analisi SQL
  - poche aziende con Big Data
  - problema di volume dei dati
  - Batch processing
- Adesso
  - RAM come fonte di dati primaria
  - multi core
  - functional programming
  - HDFS, YARN, ...
  - NoSQL
  - tutti usano Big Data
  - problema di velocità delle elaborazioni
  - processing in tempo reale

Per superare le limitazioni di MapReduce nasce Spark.

Non è una versione modificata di Hadoop.

Il suoi vantaggi principali sono:

- **sfruttamento della RAM rispetto al disco** (10-100x + veloce)
- permette diversi paradigmi di analisi (MapReduce, SQL, grafi, ...)
- permette analisi interattive
- **computazione lazy** (elaborazione non effettuata finchè non realmente necessaria)
- evita ridistribuzioni inutili mettendo in pipeline le elaborazioni quando possibile

## RDD (Resilient Distributed Dataset)

Collezioni immutabili di oggetti.

- **Resilienti:** recuperano automaticamente dai guasti.
- **Distribuiti:** gli oggetti di un RDD sono divisi in partizioni e sparsi tra i nodi.
- **Immutabili:** una volta creato un dato, non può essere modificato.
  - ✓ parallelizzazione molto + facile (no race condition)
  - ✓ abilità laziness e cachabilità
  - X maggior occupazione di spazio
- **Valutazione lazy:** non faccio trasformazioni intermedie finché non ne ho realmente bisogno.
  - ✓ ottimizzazione, migliori performance
  - X complesso inferire i tipi
- **Cachabili:** possono persistere in RAM.
  - ✓ Miglioramento drastico delle performance.
- **Inferenza dei tipi:** il tipo di dato è inferito.
  - ✓ feature di Scala, che è il linguaggio con il quale Spark è scritto.

### Creazione RDD

2 vie:

- da un dataset esterno
- distribuendo un'altra collezione

Di default gli RDD non sono persistenti, quindi non vengono cachiati.

Se viene chiesto esplicitamente (comandi *persist* o *cache*), ogni macchina si salva le partizioni in cache in RAM e quando serviranno verranno semplicemente lette.

Si può decidere a che livello cachare

- tutto in RAM
- tutto in RAM, ma usando il disco quando la RAM non basta
- con compressione
- solo disco

### Operazioni

#### Trasformazioni

Definizione di un RDD da un RDD esistente (map, flatMap, filter, ...).

Non vengono eseguite appena dichiarate.

Generano solamente dei metadati che indicano come i dati andranno elaborati (dipendenze, ...).

#### Azioni

Ottenimento di un risultato (count, collect, saveAsTextFile, ...).

Un'azione scatena l'esecuzione di tutte le trasformazioni definite (essendo lazy).

## RDD Lineage

Rappresentazione logica delle dipendenze di un RDD.

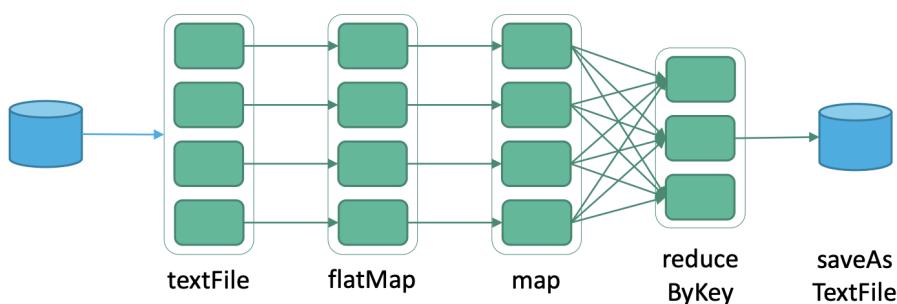
2 tipi di dipendenze:

- **dipendenze larghe**: ogni partizione del RDD padre verrà usata da diverse partizioni del RDD figlio. (serve ridistribuire)
- **dipendenze strette**: ogni partizione del RDD padre verrà usata da al massimo una partizione del RDD figlio. (non serve ridistribuire  $\Rightarrow$  posso mettere in pipeline)

## DAG (Directed Acyclic Graph)

Rappresenta la sequenza di computazioni da eseguire sui dati.

- i nodi sono RDD.
- gli archi sono operazioni su RDD.

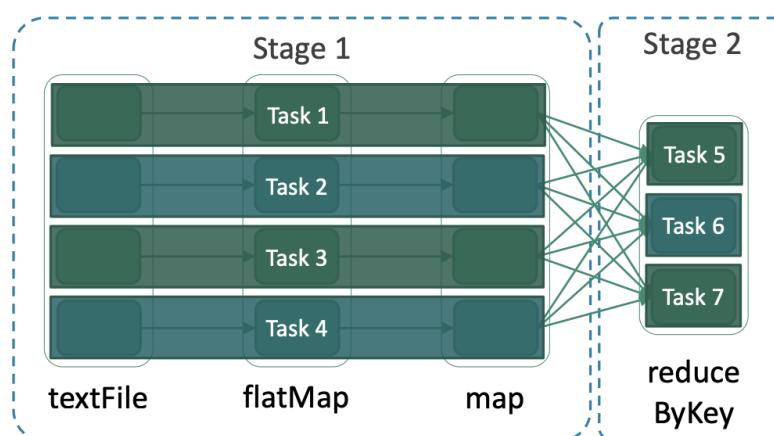


Il piano di esecuzione viene suddiviso in **stage**.

- ogni blocco che può essere eseguito localmente (dipendenze strette) viene messo in uno stage.
- una trasformazione con dipendenza larga causa la creazione di un nuovo stage.

L'unità di esecuzione viene chiamata **task**.

- un task viene create per ogni partizione nel nuovo RDD.



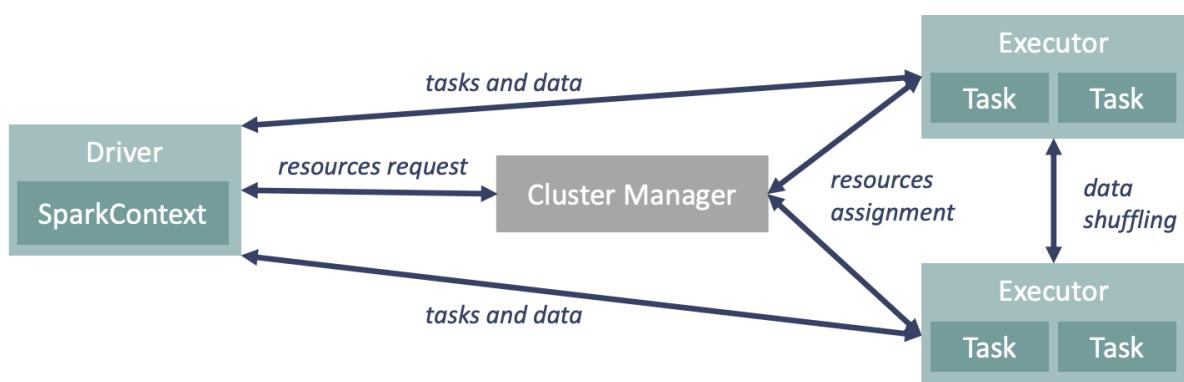
Un'applicazione Spark ha 4 livelli:

1. **Applicazione**: l'insieme delle operazioni da fare
2. **Job**: un'azione da eseguire
3. **Stage**: parti del Job che possono essere messe in pipeline
4. **Task**: parte minima di lavoro di uno Stage.

## Architettura Spark

Usa un'architettura master/slave.

- **Driver:** corrisponde all'applicazione
  - Ogni applicazione ne ha 1
  - Converte il programma in task
  - Schedula i task sugli executor
  - Lancia la WebUI
- **Executor:** corrisponde al container di YARN.
  - Ogni applicazione ne ha diversi
  - Va per tutta la durata dell'applicazione
  - Cacha gli RDD
  - Esegue Task
- **Cluster Manager:** gestisce l'allocazione delle risorse (spesso è YARN).



1. Spark si connette al Cluster Manager
2. Il CM alloca gli Executor secondo indicazione del Driver
3. Il Driver comunicherà direttamente con gli Executor per dar loro Task da eseguire.

Viene istanziato comunque anche l'Application Master Process.

Ci sono 3 modalità per sottomettere applicazioni:

- **modalità Cluster:** il Driver viene allocato all'interno del cluster.
- **modalità Client:** il Driver viene allocato su una macchina esterna.
- **modalità Locale:** Driver ed Executors eseguono sulla stessa macchina.

## Ottimizzazione di Spark

### Partizionamento

Gli RDD sono divisi in partizioni salvate su nodi diversi.

Il partizionamento serve per:

- svolgere le computazioni in modo parallelo
  - incide il numero di partizioni
- minimizzare le comunicazioni di rete
  - incide il criterio di partizionamento

Sono controllate da:

- **Numero di partizioni**
  - Se non specificato, il numero di partizioni viene deciso da Spark.
  - **Un task viene eseguito in ogni partizione**
    - 2-4 partizioni ogni core
- **Criterio di partizionamento**
  - **Definisce come distribuire le chiavi nelle partizioni.**
  - Valori con la stessa chiave finiscono nella stessa partizione.
  - Utile quando un RDD viene riusato con operazioni sulle chiavi.
  - Ripartizionare richiede mescolamento di dati ⇒ **costa!**
  - Ne esistono alcuni predefiniti in Spark come:
    - Hash partitioning: trovo la partizione calcolando l'hash della chiave.
    - Range partitioning: trovo la partizione usando dei range di chiavi.
    - Si può specificarne di personalizzati.

Il criterio di partizionamento è utile quando il dataset è usato da più operazioni sulle chiavi, in modo da evitare ridistribuzioni inutili dei dati.

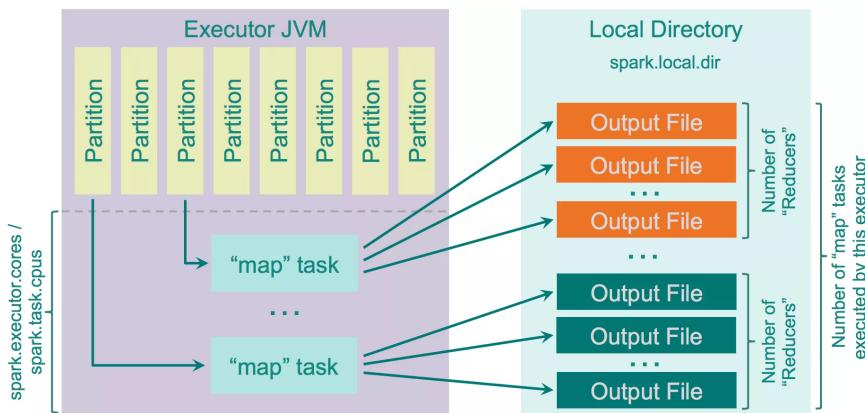
Alcune operazioni (quelle che possono modificare le chiavi, i.e. *map*) possono far perdere il partizionamento dei dati.

### Shuffling

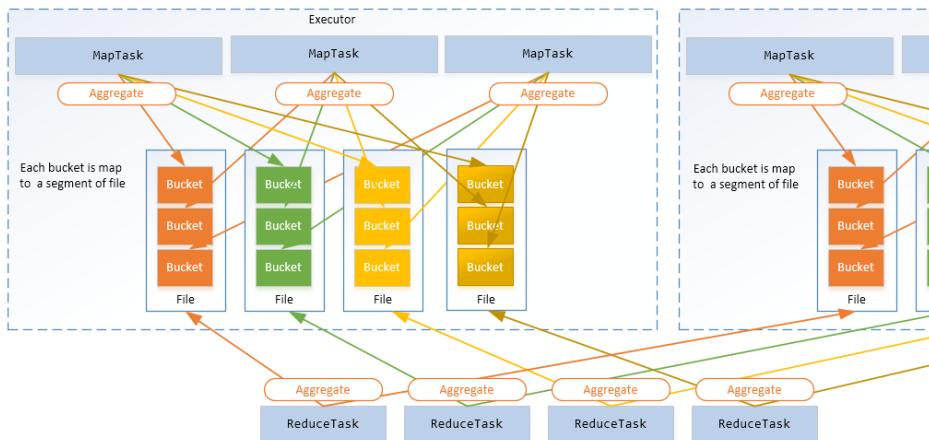
Lo shuffling è la ridistribuzione dei dati tra le partizioni attraverso la rete, utile per fare diverse operazioni. Si tratta di un'operazione costosa e complessa.

Su Spark ci sono diverse tecniche per fare shuffling:

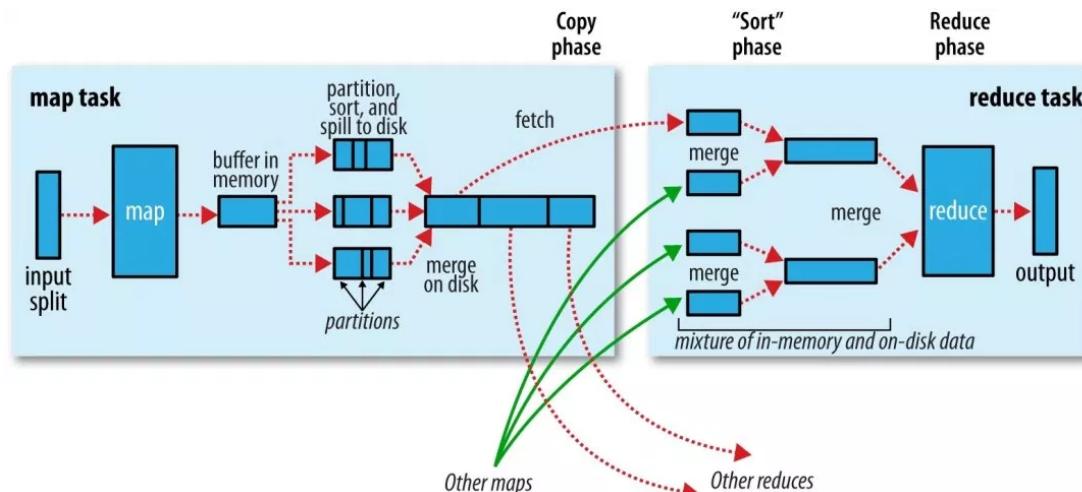
- **Hash shuffle:** ogni task di map crea un file di output per ogni reducer.



- **Hash shuffle evolved:** si usa lo stesso file di output da + task di map. Bisogna fare attenzione alla scrittura in contemporanea.



- **Sort shuffle:** per ogni task di map si crea un unico file di output con i dati ordinati per partizione. Servirà un indice per capire dove sono le varie parti che serviranno poi ai reduce.



- **Tungsten sort:** come Sort Shuffle, ma usa dati serializzati (meno spazio in memoria), permettendo di riordinare i dati senza deserializzare, quindi velocizzando il passo di sort. Questo meccanismo non permette l'utilizzo di combiner, che richiederebbero una deserializzazione, perdendo così il vantaggio.

La tecnica migliore viene scelta da Spark in automatico. Normalmente usa il Sort shuffle. Se ci sono poche partizioni usa l'Hash Shuffle. Se possibile usa il Tungsten sort.

## Serializzazione

La serializzazione serve per salvare dati su file ed inviarli.

Serializzare dati riduce l'utilizzo di memoria e I/O, ma costa di + in termini di CPU.

Spark usa di default la serializzazione di Java, ma viene suggerito di usare .

## Configurazione del Cluster

Ci sono 2 principali risorse, la **CPU** e la **memoria**.

Ogni executor ha le stesse risorse degli altri:

- numero di core
- dimensione dell'heap

Anche il numero di executor è normalmente fisso, almeno che non sia attiva l'allocazione dinamica.

### CPU

Si possono settare:

- **numero di executor** ⇒ `--num-executors`
- **cores per ogni executor** ⇒ `--executor-cores`

L'accesso concorrente ad HDFS da parte di più task all'interno di un executor è problematico.

L'AMP richiede risorse, che vanno tenute in considerazione.

Vanno sempre lasciate abbastanza risorse ai vari nodi per poter eseguire i vari servizi.

Indicativamente:

- `--executor-cores` = da 3 a 5
- `executors per node` =  $\frac{\#cores - 1}{--executor-cores}$
- `--num-executors` = #nodi \* executors per node - 1

### Memoria

Per quanto riguarda la memoria, si può impostare la **quantità di memoria per ogni executor** ⇒ `--executor-memory`.

In realtà Spark alloca il 10% in più rispetto a quella richiesta per gestire internamente

Indicativamente:

- `--executor-memory` =  $\frac{memoria\ disponibile * 0.75 * dimensione\ heap}{executors\ per\ node}$

## Variabili Condivise

In realtà non ci sono variabili condivise per mantenere le operazioni indipendenti tra di loro. Se si tenta di usare la stessa variabile, ogni task ha la propria copia locale.

Spark permette di condividere variabili tramite 2 meccanismi:

- **Variabili broadcast:** condivisione in sola lettura di certe variabili da parte degli executor. Utile principalmente per condividere in lettura files grandi o quando una stessa variabile è usata in diversi punti.
- **Accumulatori:** condivisione in sola scrittura, aggiornabile dai vari task. Utile principalmente per debug. Se per un qualche motivo un task deve essere eseguito più volte, l'accumulatore non ha modo di distinguere questi casi e viene aggiornato più volte.

## Spark SQL

È un modulo di Spark progettato per funzionare su dati strutturati e semistrutturati.

Usa una nuova astrazione basata sugli RDD, il **DataFrame**.

Permette di manipolare DataFrames in diversi linguaggi: Scala, Java o Python.

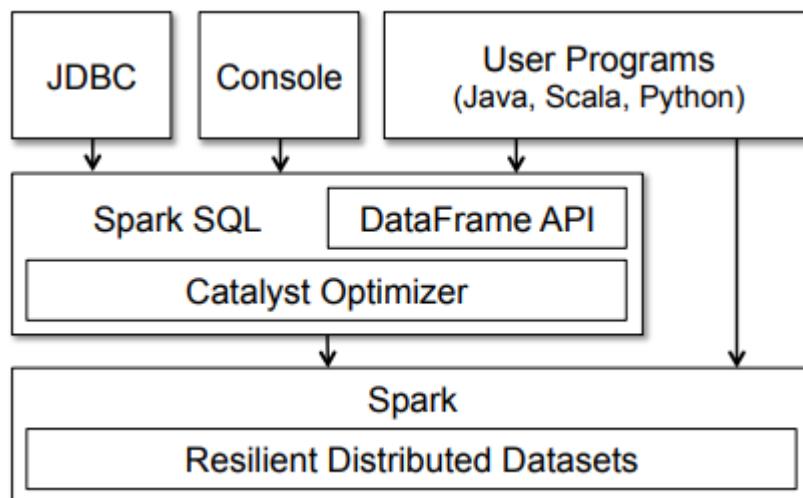


Spark SQL mira a:

- Supportare elaborazioni relazionali sia su Spark sia su sorgenti esterne.
- Fornire risultati con alta efficienza usando la logica dei DBMS.
- Supportare facilmente nuove sorgenti di dati.
- Permettere l'estensione con tecniche di analisi avanzate.

## Features

- **Integrato**: è possibile mescolare query SQL con Spark.
- **Accede allo stesso modo ai dati** da diverse sorgenti.
- **Compatibile con Hive**.
- Ha **connettività standard**, ci si può connettere con JDBC.
- **Scalabile**: lo stesso motore funziona sia in modalità interattiva sia per interrogazioni lunghe.



## Utilizzo

Spark SQL si basa sul concetto di **SQLContext** (basato sullo *SparkContext*), e da lì si potranno poi creare i DataFrame.

**DataFrame** ⇒ collezione distribuita di dati organizzata in colonne. Equivalente ad una tabella con ottimizzazioni.

I DataFrame sono dati distribuiti organizzati come tabelle.  
Si possono trasformare DataFrame in RDD e viceversa.

## Ottimizzatore Catalyst

Ottimizzatore che agisce nella maniera + automatica possibile.

**2 tipi di ottimizzazione, entrambe applicate:**

- basata su regole
- basata sul costo

### Alberi e regole

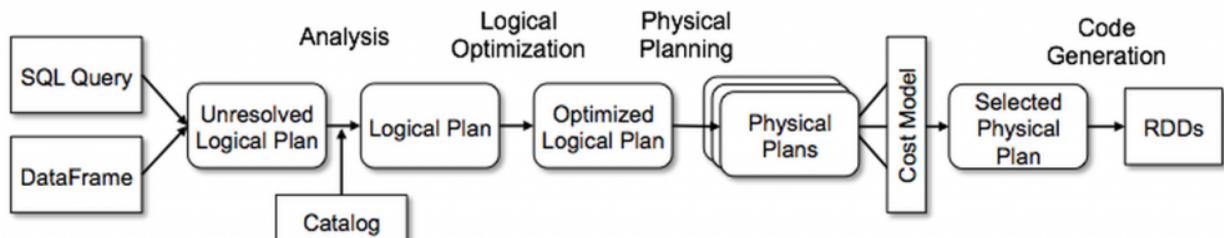
Catalyst si basa sull'utilizzo di **alberi**.

I nodi di tali alberi sono immutabili e possono essere manipolati dalle **regole**.

Le regole prendono in ingresso un albero e restituiscono un altro albero. L'input viene rimpiazzato con il risultato.

La trasformazione avviene ricorsivamente su tutti i nodi dell'albero applicando pattern matching.

Catalyst raggruppa le regole in **batch** e li esegue finché non raggiunge un **punto fisso**, cioè quando l'albero non cambia più applicando le regole.



### Analisi

Data una query SQL o un DataFrame, costruisce un piano logico non risolto, cioè con attributi e tipi slegati.

Applica regole per:

- trovare relazioni dal catalogo
- mappare gli attributi agli input
- determinare quale attributo si riferisce allo stesso valore per dare ID unici
- propagare e forzare i tipi attraverso espressioni

### Ottimizzazione logica

Ottimizzazione tramite regole.

- constant folding

- predicate pushdown
- null propagation
- boolean logics simplification
- ...

### Pianificazione fisica

Dato un piano logico, si generano uno o più piani fisici, poi sceglie il migliore usando un modello di costi.

Le operazioni di join sono quelle che incidono maggiormente.

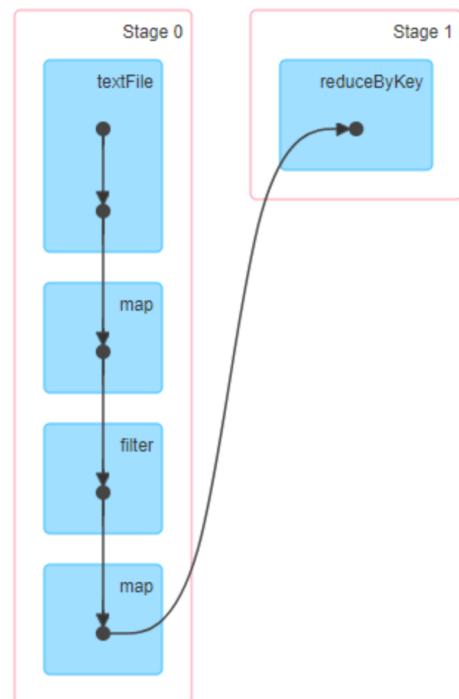
Esistono diversi modelli di join:

- **Shuffle Hash Join**: modello usato di default da Spark SQL. Richiede invio di dati in rete, quindi è costoso.  
Fa in modo che i dati con la stessa chiave finiscano nella stessa partizione.
- **Broadcast Hash Join**: usate per aumentare le performance, nel caso in cui una tabella sia molto più piccola dell'altra.  
In questo caso la tabella piccola viene inviata a tutte le partizioni dell'altra.

Tramite il comando `explain` è possibile vedere i piani (logico e fisico) generati.

## RDD Lineage

Rappresentazione grafica del piano eseguito da Spark.



# Data Streaming

Big Data non consiste solamente di analisi batch, ma anche di analisi di stream di dati.

Esistono diversi casi nei quali si usano streams:

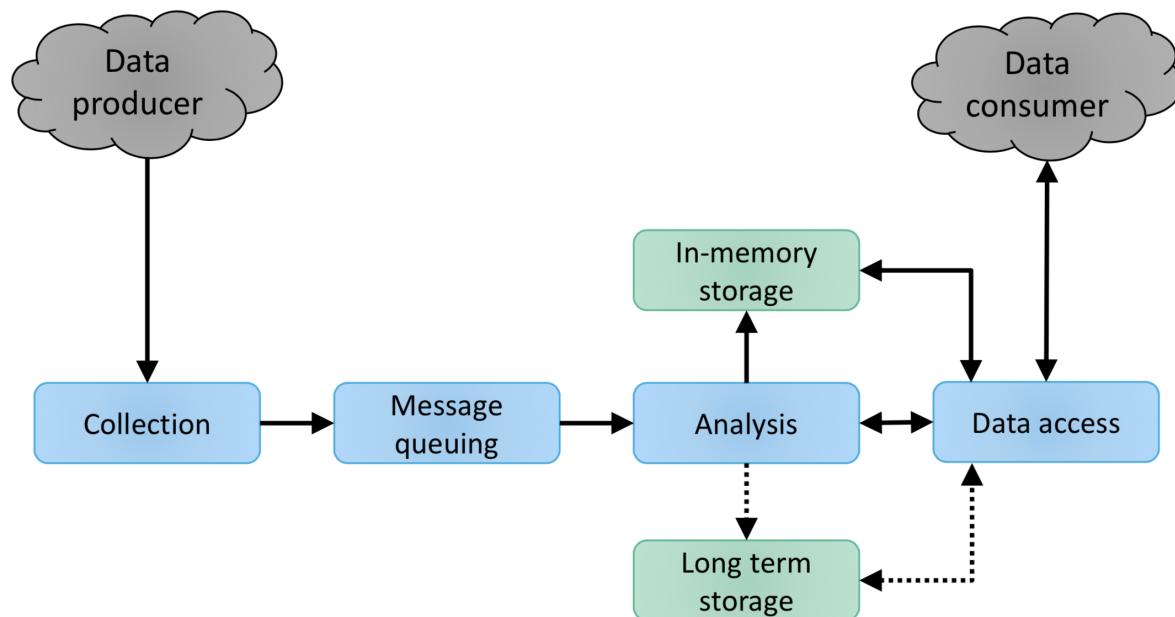
- Internet of Things
- Social Media
- Web Analytics
- Operational Monitoring
- ...

Spesso fare analisi **streaming ricade nella categoria soft real-time** (se il risultato arriva con ritardo non succede nulla di male).

**Un sistema per data streaming è progettato per lavorare con dataset infiniti.**

- **dataset infiniti**: dati generati continuamente e non si ha controllo sull'ordine di arrivo
- **computazione infinita**: il sistema è sempre attivo e deve evitare overflowing (troppi dati, non si riesce a processarli tutti)
- **algoritmi approssimati**: serve dare risposte molto rapidamente, quindi si ricorre ad algoritmi approssimati

## Architettura



## Collezione

- composto da server edge che ricevono dati da fonti esterne
- diversi pattern di interazione (*publish/subscribe, request/response, ...*)
- diverse strategie per tolleranza agli errori
- diversi formati per i dati

## Message Queuing

Gestisce la trasmissione dei dati dal livello di collezione a quello di analisi e da analisi a data access.

- disaccoppiamento delle operazioni, ogni nodo del cluster fa 1 cosa
- comunicazioni sicure tra nodi
- tunneling tra n sorgenti ed n consumatori
- Producer-Broker-Consumer
  - Il livello di collezione è il Producer
  - Il Message Queuing è il Broker
  - Il livello di analisi è il Consumer
- Per inviare i dati al livello di analisi si possono individuare 3 livelli di garanzia:
  - **Exactly once**: ogni messaggio viene letto una sola volta e non viene mai perso.
  - **At most once**: un messaggio può perdersi, ma non verrà mai letto 2 volte (potenzialmente non tutti i dati arrivano al consumer).
  - **At least once**: un messaggio non verrà mai perso, ma può essere letto più volte (tutti i dati vengono letti certamente dal consumer).

## Analisi

Si analizzano i dati che arrivano in streaming

- **modello di query continuo**
  - **job lanciato e continuamente eseguito sui dati che arrivano**
  - **Vincoli di memoria**
    - memoria disponibile limitata
    - scrivere su disco impattierebbe sulle performance
    - una volta processati, i dati vengono scartati
    - devono essere usati algoritmi di sommarizzazione per salvare i risultati.
  - **Vincoli di tempo**
    - I dati che non possono essere processati in tempo vanno scartati.
- **algoritmi specifici per il problema**
- **job potenzialmente idempotenti**: se il job riceve lo stesso messaggio, il risultato prodotto è lo stesso.

## Modelli di dati

- **Serie temporale:** ogni evento rappresenta il nuovo stato dell'elemento
- **Registratore di cassa:** ogni evento è un incremento rispetto al valore precedente.
- **Tornello:** ogni evento è un aggiornamento (come registratore di cassa, ma ci possono essere anche decrementi)

## Windowing

L'analisi può essere fatta in due modi:

- ogni elemento alla volta
- a **finestre**.

Le finestre sono definite da **lunghezza** (quanti elementi contengono) e **periodo** (ogni quanto analizzare dati).

Esistono 2 tipi principali di finestre:

- **Sliding Windows:** lunghezza e periodo sono definiti in base al tempo.
  - Fixed Windows ⇒ periodo = lunghezza
  - Overlapping Windows ⇒ periodo < lunghezza.
  - Sampling Windows ⇒ periodo > lunghezza
- **Data-driven Windows:** lunghezza definita dal contenuto dei dati.
  - Il periodo determina l'intervallo di aggiornamento.
  - La lunghezza non può essere definita a priori.

Si possono fare finestre basandosi su

- **Stream Time** (quando l'evento entra nel sistema)
  - ✓ Di semplice implementazione
  - ✓ Finestre sempre complete
  - X Il tempo di generazione dell'evento non può essere usato
  - X Non c'è alcuna garanzia sull'ordine di arrivo
- **Event Time** (quando l'evento viene generato)
  - ✓ È lo standard
  - X Non è possibile dire con certezza quando chiudere una finestra.
  - X Finestre grandi necessitano di molto buffering dei dati
  - X Molti sistemi non lo supportano
  - Richiede la definizione di:
    - **Watermark:** asserzione del fatto che non arriveranno più dati generati entro una soglia.
    - **Triggers:** intervalli di elaborazioni intermedie, validi fino al raggiungimento del watermark.
    - **Latenza accettata:** quali dati in ritardo rispetto al watermark accettare.
    - **Strategia di accumulazione:** come aggregare i risultati intermedi.
      - Discarding Strategy: i nuovi valori sono indipendenti dai precedenti.
      - Accumulating Strategy: i nuovi valori si aggregano ai precedenti.

- Accumulating and Retracting: i nuovi valori contengono il totale e la differenza.

La differenza tra Stream ed Event time viene detta **skew**.

## Algoritmi

Gli algoritmi di streaming hanno le seguenti caratteristiche:

- **One-pass**: dati esaminati una sola volta
- Uso di **poca memoria**
- **Aggiornamenti veloci**
- **Risposte veloci**, ma **approssimate con certe garanzie** (% di probabilità che il risultato sia in un certo intorno del risultato reale)

2 concetti fondamentali:

- **Campionamento**
- **Proiezione Random**: convertire il dato in un formato compresso, ma che permette di farci analisi.

## Random Sampling

Si vuole fare qualche analisi statistica su un campione significativo di dati.

Uno degli algoritmi più usati è il **reservoir sampling** (campionamento con riserva): si mantiene un determinato numero di valori, quando arriva un nuovo elemento nello stream si decide se aggiungerlo al campione o scartarlo.

1. I primi r elementi che arrivano vengono inseriti nella riserva.
2. Per i successivi, si decide se inserirli in base ad una probabilità

$$P = \frac{\text{dimensione della riserva}}{\text{indice dell'elemento}}$$

Se risulta che l'elemento nuovo è da aggiungere, l'elemento da togliere per far posto viene scelto casualmente.

## Contare gli elementi distinti

Si usa una funzione hash che trasforma gli oggetti in entrata in una distribuzione finita.

Ci sono 2 categorie di algoritmi:

- Bit-pattern: basati sull'osservazione di pattern di bit
  - **HyperLogLog**
    - La cardinalità di un multiset di numeri casuali uniformemente distribuiti può essere stimata calcolando il **numero massimo di leading zeros** (quelli a destra, prima di una qualsiasi altra cifra)
    - Il numero di elementi è dato da  $2^{\#leading\ zero}$
    - Per mitigare potenziali problemi, si spezza lo stream in m sottoinsiemi, si tengono m contatori e si fa la media armonica.
    - Ogni contatore deve essere grande  $\log(\log(N))$  [N = limite superiore stimato]
    - Più contatori si usano, più la stima sarà accurata.
  - **HyperLogLog++**
    - Versione ottimizzata di HyperLogLog
- Order statistics: basati sulla valutazione dell'ordine statistico

## Membership

Verificare se un elemento è già apparso nello stream.

Si usano i **bloom filters** per risolvere il problema.

- Si ha un array di bit, ogni elemento viene processato da n funzioni hash e si settano ad 1 gli elementi corrispondenti dell'array.
- Se tutte le celle indicate dalle funzioni hash sono ad 1, allora un elemento è già apparso.
- È possibile che diano falsi positivi, ma non danno mai falsi negativi (se un elemento non è comparso, allora dice che non è comparso).
- Il numero di funzioni hash ottimale dipende dalla probabilità di falsi positivi che si vuole accettare.

## Frequenza

Determinare quante volte un elemento è comparso nello stream.

Anche in questo caso si frutto i bloom filter, l'algoritmo più usato è **Count-Min Sketch**.

- Si hanno tanti array di contatori, ognuno aggiornato solamente da 1 funzione hash.
- Le celle risultanti dalle funzioni hash vengono incrementate.
- Il numero di occorrenze di un dato elemento è dato dal minimo tra i valori indicati dalle funzioni hash.
- È possibile che il conteggio sia maggiore del numero effettivo, ma mai minore.
- Il numero di funzioni hash e di contatori dipende dalle garanzie che si vogliono garantire.

## Storage dei dati

Che fine fanno i dati una volta analizzati

- scartati
- salvati su disco ⇒ per farci analisi batch in seguito
- messi in RAM
- rimessi nella pipeline

## Accesso ai dati

Fornisce i dati analizzati ai consumatori.

## WebHook

Come le callback di molti linguaggi di programmazione.

Il client registra la sua callback, che verrà chiamata ogni qualvolta ci saranno dati da fornire.

## HTTP Long Polling

Il client crea una connessione con il server, che resta aperta. Appena i dati diventano disponibili gli vengono mandati.

## Server-sent events

Come il precedente, ma senza creare connessioni. Il server manda info al client di sua iniziativa.

## WebSockets

Connessione tra client e server a 2 vie tramite TCP.

# Database NoSQL (Not only SQL)

NoSQL indica quei DBMS che usano modelli diversi da quello relazionale.

	RDBMS	NoSQL
benefici	<b>Transazioni</b> Garanzia in termini di consistenza ed accessi concorrenti	<b>Non solo righe e tabelle</b> I dati sono memorizzati secondo vari modelli.
	<b>Integrazione</b> Molte applicazioni possono condividere e riutilizzare dati	<b>Libertà dai Join</b> I join sono sconsigliati o non supportati.
	<b>Standard</b> Il modello relazionale e SQL sono standard.	<b>Libertà dalla rigidità</b> I dati possono essere memorizzati senza predefinire uno schema.
	<b>Robustezza</b> Usato da 40 anni!	<b>Distribuito</b> Scala senza problemi.
debolezze	<b>Disaccoppiamento di impedenza</b> I dati sono memorizzati secondo il modello relazionale, ma le applicazioni li usano secondo il modello ad oggetti.	
	<b>Scalatura complessa</b> Non sono pensati per scalare, soprattutto nel distribuito.	
	<b>Consistenza vs Latenza</b> La consistenza è sempre garantita, anche a spese della latenza.	
	<b>Rigidità</b> Cambiare lo schema è costoso.	

I NoSQL vengono usati principalmente per gestire OLTP (molte letture/scritture su pochi dati).

Tecnologie Big Data vengono usate principalmente per OLAP (lettura su molti dati).  
 NoSQL è come uno storage di dati operazionali per tecnologie Big Data.

## Modelli di dati

### Aggregati

Creare degli aggregati che contengano tutte le informazioni necessarie per fare le operazioni più comuni. Un aggregato è un gruppo di oggetti gestiti come un unico blocco. Questo permette di limitare il numero di join necessari, aumentando l'incapsulamento. Tutto ciò semplifica la distribuzione e il lavoro dello sviluppatore, ma non esistono strategie consolidate per la loro definizione, ci si può basare sulle interrogazioni che andranno fatte sui dati per la modellazione.

### Chiave-Valore

Un DB contiene una o più collezioni (tabelle).

Ogni collezione ha un elenco di coppie chiave-valore, con chiave univoca.

All'interno di una chiave, i suoi valori vengono modificati in modo atomico.

Le operazioni fattibili sono:

- Aggiunta di 1 chiave
- Ritrovamento di 1 chiave
- Cancellazione di 1 chiave

I DB chiave-valore più usati sono:

- Radis
- Memcached
- DynamoDB

È bene evitarli quando i dati sono molto interrelazionati, quando serve fare interrogazioni sui dati (solo la chiave è visibile) o quando serve fare operazioni su più chiavi.

### Documentale

Un DB contiene una o più collezioni (tabelle).

Ogni collezione contiene dei documenti (tipicamente in JSON).

Ogni documento contiene dei campi, con ID obbligatorio.

Ogni campo è strutturato come chiave-valore.

Le operazioni fattibili sono:

- Creare indici
- Creare filtri sui campi
- Creare più documenti con 1 query
- Selezionare solo certi campi
- Aggiornare solo certi campi

I DB Documentali più usati sono:

- MongoDB

È bene evitarli quando serve fare transazioni o query su dati con struttura eterogenea.

### Wide Column

Un DB contiene una o più famiglie di colonne.

Ogni famiglia contiene delle righe chiave-valore.

Ogni colonna all'interno delle righe è a sua volta una coppia chiave-valore.

Le interrogazioni fattibili sono:

- L'uso di indici è scoraggiato
- Creare filtri sulle colonne (non sempre!)
- Ritornare più righe con 1 query
- Selezionare solo certe colonne
- Aggiornare solo certe colonne

I DB colonnari più usati sono:

- Cassandra
- HBase
- Google BigTable

È bene evitarli quando sono richieste transazioni ACID.

## Grafo

Un DB contiene uno o più grafi.

Ogni grafo contiene nodi ed archi.

I nodi rappresentano le entità, gli archi le relazioni.

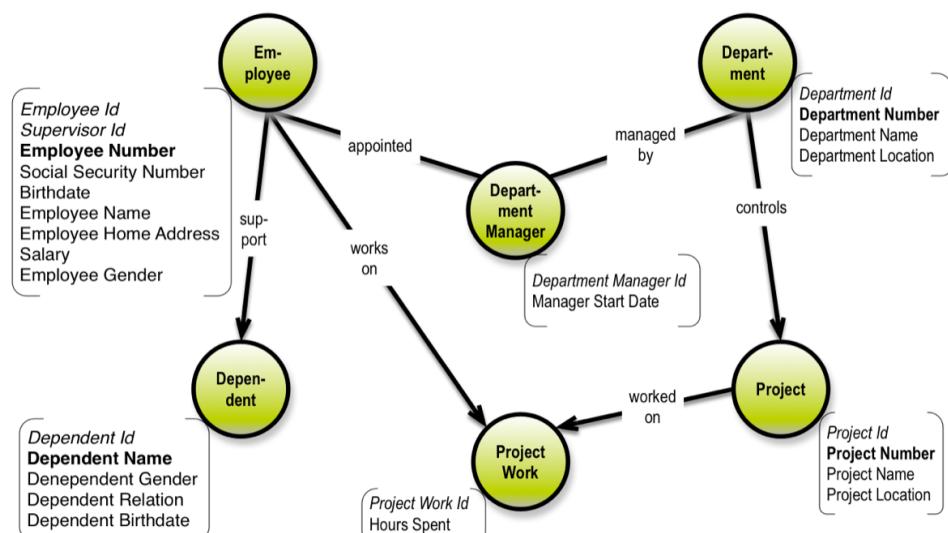
Nodi ed archi sono descritti dalle loro proprietà.

Le interrogazioni vengono fatte sulla base di pattern, che verrà ricercato nel grafo.

I DB a grafo più usati sono:

- Neo4J

È bene evitarli quando si hanno delle query che accedono a tutto il grafo.



## Consistenza

I database NoSQL rispondono a proprietà diverse da ACID:

- **BASE**
  - **Basic Availability**: il sistema deve essere sempre disponibile
  - **Soft-state**: è accettabile che il sistema sia temporaneamente inconsistente
  - **Eventual Consistency**: prima o poi tutto tornerà consistente

- **CAP theorem:** Solo 2 delle 3 proprietà possono essere garantite contemporaneamente.
  - **Consistency:** il sistema è sempre consistente
  - **Availability:** il sistema è sempre disponibile
  - **Partition Tolerance:** il sistema può soffrire di partizionamento di rete
  - È possibile usare una versione rilassata del CAP, che usa cezioni meno restrittive di consistenza e disponibilità, permettendo inconsistenza per brevi periodi.  
Utilizzabile solo in determinati scenari.
- PACELC theorem: evoluzione del CAP
  - Se c'è partizionamento di rete, allora il sistema deve scegliere tra Disponibilità e Consistenza.
  - Se non c'è deve scegliere tra Latenza e Consistenza.

## Sharding

Distribuire i dati nei diversi nodi.

Fondamentale adottare un criterio di suddivisione sensato rispetto al problema.

Le regole da seguire per definire una strategia di sharding sono:

- **Data locality:** i dati vanno messi vicino agli utenti che dovranno usarli
- **Mantenere una distribuzione bilanciata:** il carico di lavoro nei vari nodi deve essere pressoché lo stesso.
- **Mettere insieme i dati che saranno usati insieme.**

Molti sistemi NoSQL offrono policy di **auto-sharding**, cioè suddividono i dati in base al carico di lavoro.

## Replication

Creare copie dei dati in diversi nodi.

Esistono 2 tecniche per fare repliche:

- **Master-Slave**
  - Un nodo assume il ruolo di gestore dei dati, gestendo tutte le scritture
  - Gli altri garantiscono solo letture di dati, sempre sincronizzati col master
  - ✓ Letture gestite facilmente
  - ✓ Funziona bene se il carico di lavoro consiste principalmente in letture
  - ✗ Il master è il collo di bottiglia, in quanto deve gestire tutte le scritture
  - ✗ La propagazione delle scritture è fonte di inconsistenza, in quanto non immediata
  - ✗ Non funziona se il carico di lavoro consiste principalmente in scritture.
- **Peer-to-Peer**
  - Ogni nodo gestisce anche le sue scritture
  - ✓ Il fallimento di un nodo non interrompe nulla
  - ✓ Le performance in scrittura aumentano
  - ✗ **Inconsistenza!**
    - ✗ La propagazione delle scritture è fonte di inconsistenza, in quanto non immediata
    - ✗ 2 aggiornamenti possono avvenire in contemporanea, peggiorando di molto l'inconsistenza

**La replicazione in sé è fonte di inconsistenza.**

Utenti diversi possono leggere dati diversi (normalmente questa situazione è tollerata). Si adotta una politica del tipo “**read your writes**”.

I conflitti in scrittura vengono gestiti nei seguenti modi:

- **Last write wins**: l'ultimo che scrive sovrascrive i precedenti.
- **Conflict prevention**: prima di scrivere, leggo il dato per sincerarmi del fatto che non sia stato modificato.
- **Conflict detection**: il DB si accorge del conflitto e lo segnala.

Soltamente esiste il meccanismo del **quorum**, cioè viene sfruttato il fatto che il dato è replicato per rispondere a certe query in modo consistente.

Vengono definite 2 soglie, una per scrivere ed una per leggere, che indicano il numero minimo di repliche del dato voluto per poter effettuare un'operazione.

Dato il fattore di replicazione N, il quorum in scrittura (W) e quello in lettura (L) devono essere:

$$W > N/2 \quad L > N-W$$

## **Persistenza poliglotta**

Usare un unico DBMS per gestire tutto porta a soluzioni inefficienti.

Usando più DB per gestire diverse applicazioni può portare a miglioramenti, anche se poi sarà più complesso fare analisi.

Un modo potrebbe essere quello di usare DB NoSQL come supporto per migliorare le prestazioni di un RDBMS.

Conviene usare DB NoSQL rispetto a RDBMS quando è necessario scalare e distribuire i dati, per migliorare le performance quando non si hanno grossi vincoli di consistenza immediata.

# Big Data Mining

Molti algoritmi di data mining non sono semplici e non sono stati pensati per essere eseguiti in parallelo.

**Ripensare un algoritmo per renderlo parallelizzabile è una sfida.**

Un modo per modellare algoritmi parallelizzabili è seguire la metodologia PCAM.

## Partitioning, Communicating, Agglomerating, Mapping (PCAM)

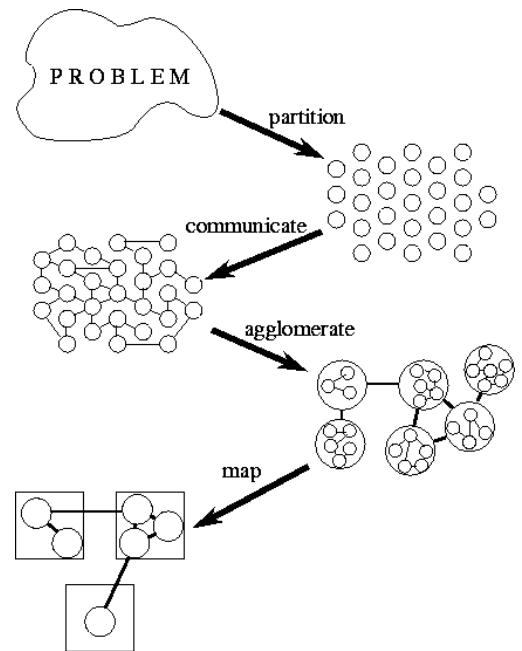
Metodologia per promuovere un **approccio esplorativo** al **design** nel quale i problemi indipendenti dalla macchina vengono considerati prima.

### Partizionamento

Divide la computazione e i dati.

2 approcci:

- **Decomposizione di dominio:** prima decomponi i dati e poi associa la computazione
- **Decomposizione funzionale:** prima dividi la computazione poi associa i dati ai task



### Comunicazione

Specifica il flusso delle informazioni tra task.

4 pattern:

- Locale / Globale
- Strutturato / Non strutturato
- Statico / Dinamico
- Sincrono / Asincrono

### Agglomerazione

Rivedi le decisioni prese in fase di partizionamento e comunicazione.

Potrebbe essere utile agglomerare task per ragioni di efficienza.

Potrebbe essere utile replicare dati.

### Mappatura

Specifica il processore nel quale ogni task verrà eseguito.

2 strategie:

- Aumenta la concorrenza piazzando i task in processori differenti
- Aumenta la località piazzando i task nello stesso processore.

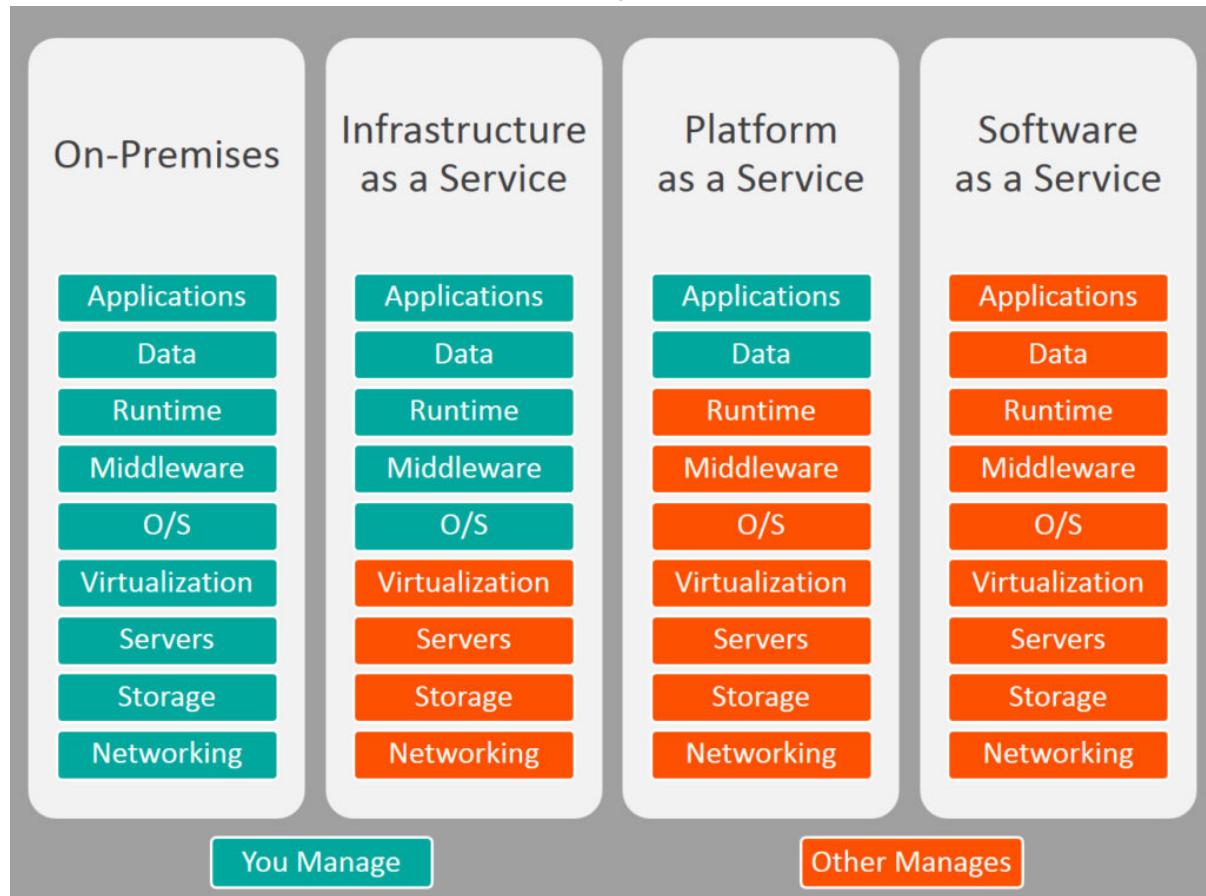
### Conclusioni

In un framework Big Data molte delle problematiche sono già state gestite:

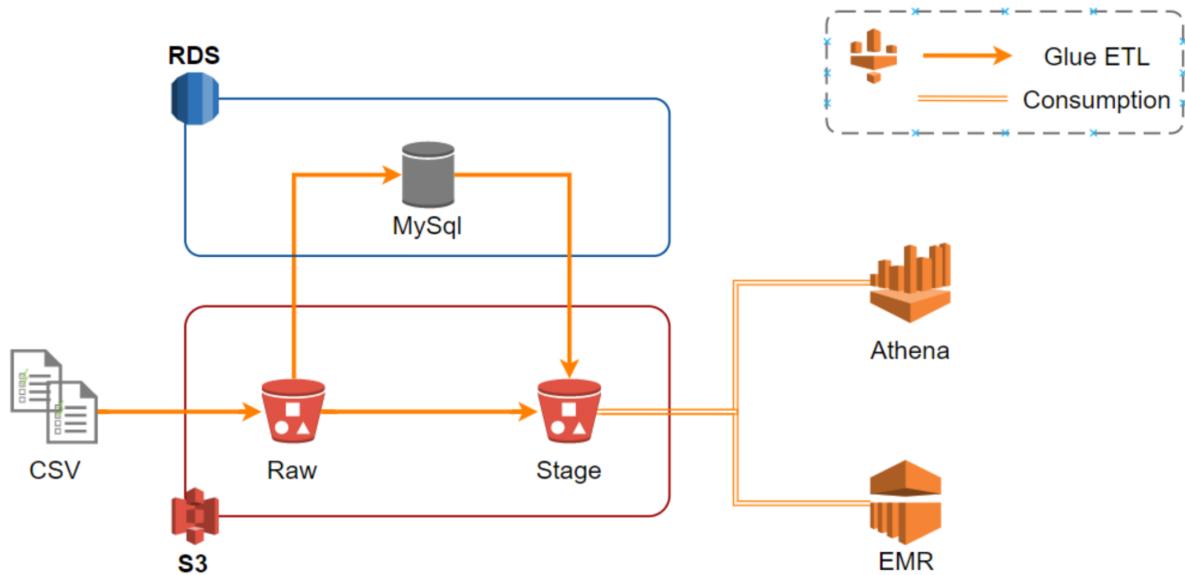
- Il partizionamento viene fatto con decomposizione del dominio
- La comunicazione non viene modellata
- Non serve agglomerare
- Il mapping è automatico.

# Cloud Computing

Cose fornite tramite la rete senza difficoltà nel gestirle da parte di chi ne vuole fare uso.



## Amazon Web Services (AWS)



## Storage



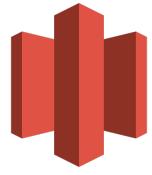
### Simple Storage Service (S3)

Storage di qualsiasi tipo di dato

Offre

- Scalabilità
- Disponibilità
- Durabilità
- Storage a basso costo

Usa **bucket** come base dove poter mettere dati. Il nome del bucket deve essere univoco a livello globale.



### Glacier

## Database



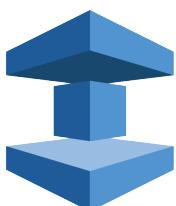
### DynamoDB



### Relational Database Service (RDS)



### Redshift



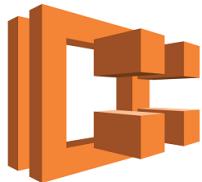
### ElastiCache

## Computazione



**Elastic Compute Cloud (EC2)**

Una macchina con processori, RAM e storage. Tutti i servizi Amazon richiedono di istanziare macchine EC2 per fare cose.



**Elastic Container Service (ECS)**

Servizio per la gestione di applicazioni containerizzate.



**Lightsail**

Sta sopra EC2, fornendo la possibilità di usare GUI.



**Lambda**

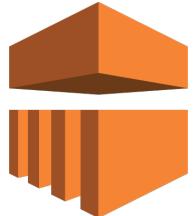
Base per tutti i servizi che non necessitano di server per funzionare.

## Analisi



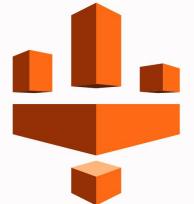
Athena

Permette di eseguire SQL su tabelle (gestiti internamente come job).



Elastic MapReduce (EMR)

Servizio per gestione del paradigma MapReduce.



Glue

Mondo ETL (extract-transform-load).

- Serverless
- permette di definire job Spark senza scrivere codice
- I componenti principali sono:
  - **Data Catalog**
  - **Crawler**: scansionatori di storage, che scriveranno poi i metadati trovati nel data catalog.
  - **Job**: accedono al data catalog ed agiscono sui dati. Possono applicare trasformazioni.

# Data Science

Estrazione di conoscenza dai dati.

## Data Scientist

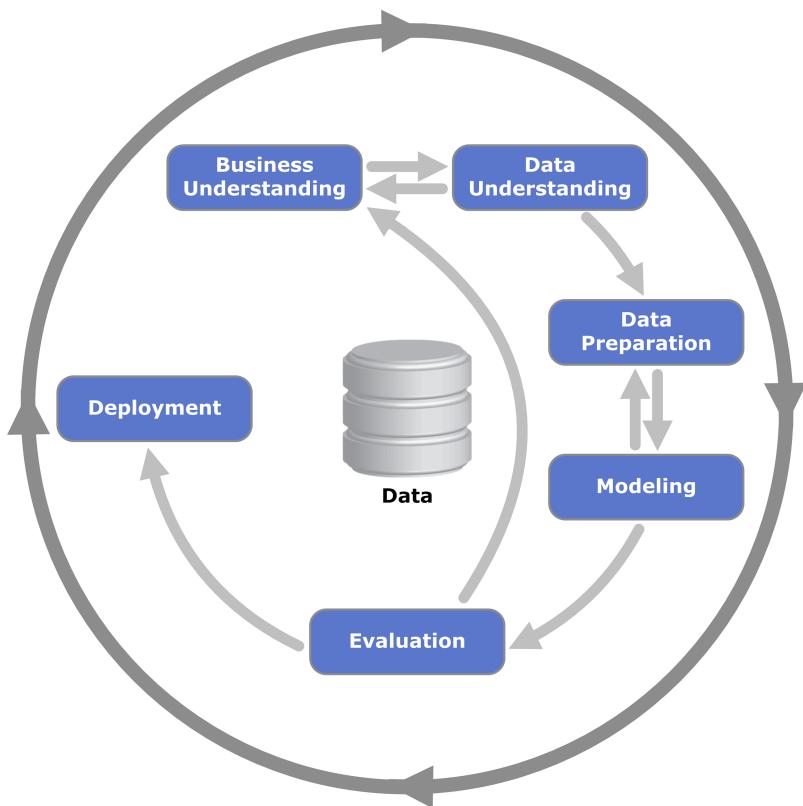
Colui che estrae conoscenza dai dati e la rende fruibile.

Le capacità utili sono:

- **Matematica e statistica**
- **Programmazione e database**
- **Conoscenza del dominio**
- **Comunicazione e visualizzazione**

Un approccio possibile è CRISP-DM (Cross Industry Standard Process for Data Mining)

## CRISP-DM



### Comprensione del business

- Capire gli obiettivi da una prospettiva di business.
- Trasformare i requisiti in problemi di data mining
- Fare piani preliminari

## Comprensione dei dati

- Collezionare dati
- Capirne il significato
- Valutarne la qualità
- Fare ipotesi

## Preparazione dei dati

- Pulizia dei dati
- Miscelare le diverse sorgenti
- Scelta delle feature interessanti
- Creazione nuove feature

## Modellazione

- Applicazione di tecniche di modellazione per risolvere il problema
- Calibrazione per raggiungere un risultato ottimale

## Valutazione

- Determinare se l'obiettivo è stato raggiunto
- Assicurarsi che tutto funzioni come dovrebbe
- Valutare la capacità del modello di funzionare con dati nuovi

## Distribuzione

- Rendere disponibili i risultati alle giuste persone

# Machine Learning in Spark

---

Gli algoritmi vengono appresi e non scritti direttamente dal programmatore.

Prima di agire potrebbero essere necessarie normalizzazioni ed aggiustamenti dei dati.

Si usa la libreria MLlib.

Fornisce algoritmi di machine learning per:

- Clustering
- Regressione
- Classificazione
- Estrazione di Feature
- Riduzione di dimensionalità
- ...

## Valutazione

L'errore non è abbastanza per stabilire che un modello è buono.

Per stabilire ciò si suddividono i dati in **training** e **test set**, i dati di training verranno usati durante l'algoritmo, quelli di test vengono usati solamente per valutare i risultati ottenuti tramite il training set.

**Underfitting:** l'algoritmo non è abbastanza complesso per spiegare i dati.

**Overfitting:** l'algoritmo è troppo complesso e rappresenta anche le piccole particolarità del training set, ma non generalizza bene su dati nuovi.

## Matrice di confusione

Usata per valutare classificazioni.

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

## Metriche per classificazione

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 \times precision \times recall}{precision + recall}$$

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP}$$

$$specificity = \frac{TN}{TN + FP}$$

## Errore di regressione

**Root Mean Square Error (RMSE):**  $\frac{1}{n} \sqrt{\sum_{i=1}^n (y_i - \hat{y}_i)^2}$

**Mean Absolute Error (MAE):**  $\frac{\sum_{i=1}^n |y_i - \hat{y}_i|}{n}$

## Pratica

---

### Web GUI

<http://137.204.72.233:7180/cmf/home> (username = student, password = student)

## Comandi HDFS

**hdfs dfs –<comando>**

ls

rm

...

put <src> <dst> ⇒ copia un file da locale a cluster

get <src> <dst> ⇒ scarica un file

## MapReduce

Un progetto che verrà eseguito su Hadoop **deve** contenere le sue librerie.

<http://isi-vclust0.csr.unibo.it:50070/webhdfs/v1/bigdata/jars/hadoop-min-jars.zip?op=OPEN>

Una volta scritto, esportare un jar senza includere le librerie.

Per sottomettere un job:

**hadoop jar <jar-file> [Main-class] <inputDir> <outputDir> [params]**

inputDir ⇒ directory su HDFS contenente i file di input

outputDir ⇒ verrà creata dal job per salvare i risultati

## Spark UI

### Resource Manager

Mostra le applicazioni che stanno (o hanno) eseguendo nel cluster.

<http://isi-vclust0.csr.unibo.it:8088/cluster/apps>

### History Server

Mostra le applicazioni che sono state lanciate.

<http://isi-vclust0.csr.unibo.it:18089/>

## Spark Streaming

Creare uno *StreamingContext* a partire da uno *SparkContext*.

Vengono usati dei *DStream*, cioè delle sequenze di RDD, ognuno contenente i dati di una determinata finestra con lunghezza e periodo definiti in fase di creazione dello *StreamingContext*.