



MODELLI E TECNICHE PER BIG DATA

INTRODUZIONE

Il mondo di oggi genera una quantità di dati senza precedenti e la capacità di estrarre informazioni preziose da questi dati è fondamentale per il successo in molti campi, tra cui affari, scienza e governo.

Il modo migliore per sfruttare il valore dell'enorme quantità di dati a disposizione è implementare applicazioni di analisi e gestione dei dati scalabili per estrarre in modo efficiente modelli, pattern e tendenze utili da essi.

La programmazione di applicazioni big data è un compito impegnativo e diversificato che richiede una profonda comprensione di vari concetti tra cui analisi dei dati, elaborazione distribuita, elaborazione parallela e apprendimento automatico

OBIETTIVI

- Fornire indicazioni per gli ingegneri che cercano come sviluppare applicazioni big data robuste e scalabili.
- Offrire una comprensione approfondita dei principi e delle pratiche necessarie per implementare applicazioni di analisi big data efficienti.
- Utilizzare gli strumenti big data più diffusi in applicazioni del mondo reale, fornendo indicazioni su come scegliere gli strumenti giusti per ogni caso d'uso specifico.
- Trattare le ultime novità, come l'elaborazione Exascale e il machine learning parallelo e distribuito, in particolare, discutere di come possono essere sfruttate per analizzare ed elaborare grandi dataset

ARGOMENTI PRINCIPALI

1. I principali sistemi di storage distribuiti, essenziali per fronteggiare l'attuale crescita esponenziale dei dati da archiviare, garantendo scalabilità, efficienza, tolleranza ai guasti, disponibilità e consistenza
2. I principi fondamentali alla base dei processi della data analysis e della data science, nonché il loro sviluppo su sistemi di elaborazione scalabili.
3. I vantaggi di tecnologie come l'high-perfomance computing, il cloud computing e l'elaborazione distribuita, che sono utili nell'elaborazione di grandi quantità di dati in contesti reali.
4. I principali modelli di programmazione per i big data, che supportano gli utenti nell'espressione di algoritmi e applicazioni parallele, fornendo un'astrazione per un'architettura di computer parallela.
5. Le ultime proposte nell'area dell'elaborazione Exascale, che mirano a fornire soluzioni e strumenti scalabili in un'ampia gamma di campi scientifici, tra cui fisica, biologia e simulazione di fenomeni naturali
6. Gli strumenti di programmazione più utilizzati per l'elaborazione di big data, che gestiscono diversi tipi di dati (dai dati strutturati ai grafi, agli stream) e domini (applicazioni basate su batch, streaming, grafici e query).
7. Le caratteristiche principali dei diversi framework per supportare i programmatore nella scelta del framework più appropriato, insieme ad altri fattori importanti che possono guidare questa scelta, come il tipo di dati, la scala dell'infrastruttura, le competenze degli sviluppatori e le dimensioni della comunità.

CONCETTI FONDAMENTALI

DEFINIZIONI

- Definizione fornita da Gartner

"I big data sono asset informativi ad alto **volume**, alta **velocità** e/o alta **varietà** che richiedono forme di elaborazione delle informazioni innovative ed economiche che

consentano una migliore comprensione, un processo decisionale e un'automazione dei processi”

Questa definizione è basata sulle 3V **Volume, Velocità, Varietà**

- Definizione fornita da Gantz and Reinsel 2011

“Le tecnologie Big Data descrivono una nuova generazione di tecnologie e architetture, progettate per estrarre **valore** economico da **volumi** molto grandi di un'ampia **varietà** di dati, consentendo l'acquisizione, la scoperta e/o l'analisi ad alta **velocità**”

Questa definizione è basata sulle 4V **Volume, Velocità, Varietà, Valore**

- Definizione di Chang and Grady 2015

“I big data sono costituiti da set di dati estesi, principalmente nelle caratteristiche di **volume, varietà, velocità e/o variabilità**, che richiedono un'architettura scalabile per un'archiviazione, una manipolazione e un'analisi efficienti”

Questa definizione è basata sulle 4V **Volume, Velocità, Varietà, Variabilità**

Negli anni in molti hanno cercato di aggiungere altri aggettivi che iniziano con la V, tra questi troviamo

- **Viralità:** capacità dei dati di trasmettere un messaggio che può raggiungere un gran numero di persone
- **Visualizzazione:** caratteristica che permette di rappresentare graficamente i dati
- **Viscosità:** capacità delle informazioni estratte dai dati di colpire l'interesse delle persone.
- **Venue (Luogo):** si riferisce all'origine dei dati, che possono essere raccolti da più fonti distribuite ed eterogenee.

DATA SCIENCE

La data science è una disciplina che combina **informatica, matematica applicata** e tecniche di **analisi dei dati** per fornire approfondimenti basati su grandi quantità di dati. Migliora le scoperte basando le decisioni su informazioni estratte da grandi set di dati tramite l'uso di algoritmi per: Collezionare, pulire, trasformare e analizzare

Step principali per il processo della data science

1. Inquadrare il problema
2. Raccogliere i dati necessari
3. Elaborare i dati per l'analisi
4. Esplorare i dati
5. Eseguire un'analisi approfondita
6. Comunicare i risultati ottenuti

Data science **Skills**

- Conoscenza e studio del dominio applicativo
- Comunicazione con i proprietari/utenti dei dati
- Prestare attenzione alla qualità dei dati
- Sapere come i dati possono essere rappresentati
- Gestire la trasformazione e l'analisi dei dati
- Conoscere la visualizzazione e la presentazione dei dati
- Considerare le questioni etiche

BIG DATA STORAGE

SCALABILITA' VERTICALE

Aumentare le risorse (CPU, RAM, Disk, network I/O) di ogni singolo server, rendendolo più veloce e più potente

SCALABILITA' ORIZZONTALE

Aggiungere più nodi di memoria o nodi di calcolo al sistema e distribuire il carico di lavoro tra di essi

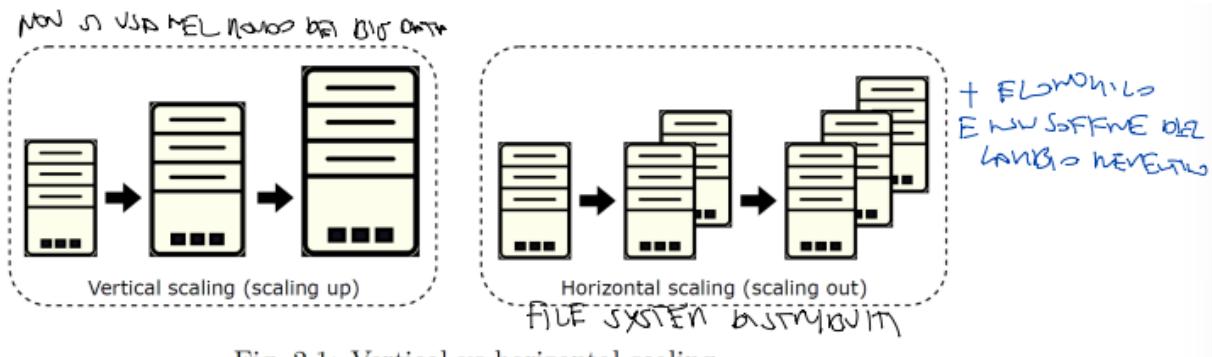


Fig. 2.1: Vertical vs horizontal scaling.

DATABASE NoSQL

Molti database relazionali sono poco flessibili alla scalabilità orizzontale su molti server. Per questo esistono quelli NoSQL che hanno come obiettivo quello di garantire la scalabilità orizzontale nelle operazioni di lettura e scrittura. I database NoSQL sfruttano i nuovi nodi in modo trasparente, senza richiedere la distribuzione manuale delle informazioni o una gestione aggiuntiva del database. Inoltre sono progettati per assicurare la distribuzione automatica dei dati e sono anche tolleranti ai guasti.

I database NoSQL consentono di archiviare valori scalari (es. numeri, stringhe), oggetti binari (es. immagini, video) o strutture più complesse (es. grafi).

CATEGORIE PRINCIPALI DI DB NoSQL

- **Key-Value:** i database Key-Value offrono meccanismi per archiviare dati come coppie <chiave, valore> distribuite su più server. Una **Distributed Hash Table (DHT)** può essere utilizzata per implementare una struttura di indicizzazione scalabile, in cui il recupero dei dati avviene utilizzando una chiave per trovare un valore. Utilizzati negli e-commerce garantiscono un'ottima velocità di recupero delle informazioni ma non supportano range-query. Alcuni esempi reali sono: **DynamoDB, Redis**

Key-value databases

Horizontal scaling	Very high scale provided via sharding.
When to use	Simple data schema or extreme speed scenario (e.g., in real time).
CAP tradeoff	Most solutions prefer <u>consistency over availability</u> . PREF JIA
Pros	Simple data model; very high scalability; data can be accessed using query language, such as SQL.
Cons	Some queries could be inefficient or limited due to sharding (e.g., join operations across shards); no API standardization; maintenance is difficult; unsuitable for complex data. NAME QUERY

- **Document-based:** I database basati su documenti sono progettati per gestire i dati in documenti formattati in maniera diversa (es. JSON), in cui ogni documento è assegnato a una chiave unica per identificarlo e recuperarlo. Estendono i Key-Value Stores, consentendo di archiviare, recuperare e gestire informazioni semi-strutturate anziché singoli valori. Generalmente supportano: indici secondari, tipologie multiple di documenti per database, meccanismi per interrogare collezioni basandosi su vincoli multipli di valore-attributo. Un esempio: **MongoDB**

Document-based databases

Horizontal scaling	Scale provided via replication and sharding.
When to use	When record structure is relatively small, and it is possible to store all of its related properties in a single document.
CAP tradeoff	In most cases, prefer consistency over availability.
Pros	High scalability and simple data model; generally support for secondary indexes, multiple types of documents per database; nested documents; MapReduce support for ad hoc querying.
Cons	Eventual consistency with limited atomicity and isolation; queries limited to keys and indexes; unsuitable for interconnected data.

- **Column-based:** i Column-based database (detti anche **Extensible Record Stores**) forniscono meccanismi per archiviare record estensibili che possono

essere partizionati su più server. I record sono detti **estensibili** perché è possibile aggiungere nuovi attributi a ogni singolo record. I sistemi di archiviazione a colonne offrono sia partizionamento orizzontale (archiviazione di record su nodi diversi) e sia partizionamento verticale (archiviazione di parti di un singolo record su diversi server). In alcuni sistemi, le colonne di una tabella possono essere distribuite su più server utilizzando **gruppi di colonne**, dove i gruppi predefiniti indicano quali colonne è meglio archiviare insieme.

Alcuni esempi: **Cassandra**, **HBase**, **BigTable**

Table 2.3. Summary considerations about column-based databases.

Column-based databases	
Horizontal scaling	Very high scale capabilities.
When to use	When consistency and high scalability are needed, without using indexed caching front end.
CAP tradeoff	Most solutions prefer consistency over availability.
Pros	Higher throughput and stronger concurrency when it is possible to partition data; multi-attribute queries; data are naturally indexed by columns; support for semi-structured data.
Cons	Greater complexity; unsuitable for interconnected data.

- **Graph-based:** I database basati su grafi sono sistemi ampiamente utilizzati per archiviare e interrogare informazioni rappresentabili sotto forma di grafi, anziché tabelle o documenti. Un grafo è rappresentato come un insieme di nodi, archi e proprietà, che sono difficili da gestire utilizzando un database relazionale. I database basati su grafi consentono di eseguire efficientemente un'ampia gamma di query sui grafi, senza la necessità di costose operazioni di join tra tabelle. Permettono di accelerare l'esecuzione di algoritmi sui grafi, come l'identificazione di comunità, gradi, centralità, distanze, percorsi e altre relazioni tra nodi. Un esempio: **Neo4j**

Table 2.5. Summary considerations about graph-based databases.

Graph-based databases	
Horizontal scaling	Poor horizontal scaling.
When to use	For storing and querying entities linked together by relationships; use cases are social networking and recommendation engines.
CAP trade-off	Usually prefer availability over consistency.
Pros	Powerful data modeling and relationship representation; locally indexed connected data; easy and efficient to query.
Cons	Unsuitable for non-graph data.

DATABASE NoSQL REALI

MongoDB

Database di tipo document-based. Progettato per supportare applicazioni internet e web-based.

Rappresenta i documenti in un formato simile a JSON chiamato **BSON**, che funge da formato di trasferimento per i documenti di MongoDB.

Comprende la struttura interna degli oggetti BSON, consentendo di accedere a chiavi anche nidificate utilizzando la **dot notation**. Questa funzionalità consente a MongoDB di costruire indici e confrontare oggetti espressioni di query, che coprono sia le chiavi BSON di livello superiore che quelle nidificate. Supporta query complesse e indici completi. E' progettato per partizionare i dati su più nodi in maniera automatica, supporta la ridondanza ed è in grado di gestire lo storage in maniera automatica

Google Bigtable

Database di tipo column-based. Costruito sopra il **Google File System (GFS)**, è in grado di archiviare fino a petabyte di dati. E' di tipo master-center. I dati sono archiviati in tabelle multidimensionali sparse, distribuite e persistenti composte da righe e colonne. Ogni riga è indicizzata da una chiave di riga unica; le colonne correlate sono raggruppate in **famiglie di colonne**. Una generica colonna è identificata da una famiglia e un qualificatore che la identifica in maniera univoca all'interno della famiglia. I dati sono ordinati per chiave di riga. Le righe sono dinamicamente partizionate in blocchi contigui chiamati **tablet**. I tablet sono distribuiti tra diversi nodi di un cluster Bigtable (**tablet servers**).

HBase

Database di tipo column-based. Si propone come alternativa open-source al database di google. Similmente ad esso, utilizza **Hadoop** e il **Hadoop Distributed File System (HDFS)**. Sistemi scalabili linearmente con tabelle composte da righe e colonne. Tabelle senza schema (solo le famiglie di colonne vengono definite al momento della creazione della tabella). Ogni tabella richiede una **chiave primaria** obbligatoria per l'accesso. HBase si integra con **Hive**, utilizzato come motore di query per l'elaborazione batch di Big Data garantendo anche la tolleranza ai guasti.

Redis

Database di tipo key-value. Redis è un popolare data store open-source in-memory, utilizzato come database, cache, message broker e motore di streaming. Supporta operazioni atomiche come: aggiunta di una stringa, incremento di un valore in un hash, inserimento di un elemento in una lista, calcolo di intersezioni, unioni e differenze tra insiemi ed estrazione dell'elemento con il punteggio massimo da un insieme ordinato. Anche se lavora principalmente con set di dati in memoria per migliorare le prestazioni, può persistere i dati salvandoli periodicamente su disco.

DynamoDB

Database di tipo key-value. Servizio di database NoSQL completamente gestito offerto da Amazon Web Services (AWS). Ideale per applicazioni che richiedono accesso a bassa latenza, come giochi, applicazioni mobili e piattaforme di e-commerce. Tra le funzionalità built-in ci sono sicurezza integrata con crittografia in transito e a riposo, controllo degli accessi granulare e integrazione con AWS IAM, Backup e ripristino completamente gestiti, sincronizzazione multi-regione e multi-master per alta disponibilità e durabilità dei dati. Si integra perfettamente con altri servizi AWS (es. Amazon S3) per costruire architetture serverless potenti.

Cassandra

Database di tipo column-based. Architettura **masterless** ad anello, in cui tutti i nodi hanno ruoli identici, consentendo a qualsiasi utente autorizzato di connettersi a qualsiasi nodo in qualsiasi data center.

Architettura semplice e flessibile che permette di aggiungere nodi senza interruzioni del servizio. Tra le caratteristiche principali troviamo la distribuzione automatica dei dati sui nodi senza intervento manuale, nessun punto singolo di guasto, garantendo continua disponibilità dei dati, servizio di replica personalizzabile per replicare i dati tra nodi organizzati in un anello ed infine in caso di guasto di un nodo, una o più copie dei dati sono disponibili su altri nodi.

Neo4j

Database di tipo graph-based. Ogni nodo contiene un elenco di record di relazioni che fanno riferimento ad altri nodi e attributi aggiuntivi (es. timestamp, metadati, coppie chiave-valore). Ogni relazione deve avere: nome, direzione, nodo iniziale e finale e proprietà opzionali. I nodi e le relazioni possono avere **etichette** per rappresentare i ruoli di un nodo (es. utente, indirizzo, azienda). I cluster Neo4j sono progettati per alta disponibilità e scalabilità orizzontale con **replica master-slave**.

CONFRONTO DATABASE NoSQL

Table 2.1. Comparison of some NoSQL databases.

	DynamoDB	Cassandra	Hbase	Redis	BigTable	MongoDB	Neo4j
Type	KV	Col	Col	KV	Col	Doc	Graph
Data storage	MEM FS	HDFS CFS	HDFS	MEM FS	GFS	MEM FS	MEM FS
MapReduce	Yes	Yes	Yes	No	Yes	Yes	No
Persistence	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Replication	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Scalability	High	High	High	High	High	High	High
Performance	High	High	High	High	High	High	High, variable
High availability	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Language	Java	Java	Java	Aansi-C	Java Python Go Ruby	C++	Java
License	Proprietary	Apache2	Apache2	BSD	Proprietary	AGPL3	GPL3

Note: FS: file system; MEM: in-memory; KV: key-value; Doc: document; Col: column.

CAP THEOREM

Teorema di Gilbert and Lynch (2002) afferma che un sistema distribuito non può garantire simultaneamente le tre proprietà seguenti:

- **Consistenza (C)** : tutti i nomi vedono gli stessi dati nello stesso tempo
- **Disponibilità (A)**: ogni richiesta riceve una risposta in tempi ragionevoli
- **Partizione (P)** : il sistema continua a funzionare anche se una parte di esso è guasta

Molti database NoSQL preferiscono la consistenza rispetto alla disponibilità altri invece viceversa

DATA ANALYSIS vs DATA ANALYTICS

Le **data analysis** applicazioni che esplorano, interrogano, analizzano, visualizzano e in generale processano set di dati su larga scala. Si riferisce al "processo" di preparazione ed analisi dei estrarre informazioni utili.

La **data analytics** è la scienza che si occupa di raccogliere ed esaminare dati grezzi con l'obiettivo di trarne conclusioni significative. Include strumenti e tecniche per ottenere tale scopo come per esempio strumenti di data visualization

BIG DATA ANALYTICS

Big data analytics si riferisce a tecniche avanzate di analisi dei dati applicate a set di Big Data. Alcuni esempi sono il data mining, l'intelligenza artificiale, il natura language processing.

Alcuni campi applicativi dell'analisi dei Big Data sono: l'analisi dei testi, tecniche per effettuare previsioni, analisi di reti (grafo), analisi prescrittiva

I processi dei big data analytics sono intensivi dal punto di vista computazionale, collaborativi e distribuiti per natura. I sistemi HPC (High Performance Computing) e il cloud offrono strumenti e ambienti per supportare strategie di esecuzione parallela per l'analisi, l'inferenza e la scoperta nei dati distribuiti.

La creazione di framework basati sul calcolo parallelo e sulle tecnologie cloud è una condizione abilitante per lo sviluppo di compiti di analisi ad alte prestazioni e tecniche di machine learning.

PARALLEL COMPUTING

Il calcolo parallelo può essere definito come la pratica di affrontare un problema di dimensione n suddividendolo in $k \geq 2$ parti, risolte sfruttando p processori fisici contemporaneamente.

Questo paradigma di risoluzione dei problemi, noto anche come **divide et impera**, è applicabile solo se il problema è parallelizzabile, ovvero può essere scomposto in k sotto-problemi distinti.

Concorrenza: due o più task possono essere in progresso simultaneo

Parallelismo: due o più task sono eseguiti contemporaneamente

Essere "in progresso" non implica necessariamente "essere in esecuzione"; il parallelismo implica concorrenza, ma non il contrario.

ESEMPIO DIVIDE ET IMPERA: PRODOTTO SCALARE

Il prodotto scalare tra due vettori

$$\mathbf{a} = [a_1 \quad a_2 \quad \dots \quad a_n], \quad \mathbf{b} = [b_1 \quad b_2 \quad \dots \quad b_n]$$

è definito come

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i \cdot b_i$$

il prodotto scalare può essere parallelizzato andando a scomporre il problema in K somme parziali (*divide step*) che vengono calcolate in k processori distinti simultaneamente (*impera step*). Il risultato finale può essere ottenuto combinando le somme parziali, dopo una fase di sincronizzazione.

Formalmente

$$\mathbf{a} \cdot \mathbf{b} = \sum_{k=1}^p \left(\sum_{i \in D_k} a_i \cdot b_i \right)$$

NATURA PARALLELA DEI PROBLEMI

Un problema con dominio D può essere di due tipi differenti

- **Data-parallel** : D è un insieme di dati e la soluzione del problema può essere espressa come applicazione della funzione f su ogni sottoinsieme di D

$$f(D) = f(d_1) + f(d_2) + \dots + f(d_k)$$

- **Task-parallel**: F è un insieme di funzioni e la soluzione al problema può essere espressa come l'applicazione di ogni funzione f dell'insieme F al dominio D

$$F(D) = f_1(D) + f_2(D) + \dots + f_k(D)$$

ARCHITETTURE PARALLELE

La tassonomia di Flynn (Flynn e Rudd, 1996) classifica i diversi modelli di sistemi paralleli in base alla molteplicità dei flussi di istruzioni e dati che possono gestire:

- **SISD (Single Instruction Stream, Single Data Stream)**: un flusso di istruzioni che opera su un singolo flusso di dati (sistemi sequenziali).
- **SIMD (Single Instruction Stream, Multiple Data Stream)**: un flusso di istruzioni che opera contemporaneamente su flussi di dati multipli.
- **MISD (Multiple Instruction Stream, Single Data Stream)**: più flussi di istruzioni che operano su un singolo flusso di dati (raramente utilizzati).
- **MIMD (Multiple Instruction Stream, Multiple Data Stream)**: più flussi di istruzioni che operano su flussi di dati multipli (comune nei sistemi paralleli moderni).

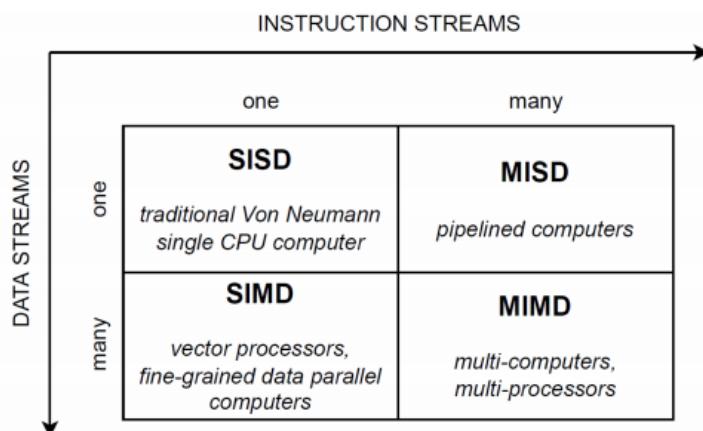


Fig. 2.2: Flynn's taxonomy.

METRICHE DI VALUTAZIONE DELLE PERFORMANCE

Dati

T_s = tempo esecuzione sequenziale (singolo processore)

T_n = tempo di esecuzione parallela (n processori)

SPEED-UP

Rapporto tra il tempo di esecuzione sequenziale di un programma e tempo di esecuzione della versione parallela dello stesso programma

$$S_n = \frac{T_s}{T_n}$$

EFFICIENZA

Misura l'effettivo uso di ogni processore di un computer parallelo quando esegue un programma

$$E_n = \frac{S_n}{n} = \frac{T_s}{(n \cdot T_n)}$$

LEGGE DI AMDAHL

Dati

n = numero di processori

F = frazione parallelizzabile ($0 < F < 1$)

Secondo la legge di Amdahl, lo speedup teorico è:

$$\hat{S}_n = \frac{1}{(1 - F) + \frac{F}{n}}$$

Casi limiti

$F = 0$ nessuna parte del programma è parallelizzabile $\rightarrow \hat{S}_n = 1$

$F = 1$ l'intero programma è parallelizzabile $\rightarrow \hat{S}_n = n$

Il massimo speed-up ottenibile per il programma che ha una frazione parallelizzabile F è ottenuto dalla seguente formula

$$S_{max} = \lim_{n \rightarrow \infty} \hat{S}_n = \frac{1}{(1 - F)}$$

l'interpretazione della formula ci permette di affermare che se si aumenta il livello di parallelismo, lo speed-up teorico è limitato dall'inverso della frazione non parallelizzabile del programma

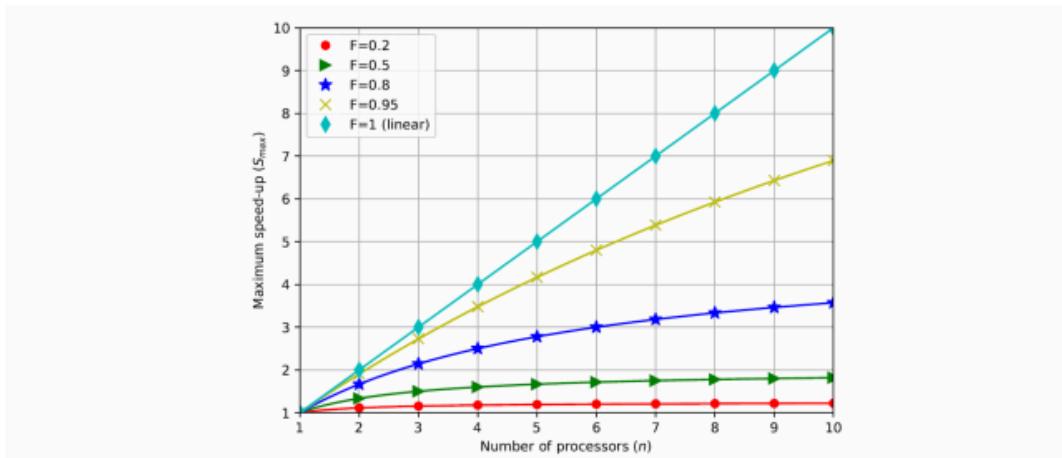


Fig. 2.3: Maximum speed-up obtainable with a varying number of available processors (n), given the parallelizable fraction of the program (F).

CLOUD COMPUTING

La definizione di Mell (2011) per il cloud computing è la seguente:

"Un modello per consentire un comodo accesso di rete su richiesta a un pool condiviso di risorse informatiche configurabili (ad esempio reti, server, storage, applicazioni e servizi) che possono essere rapidamente fornite e rilasciate con un minimo sforzo di gestione o interazione con il fornitore di servizi"

Cinque caratteristiche essenziali:

- servizio on-demand
- accesso alla rete condiviso
- risorse comuni (comdivise dagli utenti)

- elasticità => adattamento al carico
- adattamento rapido
- servizio su misura

MODELLO DI SERVIZIO CLOUD

Più ad alto livello

- **Software as a Service (SaaS)** : software e dati sono ottenuti tramite servizi internet come risorse pronte all'uso
- **Platform as a Service (PaaS)** : ambiente che include database, applicazioni server, ambienti di sviluppo per creare e testare applicazioni custom
- **Infrastructure as a Service (IaaS)** : un modello di outsourcing in base al quale i clienti noleggiano risorse come CPU, dischi o risorse più complesse come server virtualizzati o sistemi operativi per supportare le loro operazioni.

*Livello intermedio
usato da
imp. inform.
L'app viene
testata su
un cloud
e si possono
utilizzare i servizi
presenti
sul cloud*

MODELLO DI DISTRIBUZIONE CLOUD

- **Public cloud** : fornisce servizi al pubblico in generale tramite Internet e gli utenti hanno poco o nessun controllo sull'infrastruttura tecnologica sottostante. I venditori gestiscono i loro data center proprietari, fornendo servizi costruiti su di essi
- **Private cloud** : fornisce servizi distribuiti su un'intranet aziendale o in un data center privato. Spesso, le piccole e medie aziende IT preferiscono questo modello di distribuzione in quanto offre soluzioni avanzate di sicurezza e controllo dei dati che non sono disponibili nel modello di cloud pubblico
- **Hybrid cloud** : è la composizione di due o più cloud (privati o pubblici) che rimangono entità diverse ma sono collegate tra loro.

*Vengono
utilizzati
da aziende*

SERVIZI CLOUD PER I BIG DATA

I servizi cloud per i big data sono forniti dalle più popolari piattaforme cloud come:

- Amazon Web Services
- Google Cloud Platform
- Microsoft Azure
- OpenStack

VERSO I CALCOLATORI EXASCALE

- **High-performance computing (HPC)** : l'uso dell'elaborazione parallela dei dati per gestire grandi quantità di dati e calcoli complessi.
- **High-performance data analytics (HPDA)** : l'applicazione delle soluzioni HPC alla data analytics
- **Exascale** : è la nuova frontiera dell'HPC. Si riferisce ai sistemi di elaborazione capaci di almeno un exaFLOP, il che significa che sono in grado di eseguire almeno 10^{18} operazioni in virgola mobile al secondo (FLOPS)

Un sistema exascale può essere descritto da diversi attributi:

- **Attributi fisici**: sono correlati al consumo energetico totale e alle dimensioni del sistema (ad esempio, area e volume).
- **Velocità di calcolo**: la velocità con cui un certo tipo di operazioni può essere eseguito al secondo (misurata in FLOPS, istruzioni al secondo e accessi alla memoria al secondo). M° gli ~~operazione al secondo eseguite~~
- **Capacità di archiviazione**: misura quanta memoria è disponibile in varie parti della gerarchia di archiviazione (memoria principale, scratch e archiviazione persistente).
- **Velocità di larghezza di banda**: la velocità con cui i dati rilevanti per il calcolo possono essere spostati nel sistema (le metriche includono la larghezza di banda della memoria locale, la larghezza di banda del checkpoint, la larghezza di banda I/O e la larghezza di banda on-chip).

Amazon Web Services

- *Data transfer*: data transport solutions designed to securely transfer huge amounts of data into the AWS cloud.
- *Data management*: a variety of database systems, including Amazon Relational Database Service (RDS) for relational tables, and NoSQL solutions like Amazon DynamoDB.
- *Compute*: Elastic Compute Cloud (EC2), for creating and running virtual servers, and Amazon Elastic MapReduce for building and executing MapReduce applications.
- *Storage*: several flexible storage options for permanent and transient data storage.

Google Cloud Platform

- *Compute*: IaaS solutions such as Google Compute Engine and several PaaS such as Google App Engine, for developing and hosting web applications in Google-managed data centers.
- *Storage*: Google Cloud Storage and Datastore, SQL-like solutions such as cloud SQL and NoSQL services such as Bigtable.
- *Networking*: services for enabling communication and load balancing across resources, including Google Cloud DNS, Content Delivery Network (CDN) and security services such as Armor.

Microsoft Azure

- *Compute*: the computational environment to execute cloud applications; each application is structured into roles: Web role, for Web-based applications; Worker role, for batch applications; Virtual Machines role, for virtual-machine images.
- *Storage*: scalable storage to manage binary and text data (Blobs), non-relational tables (Tables), and queues for asynchronous communication between components (Queues).
- *Fabric controller*: aimed at building a network of interconnected nodes from the physical machines of a single data center; the Compute and Storage services are built on top of this component.

OpenStack

- *Compute*: provides virtual servers upon demand by managing the pool of processing resources available in the datacenter; it supports different virtualization technologies such as VMware and KVM.
- *Storage*: provides a scalable and redundant storage system; it supports Object Storage and Block Storage, that allow storing and retrieving objects and files in the datacenter.
- *Networking*: responsible for managing the networks and IP addresses within the OpenStack environment.
- *Shared Services*: additional services provided to ease the use of the datacenter, such as Identity Service for mapping users and services, Image Service for managing server images, and Database Service for relational databases.

Main challenges of exascale systems

- Energy
- Concurrency
- Data locality
 - Node locality → località migliore, gestisco tutto su un simile modo
 - Intra-rack locality
 - Inter-rack locality → caso peggiore, sono costretto a spostare i dati tra diversi computer
- Memory
- Resilience

Table 2.6. The top three positions of the TOP500 list in June 2023.

Rank	#1	#2	#3
Rmax/Rpeak (PetaFLOPS)	1,194.00/1,679.81	442.01/537.21	309.10/428.70
Name	Frontier	Fugaku	LUMI
Model	HPE Cray EX235a	Supercomputer Fugaku	HPE Cray EX235a
CPU cores	591,872 (9,248 × 64-core Optimized 3rd Generation EPYC 64C @2.0 GHz)	7,630,848 (158,976 × 48-core Fujitsu A64FX @2.2 GHz)	75,264 (1,176 × 64-core Optimized 3rd Generation EPYC 64C @2.0 GHz)
Accelerator cores	36,992 × 220 AMD Instinct MI250X	—	4,704 × 220 AMD Instinct MI250X
Interconnect	Slingshot-11	Tofu interconnect	Slingshot-11
Manufacturer	HPE	Fujitsu	HPE
Site, country	Oak Ridge National Laboratory, United States	RIKEN Center for Computational Science, Japan	EuroHPC JU European Union, Finland
Year	2022	2020	2022
Operating system	Linux (HPE Cray OS)	Linux (RHEL)	Linux (HPE Cray OS)

Tra le sfide più importanti da affrontare in ambito exascale ci sono i problemi di energia, la concorrenza, la località dei dati e la resilienza

Prima di questa parte vedere 02B Introduzione al data mining

MACHINE LEARNING DISTRIBUITO E PARALLELO

Spesso le tecniche di machine learning utilizzano approcci di esecuzione centralizzata su un computer o in un data center sia per l'addestramento che per l'esecuzione del modello. Questi approcci non sono appropriati quando i dati sono molto grandi o sono ubicati in molti dispositivi di archiviazione diversi. Inoltre, quando devono essere analizzate fonti di big data, i tempi di esecuzione sequenziale possono essere molto lunghi, impiegando giorni o settimane per essere completati. L'approccio più efficace per ridurre i tempi di esecuzione è utilizzare modelli e infrastrutture di elaborazione **parallela e distribuita**.

Si possono identificare tre strategie principali per lo sfruttamento del parallelismo negli algoritmi di machine learning:

- **Parallelismo indipendente:** ogni processo accede all'intero set di dati o alla propria partizione e non comunica o si sincronizza con altri processi durante le operazioni di addestramento e apprendimento.

- **Parallelismo Single Program Multiple Data (SPMD)**: un set di processi che eseguono lo stesso algoritmo eseguito in parallelo su diverse partizioni di un set di dati; i processi SPMD cooperano scambiando risultati parziali durante la loro esecuzione.
- **Parallelismo dei task**: ogni processo può eseguire algoritmi diversi su (una diversa partizione del) set di dati; i processi possono comunicare in base a diversi modi richiesti dall'algoritmo parallelo

Nella maggior parte degli algoritmi distribuiti, lo stesso codice viene eseguito su ogni nodo contemporaneamente e viene calcolato un modello locale al nodo. Quindi, tutti i modelli locali vengono aggregati/combinati in un sito centrale o sono condivisi in tutti i nodi per produrre il modello globale. Questo schema è comune a diversi algoritmi di machine learning distribuiti

META LEARNING

Il meta-learning mira a implementare un modello globale che analizza un set di dataset distribuiti.

I set di addestramento iniziali vengono forniti in input a N algoritmi di apprendimento che vengono eseguiti su nodi diversi per creare N modelli di classificazione (classificatori di base). Un set di addestramento di meta-livello viene creato combinando le previsioni dei classificatori di base su un set di convalida comune. Infine, un classificatore globale viene addestrato dal set di addestramento di meta-livello tramite un algoritmo di meta-apprendimento.

ENSEMBLE LEARNING

L'ensemble learning mira a migliorare l'accuratezza del modello aggregando le previsioni prodotte da un insieme di algoritmi (learners).

Due strategie principali:

- Il **bagging**, chiamato anche voto per la classificazione e media per la regressione, combina le classificazioni previste da un insieme di modelli o dallo stesso tipo di modello per diversi set di dati di apprendimento.
- Il **boosting** unisce le decisioni di diversi modelli, come il bagging, ma utilizza la ponderazione per dare più influenza ai modelli di maggior successo (mentre nel bagging i modelli ricevono lo stesso peso).

Il risultato è un classificatore d'insieme che mostra una maggiore accuratezza di classificazione rispetto a ciascun classificatore di base utilizzato per comporlo.

FEDERATED LEARNING

Il federated learning è pensato per analizzare dati grezzi distribuiti senza essere spostati su un singolo server o data center. Questa strategia seleziona un set di nodi e invia la prima versione contenente i parametri del modello di un modello di apprendimento automatico a tutti i nodi. Quindi ogni nodo esegue il modello, lo addestra solo su dati locali e mantiene una versione locale del modello. Consente ai dispositivi mobili di apprendere in modo collaborativo un modello di apprendimento condiviso mantenendo tutti i dati di addestramento a bordo, migliorando così la sicurezza e la privacy.

COLLECTIVE DATA MINING *Vedere se può essere spiegato meglio*

Il data mining collettivo costruisce il modello globale attraverso la combinazione di modelli parziali elaborati nei diversi siti, a differenza di altre tecniche che combinano un set di modelli completi generati in ogni sito. La classificazione globale si basa sul fatto che qualsiasi funzione può essere espressa in modo distribuito utilizzando un set di funzioni di base appropriate che possono contenere termini non lineari. Se le funzioni di base sono ortonormali, un'analisi locale genera risultati che possono essere utilizzati efficacemente come componenti del modello globale. Se un termine non lineare è presente nella funzione di sommatoria, il modello globale non è completamente scomponibile tra siti locali e devono essere presi in considerazione i termini incrociati che coinvolgono caratteristiche da nodi diversi

MODELLI : MAP REDUCE

I **modelli di programmazione parallela** sono spesso la caratteristica principale dei framework dei big data poiché influenzano il paradigma di esecuzione dei motori di elaborazione e il modo in cui gli utenti progettano e creano applicazioni. Consentono la separazione dei problemi di sviluppo software da quelli di esecuzione parallela, fornendo **astrazione e stabilità**.

- L'astrazione è garantita perché le operazioni del modello sono a un livello superiore rispetto a quelle delle architetture sottostanti.
- Semplifica la struttura del software e la difficoltà del suo sviluppo, garantendo anche la stabilità tramite un'interfaccia standard.

Pertanto, il modello può ridurre lo sforzo di implementazione, prendendo decisioni una volta per ogni sistema di destinazione, anziché per ogni

I modelli di programmazione vengono distinti in base al livello di **astrazione**, consentendo l'espressione di meccanismi di programmazione di alto e basso livello.

- **I modelli scalabili di alto livello** consentono ai programmati di specificare la logica dell'applicazione nascondendo i dettagli di basso livello, affidandosi ai compilatori per l'ottimizzazione.
- **I modelli scalabili di basso livello** consentono l'interazione diretta con le unità di elaborazione e di archiviazione, consentendo una specifica precisa del parallelismo delle applicazioni.

I sistemi di programmazione sono implementazioni di uno o più modelli e possono essere sviluppati attraverso diverse strategie:

- **Sviluppo del linguaggio:** implica la creazione di nuovi linguaggi di programmazione paralleli o l'integrazione di costrutti paralleli e strutture dati in linguaggi esistenti.
- **Approccio di annotazione:** utilizza simboli o parole chiave specifici nelle annotazioni per identificare istruzioni parallele nel codice del programma e indicare al compilatore quali istruzioni devono essere eseguite contemporaneamente.
- **Integrazione della libreria:** questo approccio implica il miglioramento del parallelismo includendo librerie nel codice dell'applicazione, che è l'approccio più popolare poiché è ortogonale ai linguaggi host.

I modelli di programmazione, come MapReduce e Message passing, forniscono astrazione per la programmazione parallela. I sistemi di programmazione come Apache Hadoop e MPI supportano questi modelli, soddisfacendo un'ampia gamma di applicazioni big data e livelli di competenza degli utenti. Data l'ampia gamma di applicazioni big data e classi di utenti, sono

stati proposti diversi modelli di programmazione parallela, che abbracciano vari livelli di astrazione (alto e basso)

MAP REDUCE

Il modello di programmazione **MapReduce** è stato sviluppato da Google nel 2004 (rivoluzionando internet con l'algoritmo page-rank) per affrontare la sfida di elaborare efficacemente i big data. Il suo paradigma è stato ispirato dalle funzioni **map** e **reduce** disponibili nei linguaggi di programmazione funzionale, come LISP, e consente ai progettisti di creare applicazioni distribuite basate su queste due operazioni

Il modello MapReduce sfrutta ampiamente la strategia di *dividi et impera* per affrontare i problemi legati ai big data

1. Dividi il problema in piccoli sotto-problemi (più semplici)
2. esegui in maniera indipendente i sotto-problemi in parallelo usando i diversi nodi workers
3. combina i risultati intermedi ottenuti da ogni singolo nodo worker

Il programmatore definisce due fasi per il processo di MapReduce: *map* e *reduce*

- La funzione **map** riceve in input una coppia (key,value) e produce in output una lista intermedia di coppie (key, value):

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

- La funzione **reduce** unisce tutti i valori intermedi con la stessa chiave (key)

$$reduce(k_2, list(v_2)) \rightarrow list(v_3)$$

Il parallelismo si ottiene in entrambe le fasi:

- Nella fase di **map**, in cui le chiavi possono essere elaborate contemporaneamente da computer diversi (le chiamate di map vengono distribuite tra i computer mediante sharding dei dati di input).
- Nella fase di **reduce**, in cui i reducer che lavorano su chiavi distinte possono essere eseguiti

contemporaneamente. Di solito i reducer svolgono un'attività meno onerosa rispetto ai mapper

Di conseguenza, gli algoritmi MapReduce scalano da un singolo server a centinaia di migliaia di server. L'approccio MapReduce nasconde i dettagli della parallelizzazione sottostante al programmatore, rendendolo semplice da usare. Gli sviluppatori possono concentrarsi sulla definizione dei calcoli, senza addentrarsi nei dettagli di come vengono eseguiti o di come i dati vengono inviati ai processori.

ESEMPIO: INVERTED INDEX

Un esempio di un'applicazione MapReduce consiste nella generazione dell'inverted index.

Dato un insieme di documenti, l'indice contiene un insieme di parole, specificando l'ID di tutti i documenti che contengono quella parola

- la funzione map genera una sequenza di coppie <parola, documentID> per ogni documento
- la funzione reduce prende in input tutte le coppie per una data parola, riordina gli ID dei documenti corrispondenti, e restituisce in output coppie <parola, list(documentID)>

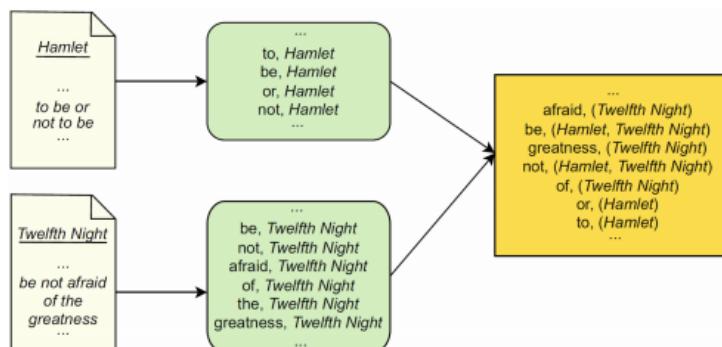


Fig. 3.1. Inverted index using MapReduce.

FASI DI MAP REDUCE

Un **job** è un programma MapReduce che consiste nel codice per la fase di map e nel codice per la fase di reduce. Comprende inoltre impostazioni di setup (per

esempio dove salvare i risultati), e anche il dataset in input che viene salvato in un file system distribuito.

Ogni job MapReduce è diviso in più piccole unità note come **task**. I task che eseguono la funzione map si chiamano **mapper**, mentre quelli che eseguono la funzione reduce prendono il nome **reducer**.

Per definire applicazioni complesse che non possono essere scritte come un singolo job MapReduce, gli utenti potrebbero dover comporre workflows MapReduce, che implicano più cicli di operazioni di mappatura e riduzione.

I sistemi moderni che implementano questo modello di programmazione come Apache Hadoop, seguono il modello **master-worker**.

Un nodo **user** invia un job al nodo master che identifica nel sistema un nodo **worker** a riposo e quindi assegna ad ogni worker un mapper o un reducer. Il nodo master coordina l'intero flusso, andando a organizzare entrambi i tipi di task. Completati tutti i task il nodo master preleva il risultato e lo restituisce al nodo user.

L'elaborazione in un'applicazione MapReduce può essere descritta come segue:

1. Un **descrittore di job** viene inviato a un processo master, descrivendo l'attività MapReduce da svolgere e altre informazioni, come la posizione dei dati di input.
2. Il **master** avvia diversi processi di **mapper** e **reducer** su diverse macchine in base al descrittore. Inoltre distribuisce i dati di input, suddivisi in più chunk, a diversi mapper.
3. Ogni **mapper** utilizza la funzione map (definita nel descrittore di lavoro) per creare una lista di coppie **intermedie (chiave, valore)** dopo aver ricevuto il suo blocco di dati.
4. Lo stesso vale per i **reducer** a cui vengono assegnato a tutte le coppie con le stesse chiavi, facendo in modo che ogni reducer lavori su un numero di chiavi simile. Ogni reducer esegue la funzione di riduzione (specificata dal descrittore di job), che unisce tutti i dati con la stessa chiave per produrre un set di valori più piccolo.
5. Gli output di ogni reducer vengono quindi raccolti e inviati alla posizione specificata dal descrittore di job, formando i dati di output finali.

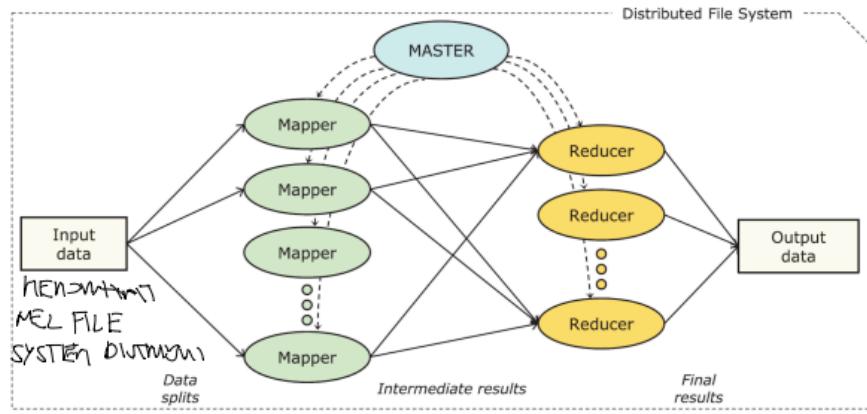


Fig. 3.2. MapReduce execution flow.

FASE DI COMBINAZIONE

Per aumentare la velocità, è possibile eseguire una fase di **combinazione**, che prevede una fase di mini-riduzione sull'output della mapper locale, che aggrega i dati prima di trasmetterli ai reducer tramite la rete. Un **combiner** viene utilizzato per aggregare l'output della mappa locale:

$$\text{combine}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3)$$

In molti casi, la stessa funzione può essere utilizzata sia per la combinazione che per la riduzione finale, con il vantaggio di ridurre la quantità di dati intermedi e il traffico di rete.

FASE SHUFFLE E SORT

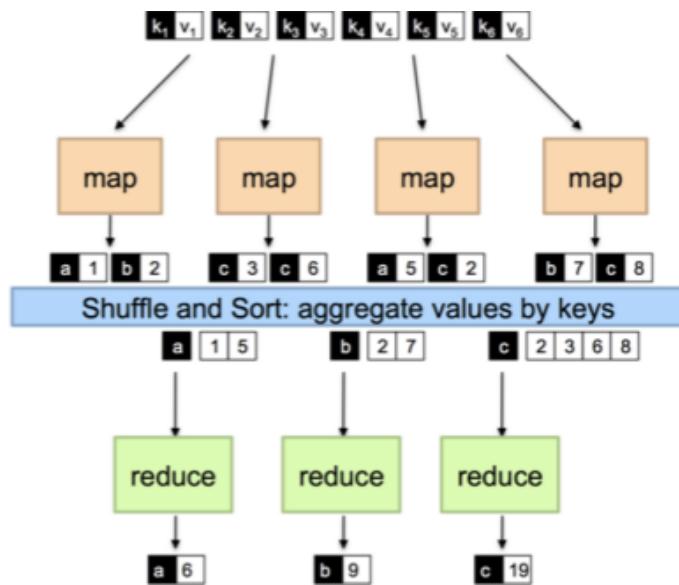
Tra le fasi di map (con combine) e reduce, avviene un'operazione di raggruppamento implicita distribuita, denominata **shuffle and sort**. Questa operazione trasferisce l'output del mapper ai reducer, unendo e ordinando i dati in base alla chiave prima di raggiungere ogni reducer.

Le chiavi intermedie, non archiviate nel file system distribuito, vengono distribuite sul disco locale di ogni computer nel cluster. Dopo che un mapper completa i suoi file di output ordinati, lo scheduler MapReduce avvisa i reducer, chiedendo loro di recuperare le coppie ordinate (chiave, valore) per le loro partizioni dai rispettivi mapper.

ESEMPI PRATICI DI PROBLEMI MAP REDUCE

In breve rivediamo il funzionamento dell'algoritmo MapReduce.

- I Mapper prendono in input tutte le coppie chiave-valore, per generare un numero arbitrario di coppie intermedie
- I Reducer invece vengono applicati a tutti i valori intermedi associati ad una stessa chiave provvisoria
- Tra la fase di map e la fase di reduce è presente una barriera che include una serie di operazione di ordinamento e raggruppamento



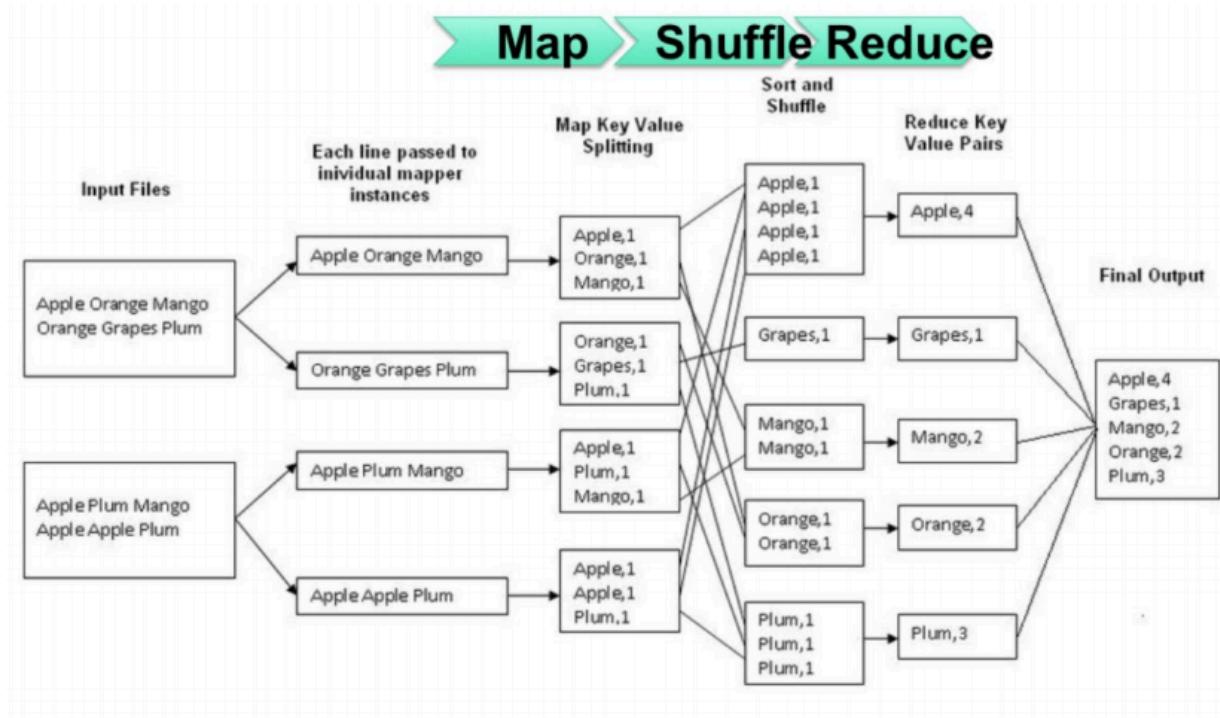
molto spesso le chiavi in input non hanno molta rilevanza nella fase iniziale

WORD COUNT

Con il word count si vuole contare il numero di occorrenze di ogni parola in una grande collezione di documenti.

Le fasi del word count sono le seguenti:

- Input:** repository di documenti, dove ogni documento è considerato un elemento
- Map:** legge il documento e restituisce una lista di coppie chiave-valore dove le chiavi sono le parole e il valore è sempre pari ad 1 $(w_1, 1)(w_2, 1), \dots, (w_n, 1)$
- Shuffle and Sort:** raggruppa le chiavi uguali ed emette coppie nella forma seguente $(w_1, [1, 1, \dots, 1]), \dots, (w_n, [1, 1, \dots, 1])$
- Reduce:** somma tutti i valori 1 della lista e restituisci le coppie $(w_1, s_1), \dots, (w_n, s_n)$
- Output:** coppie del tipo (w, m) dove w è una parola che appare almeno una volta in tutti i documenti in input ed m è il numero totale di occorrenze di w tra tutti i documenti



PSEUDO-CODICE DELLE FUNZIONI MAP E REDUCE

La funzione map crea per ogni parola presente nel documento una coppia con key = word e value = 1

```

Map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
        SI LAVORI SEMPRE CON  
STRINGS

```

La funzione reduce somma tutte le occorrenze di 1 per una data parola. La funzione reduce non riceve in input direttamente la lista perché si lascia la libertà a miglioramenti futuri, come l'aggiunta di una combine. Nel caso in cui l'operazione complessa gode della proprietà associativa allora il codice del metodo combine è sempre uguale al codice del metodo reduce

```

Reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result=0
    for each v in values:
        result += ParseInt(v)
    Emit(AsString(result))
    VOLUME FINALE

```

] Non avendo k
 Lista vuota
 Stampa 1
 Ma come
 In LISTA w
 Ha 1 > .ITOGG

WORD LENGTH COUNT

Il problema è simile al word count ma in questo caso si vuole contare quante parole di una **certa lunghezza** sono presenti in una collezione di documenti

- 1. Input:** repository di documenti, dove ogni documento è considerato un elemento
- 2. Map:** legge un documento ed emette una lista di coppie chiave-valore, dove la chiave è la lunghezza della parola e il valore è la parola stessa $(i, w_1), \dots, (j, w_n)$
- 3. Shuffle and Sort:** raggruppa in base alla chiave (lunghezza) e genera coppia dalla forma seguente $(1, [w_1, \dots, w_k]), \dots, (n, [w_r, \dots, w_s])$

4. **Reduce:** conta il numero di parole per ogni lista e restituisce coppia $(1, l), \dots, (p, m)$
5. **Output:** coppia del tipo (l, n) dove l è la lunghezza ed n è il numero totale di parole che hanno quella lunghezza in tutti i documenti

PSEUDO CODICE DELLA FUNZIONE MAP E DELLA FUNZIONE REDUCE

```

map (String key, String value):
    // key = lunghezza parola
    // value = parola
    for word w in value:
        EmitIntermediate( length(w), w)

reduce (String key, Iterator values):
    int count = 0
    for word w in values:
        count++
    Emit(AsString(count))

```

OTTIMIZZAZIONE CON IL COMBINING

La fase di riduzione non può iniziare prima che tutti i mapper abbiano completato il loro lavoro e abbiano emesso il risultato intermedio

Come migliorare le performance nella fase di map?

L'idea è quella di eseguire una **mini-reduce** locale sull'output di ogni mapper, andando ad evitare del lavoro ai reducer. Tale operazione può essere svolta solo quando la funzione di riduzione è **associativa e commutativa**. Questo significa che i valori da combinare possono essere combinati in qualsiasi ordine ottenendo lo stesso risultato

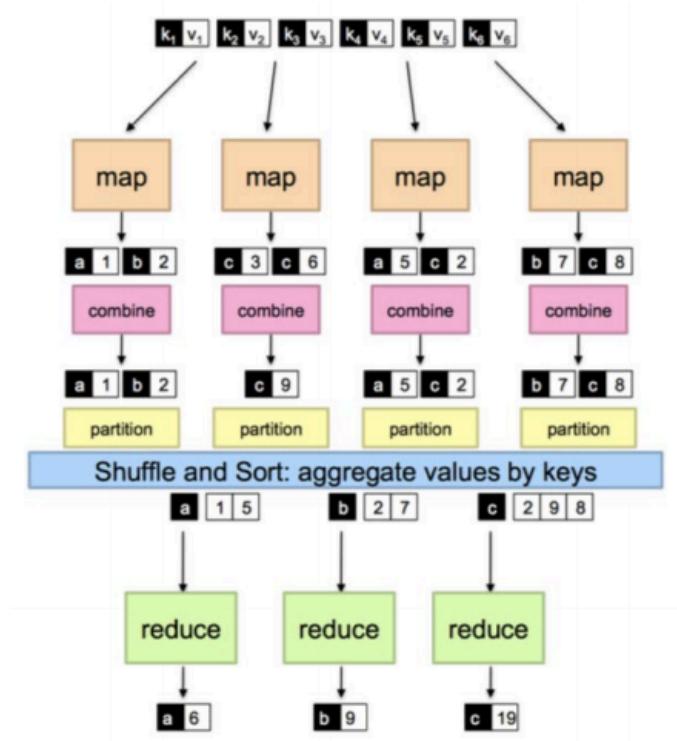
Esempio di **combine** in word count $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$

In molti casi la stessa funzione può essere usata sia per la fase di combining sia per la fase di reduce. Tra i vantaggi si ha la diminuzione del traffico di rete, ma

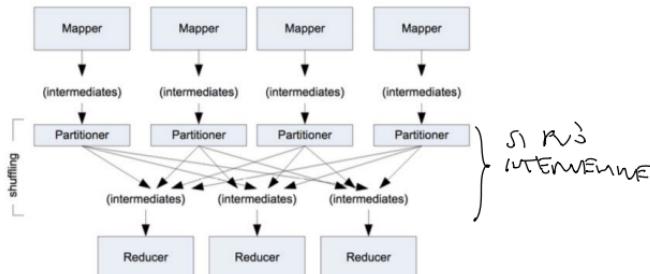
anche la diminuzione del carico di lavoro da parte dei reducer. Nonostante l'aggiunta della combine, la fase di shuffle and sort è necessaria.

WORD COUNT CON LA COMBINERS

1. **Input:** repository di documenti, dove ogni documento è considerato un elemento
2. **Map:** legge il documento e restituisce una lista di coppie chiave-valore dove le chiavi sono le parole e il valore è sempre pari ad 1 $(w_1, 1)(w_2, 1), \dots, (w_n, 1)$
3. **Combiner:** raggruppa le chiavi, somma tutti i valori 1 ed emette $(w_1, i), \dots, (w_n, j)$
4. **Shuffle and Sort:** raggruppa le chiavi uguali ed emette coppie nella forma seguente $(w_1, [p, \dots, q]), \dots, (w_n, [r, \dots, s])$
5. **Reduce:** somma tutti i valori 1 della lista e restituisci le coppie $(w_1, s_1), \dots, (w_n, s_n)$
6. **Output:** coppie del tipo (w, m) dove w è una parola che appare almeno una volta in tutti i documenti in input ed m è il numero totale di occorrenze di w tra tutti i documenti



Per dividere le chiavi intermedie si utilizza un elemento basato su hashing detto **partitioner** che assegna le coppie ai reducer



Un numero limitato di problemi può essere risolto con un job MapReduce. Alcune esempi sono:

- ricerca del più popolare URL in un file di log. Il primo job calcola il numero di pagine in cui è presente l'URL, il secondo job le riordina andando ad invertire key=frequenza e value=word

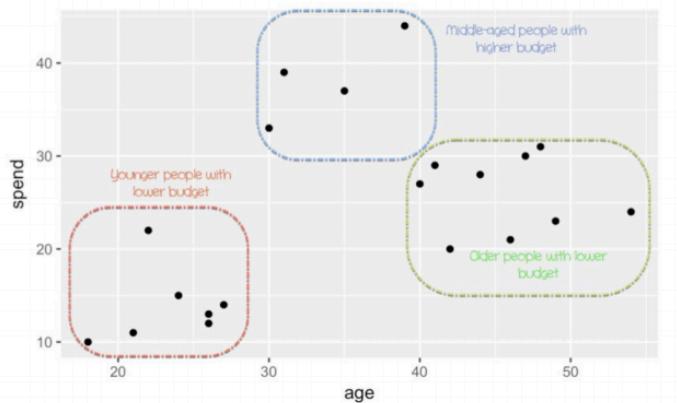
I job MapReduce possono essere **concatenati in un workflow** dove l'output di uno diventa l'input per il prossimo job. I job scrivono e leggono i dati da un file system

distribuito, questo potrebbe causare un drop delle performance

K-means CON MAP REDUCE

Il clustering è il processo di esaminazione di una collezione di "punti" per raggrupparli in "cluster" seguendo una determinata misura di distanza. Un esempio di clustering è il customer segmentation

Total spend and age of customers



DISTANZA TRA PUNTI

Esistono diverse misure di distanza

- **Distanza Euclidea:** $d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2}$ dove n è il numero di variabili indipendenti nello spazio
- **Distanza Manhattan :** $d_{Manhattan}(p, q) = \sum_{i=1}^n |p_i - q_i|$ somma dei valori assoluti

I punti più importanti sono i centroidi. Sono quei punti che hanno la posizione media di tutti i punti per ogni coordinata. Possono essere punti che non fanno parte del dataset iniziale. La distanza si misura tra i centroidi. Ogni centroide identifica un cluster

K-means

K-means è l'algoritmo di clustering più utilizzato. Utilizza la distanza Euclidea e restituisce dei valori buoni. L'algoritmo (di Lloyd) è il seguente

```

Specify desired number of clusters  $k$ ;
Initially choose  $k$  data points that are likely to be in different
clusters;
Make these data points the centroids of their clusters;
Repeat
    For each remaining data point  $p$  do
        Find the centroid to which  $p$  is nearest;
        Add  $p$  to the cluster of that centroid;
    Re-compute cluster centroids;
Until no improvement is made;

```

SINGOLA ITERAZIONE MAP-REDUCE PER K-means

Classificazione: Assegna ogni punto del dataset al centroide del cluster più vicino

$$z_i \leftarrow \arg \min_j \|u_j - x_i\|_2^2$$

La classificazione si traduce con la **Map**. Dati una coppia $(\{u_j\}, x_i)$, per ogni punto genera in parallelo tra i punti del dataset (z_i, x_i)

Riposizione: aggiorna i centroidi dei diversi cluster calcolando la media tra i punti

$$u_j = \frac{1}{n_j} \sum_{i:z_i=j} x_i$$

questa fase si svolge nella **reduce**: media, in parallelo tra cluster, tra tutti i punti nel cluster j

<pre> reduce(j, x_in_clusterj: [x_i, ...]) sum = 0 count = 0 for x in x_in_clusterj sum += x count += 1 emit (j, sum/count) </pre>	<i>Reduce on data points assigned to cluster j (have key j)</i>
	<i>Emit new centroid for cluster j</i>

Quella appena descritta è una sola iterazione del k-means. Per ottenere ottimi risultati il k-means ha bisogno di una versione iterativa di MapReduce. Ogni mapper ha bisogno di leggere un insieme di punti e tutti i centroidi (evitando moltissimi mapper). Ogni nuova iterazione bisogna condividere in broadcast la posizione dei nuovi centroidi e ripetere un'altra iterazione di MapReduce fino alla convergenza

MODELLI: WORKFLOWS

Un **workflow** identifica una serie di attività, eventi o task che devono essere completati in ordine per raggiungere un obiettivo o ottenere dei risultati

Il **Workflow Management Coalition (WMC)** definisce i workflow come

"l'automazione di un processo aziendale, in tutto o in parte, durante il quale documenti, informazioni o attività vengono passati da un partecipante all'altro per azione, secondo un insieme di regole procedurali"

I workflows sono diventati un buon modello di programmazione che permettono a scienziati e ingegneri di creare programmi complessi per l'elaborazione di repository di big data su piattaforme di elaborazione distribuita, combinando analisi dei dati, calcolo scientifico e metodi di simulazione complessi.

- Un **processo** rappresenta un insieme di attività che sono connesse tra di loro con l'obiettivo di produrre un prodotto, calcolare un risultato, fornire un servizio
- Un **task** (attività) è una parte di lavoro che rappresenta uno step logico dell'intero processo

I workflows, come modello di programmazione, rappresentano pattern ben definiti e (possibilmente) ripetibili o raggruppamenti sistematici di attività mirati a raggiungere una certa trasformazione dei dati.

I **workflow** adottano un approccio dichiarativo per esprimere la logica ad alto livello di molte applicazioni, nascondendo dettagli di basso livello non essenziali alla progettazione. Un vantaggio significativo dei workflow è la possibilità di **essere salvati e riutilizzati**, facilitando modifiche e nuove esecuzioni. Questo consente agli utenti di progettare e riutilizzare schemi comuni in più contesti.

Un **Workflow Management System (WMS)** facilita la definizione, lo sviluppo e l'esecuzione dei processi, coordinando le attività e svolgendo un ruolo chiave durante l'esecuzione del workflow.

Un workflow è strutturato come un **grafo** composto da un insieme finito di nodi e archi :

- **Vertici:** rappresentano compiti, attività o fasi specifiche del processo.

- **Archi:** rappresentano il flusso o la sequenza di esecuzione dei compiti, indicando l'ordine in cui devono essere eseguiti.

I workflow possono essere implementati come programmi software utilizzando linguaggi di programmazione, librerie o sistemi che permettono di esprimere le fasi fondamentali del workflow. Inoltre, questi strumenti forniscono meccanismi per orchestrare l'esecuzione dei workflow.

WORKFLOW PATTERN

I compiti di un workflow possono essere combinati in modi diversi per soddisfare le esigenze di varie applicazioni, sfruttando **schemi ricorrenti e riutilizzabili** (sequenziali e paralleli).

I principali **pattern** dei workflow sono:

1. **Sequenza**
2. **Diramazione (Branching)**
3. **Sincronizzazione**
4. **Ripetizione**

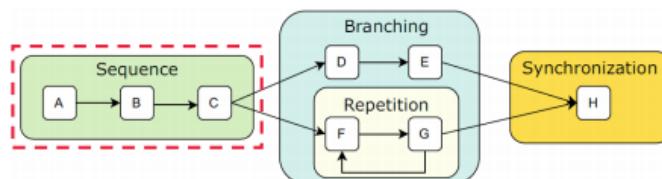


Fig. 3.3. Workflow patterns.

SEQUENZA

Il **pattern di sequenza** indica una serie di compiti che devono essere completati in un ordine specifico. Questi compiti sono connessi da **archi direzionali**, che definiscono il flusso di controllo e stabiliscono l'ordine sequenziale di esecuzione.

DIRAMAZIONE

La **diramazione** descrive situazioni in cui il workflow si divide in due o più percorsi distinti in base a certe condizioni (es. esiti di compiti precedenti, valori di dati, input utente o altri criteri rilevanti). Questo pattern consente al workflow di adattarsi dinamicamente a diversi scenari applicativi.

Esistono tre tipi di diramazioni:

1. **AND-split**: il ramo si divide in **flussi di esecuzione concorrenti**.
2. **XOR-split**: il flusso è diretto **solo verso uno** dei rami successivi, scelto sulla base di condizioni.
3. **OR-split**: il flusso si divide in **uno o più rami successivi** in base a condizioni specifiche.

SINCRONIZZAZIONE

Il **pattern di sincronizzazione** descrive situazioni in cui **più flussi di controllo** provenienti da rami diversi devono essere uniti in un unico flusso. Questi scenari sono comuni nei workflow reali, dove l'esecuzione di un compito specifico deve attendere il completamento di uno o più compiti precedenti.

Esistono tre varianti della sincronizzazione

1. **AND-join**: tutti i rami devono completarsi prima di procedere al compito successivo.
2. **XOR-join**: solo uno dei rami deve completarsi prima di procedere.
3. **OR-join**: almeno uno dei rami deve completarsi prima di trasferire il controllo al compito successivo.

RIPETIZIONE

I **pattern di ripetizione** definiscono diversi modi di specificare la ripetizione di compiti:

1. **Ciclo arbitrario**: uno o più compiti vengono ripetuti senza una struttura rigida (simile all'uso di "goto").
2. **Ciclo strutturato**:
 - **While-do**: il ciclo si ripete **finché** una condizione è soddisfatta.
 - **Repeat-until**: il ciclo si ripete **fino a quando** una condizione è soddisfatta.
3. **Ricorsione**: un compito si ripete attraverso l'**auto-invocazione**.

DIRECTED ACYCLIC GRAPHS

Un **DAG (Directed Acyclic Graph)** è un workflow che presenta le seguenti caratteristiche:

- **Diretto**: ogni compito ha almeno un predecessore o un successore (o entrambi).
- **Aciclico**: non possono esistere cicli, per evitare loop infiniti.

I **DAG** sono il modello di programmazione più usato nella gestione dei workflow e sono ampiamente adottati in framework per big data, come **Apache Spark**.

Modellano processi complessi di analisi dei dati, come il data mining. Sono utili per applicazioni in cui input, output e compiti dipendono da altre operazioni.

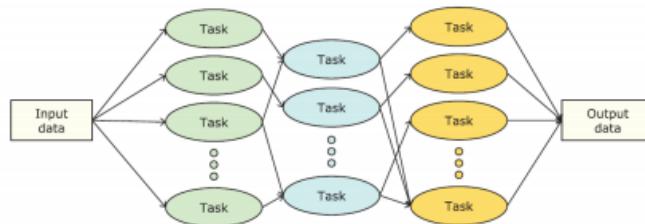


Fig. 3.4. DAG execution flow.

I DAGs possono avere due tipi di dipendenze:

- **Dipendenze sui dati**: l'output di un compito serve come input per il successivo.
- **Dipendenze sul controllo**: un compito deve essere completato prima di avviare un altro.

I task dei DAG e le loro dipendenze possono essere definite in maniera:

1. **Esplicita**: definite direttamente nel workflow (es. T2 dipende da T1).
2. **Implicita**: derivate automaticamente dal sistema analizzando le relazioni tra input e output.

Il modello **DAG** è una generalizzazione di **MapReduce**, poiché permette più fasi di mappa e riduzione, risultando in un grafo di operazioni. Offre maggiore flessibilità rispetto a MapReduce e consente una migliore ottimizzazione globale, permettendo la riorganizzazione e combinazione delle operazioni.

DIRECTED CYCLIC GRAPHS

I grafi ciclici diretti rappresentano modelli di workflow più complessi, dove i cicli indicano loop o meccanismi di iterazione.

In questo caso, il workflow spesso descrive una rete di compiti, dove:

- I nodi rappresentano servizi, componenti software o oggetti di controllo.
- Gli archi rappresentano messaggi, flussi di dati o canali di comunicazione tra i servizi.

MODELLO: MESSAGE PASSING

Il modello di passaggio di messaggi è un paradigma per la comunicazione tra processi (IPC) nell'informatica distribuita, in cui ogni elemento di elaborazione ha una memoria privata. I meccanismi di IPC, forniti dal sistema operativo, includono la memoria condivisa e la memoria distribuita o il passaggio di messaggi. I modelli di programmazione parallela sono generalmente classificati in base all'uso della memoria.

MEMORIA CONDIVISA vs MODELLO MESSAGE PASSING

Nel modello a memoria condivisa, più processi accedono a uno spazio di indirizzamento comune. Questi processi possono comunicare condividendo direttamente variabili, con una comunicazione generalmente più veloce, ma che richiede meccanismi di sincronizzazione.

Nel modello a passaggio di messaggi, un'applicazione opera come un insieme di processi indipendenti, ciascuno con una memoria locale, comunicando con altri attraverso lo scambio di messaggi. I processi mittente e destinatario devono trasferire i dati dalla memoria locale di uno a quella dell'altro. Questo modello può essere più flessibile nei sistemi distribuiti, ma può comportare un maggiore overhead di comunicazione.

La distinzione principale sta nel modo in cui i processi interagiscono e condividono i dati:

- La memoria condivisa si basa su uno spazio di indirizzamento comune.

- Il passaggio di messaggi si basa sulla comunicazione tramite scambio esplicito di messaggi.

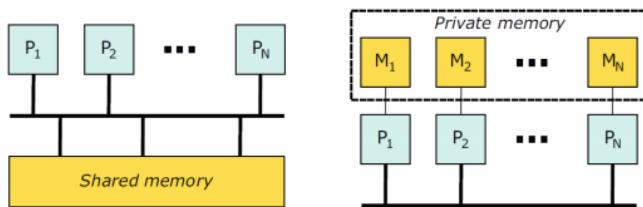


Fig. 3.5. Shared memory vs. message passing.

PRIMITIVE

Il modello di passaggio di messaggi è basato su due primitive principali:

- `Send(destinazione, messaggio)`: un processo invia un messaggio a un altro processo identificato come destinazione.
- `Receive(sorgente, messaggio)`: un processo riceve un messaggio da un altro processo identificato come sorgente.

Il processo mittente crea un messaggio contenente i dati da condividere (stringa di byte) con il processo destinatario e lo trasmette in rete eseguendo un'operazione di *send*. Il processo destinatario deve essere consapevole di attendere i dati ed eseguire un'operazione di *receive* per indicare la sua disponibilità a ricevere il messaggio.

DIVERSE PRATICHE DI IMPLEMENTAZIONE

L'implementazione pratica delle operazioni di invio e ricezione determina diverse tipologie di passaggio di messaggi, che possono essere classificate come:

- Diretto o indiretto
- Con buffer o senza buffer
- Bloccante o non bloccante

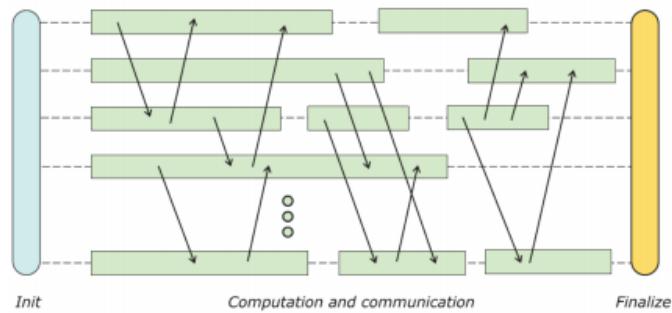


Fig. 3.6. Message-passing execution flow.

M.P. DIRETTO O INDIRETTO

Nel passaggio di messaggi diretto, esiste un collegamento diretto tra due processi per lo scambio di dati, in cui l'identità del destinatario è nota e i messaggi vengono inviati direttamente. Questo approccio manca di modularità, poiché cambiare l'identità di un processo richiede l'aggiornamento di tutti i mittenti e destinatari collegati.

Nel passaggio di messaggi indiretto, vengono utilizzate mailbox o porte per la consegna dei messaggi, che possono essere associate a un processo ricevente. A differenza del passaggio diretto, la stessa porta può essere riassegnata a un altro processo in seguito. In questo approccio, il mittente non è a conoscenza di quale processo riceverà il messaggio. Inoltre, più processi possono inviare messaggi alla stessa porta, consentendo collegamenti multi-processo e maggiore flessibilità.

M.P. BLOCCANTE E NON-BLOCCANTE

Esiste una distinzione significativa tra il passaggio di messaggi **bloccante** (sincrono) e **non bloccante** (asincrono).

Nell'invio **bloccante**, il mittente deve attendere l'accettazione del messaggio da parte del destinatario. Nella ricezione bloccante, il destinatario attende l'arrivo di un messaggio prima di continuare l'elaborazione. Le operazioni bloccanti sono spesso chiamate sincrone perché sia il mittente che il destinatario sono sincronizzati durante la comunicazione.

In un invio **non bloccante**, il mittente continua le sue operazioni senza attendere l'accettazione del messaggio. Tuttavia, il mittente può attendere una conferma in caso di errore di trasmissione. In una ricezione non bloccante, il destinatario può

ricevere un messaggio valido o nullo, con la sfida di determinare quando un messaggio è effettivamente arrivato. Se la trasmissione continua a fallire, il destinatario potrebbe attendere indefinitamente.

Le tre combinazioni fondamentali più utilizzate sono:

- **Invio bloccante e ricezione bloccante**: chiamato comunicazione "rendez-vous".
- **Invio non bloccante e ricezione non bloccante**.
- **Invio non bloccante e ricezione bloccante**: la combinazione più comune.

BUFFERING

Un modo per distinguere i modelli di passaggio di messaggi è considerare la dimensione della coda del ricevitore. Esistono tre alternative:

- **Coda a capacità zero (senza coda)**: richiede un "rendez-vous", poiché il mittente deve attendere che il destinatario sia pronto a ricevere il messaggio.
- **Coda limitata**: la coda è limitata a un numero n di messaggi o byte, quindi il mittente viene bloccato quando la coda è piena.
- **Coda illimitata**: il mittente può procedere senza attese, ma ciò può comportare rischi dovuti alle risorse fisiche limitate.

COMUNICAZIONE DI GRUPPO

Nelle applicazioni distribuite parallele, un sistema di passaggio di messaggi può necessitare di primitive di comunicazione di gruppo per migliorare le prestazioni e semplificare lo sviluppo.

- **COMUNICAZIONE UNO A MOLTI**

In questa comunicazione, un singolo mittente trasmette un messaggio a più destinatari, nota anche come **comunicazione multicast**. I processi destinatari stabiliscono un gruppo, che può essere **chiuso** o **aperto**. Nel gruppo chiuso, solo i membri possono inviare messaggi internamente. Nel gruppo aperto, qualsiasi processo del sistema può inviare messaggi all'intero gruppo.

Un caso particolare della comunicazione uno-a-molti è la **comunicazione broadcast**, in cui un messaggio viene inviato a tutti i processori collegati a una

rete.

- **COMUNICAZIONE MOLTI A UNO**

In questa comunicazione, più mittenti trasmettono messaggi a un singolo destinatario.

Il destinatario può essere, **Selettivo**, identificando un mittente specifico per lo scambio di messaggi. **Non selettivo**, rispondendo a qualsiasi mittente da un insieme predefinito.

Il non-determinismo rappresenta una sfida significativa, poiché non è certo quale dei membri del gruppo avrà il proprio messaggio elaborato per primo.

- **COMUNICAZIONE MOLTI A MOLTI**

In questa comunicazione, più mittenti possono trasmettere messaggi a più destinatari. Questo schema è flessibile e consente interazioni complesse nei sistemi distribuiti. È particolarmente utile in scenari che richiedono una comunicazione decentralizzata tra più entità. La **consegna ordinata dei messaggi** è fondamentale per garantire che tutti i destinatari ricevano i messaggi in un ordine accettabile per le applicazioni coinvolte.

MODELLI: BSP

Il **Bulk Synchronous Parallel (BSP)** è un modello di calcolo parallelo sviluppato da **Leslie Valiant** nel 1990. Valiant ha proposto un paradigma simile al modello di Von Neumann, connettendo hardware e software per macchine parallele. L'approccio BSP consente ai programmatori di evitare la gestione costosa della memoria e della comunicazione, ottenendo un calcolo parallelo efficiente con un basso grado di sincronizzazione.

Un computer BSP è costituito dai seguenti componenti:

- Un insieme di **Elementi di Elaborazione (Processing Elements, PEs)** o processori, che eseguono calcoli locali.
- Un **Router**, che gestisce la consegna dei messaggi tra coppie di PEs.
- Un **sincronizzatore hardware**, che permette ai PEs di sincronizzarsi a intervalli regolari di **L** unità di tempo (**latenza di comunicazione** o **periodicità di sincronizzazione**).

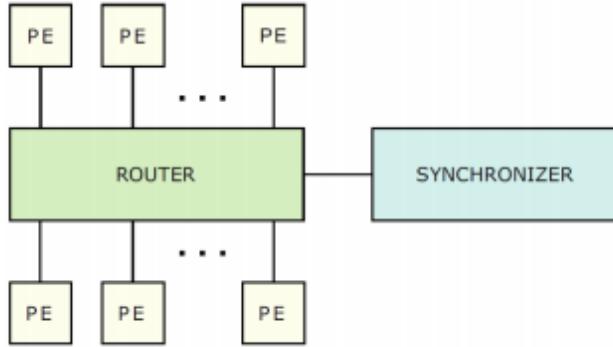


Fig. 3.7. Architecture of a BSP machine.

Una computazione nel modello BSP è costituita da una serie di **superstep**, in cui a ciascun processore viene assegnato un compito che comprende, passaggi di **calcolo locale, trasmissione** di messaggi e ricezione di messaggi

Ogni **L** unità di tempo (il parametro **periodicità**) si verifica un **controllo globale** per verificare che tutti i processori abbiano completato il superstep prima di procedere a quello successivo

Ogni superstep è composto da tre fasi ordinate:

1. **Computazione concorrente**: ogni processore esegue calcoli in modo asincrono utilizzando solo i dati locali, ossia quelli presenti nella memoria del processore stesso.
2. **Comunicazione globale**: i processi si scambiano dati in base alle richieste effettuate durante il calcolo locale.
3. **Sincronizzazione a barriera**: i processi che raggiungono una barriera devono attendere che tutti gli altri abbiano raggiunto la stessa barriera.

Comunicazione e sincronizzazione sono **separate**, garantendo **indipendenza** tra i processi in un superstep ed evitando problemi legati alla comunicazione sincrona, come **deadlock**.

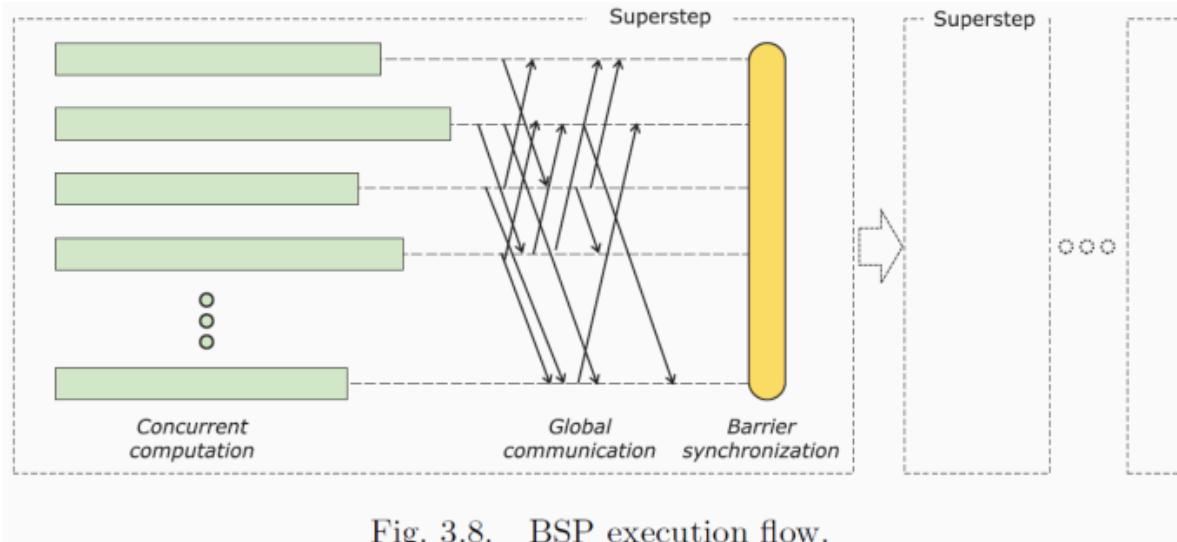


Fig. 3.8. BSP execution flow.

COMUNICAZIONE

Il modello BSP **semplifica** la gestione della comunicazione trattandola come un'operazione collettiva e imponendo un **limite di tempo** per la trasmissione di dati in batch. Tutte le operazioni di comunicazione di un superstep sono considerate **un'unica unità**, con l'assunzione che i messaggi abbiano una dimensione costante all'interno di tale unità.

Sia:

- **h** il numero massimo di messaggi in un superstep
- **g** il rapporto di throughput della comunicazione

Il tempo necessario affinché un processore invii **h** messaggi di dimensione unitaria è **hg**.

Un messaggio di lunghezza **m** è trattato come **m** messaggi di lunghezza unitaria, con un costo di comunicazione pari a **mg**.

SINCRONIZZAZIONE

Nonostante i costi potenziali, il modello BSP **dipende dalla sincronizzazione con barriere**, evitando il rischio di **dipendenze circolari, deadlock o livelock**. I costi di sincronizzazione dipendono da fattori quali, variazioni nei tempi di completamento dei calcoli locali, lo sforzo necessario per mantenere la coerenza globale tra i processori.

Per affrontare queste sfide si possono adottare strategie come:

- Assegnazione di compiti proporzionale al carico di lavoro dei processi.
- Ottimizzazione dell'efficienza della rete di comunicazione.
- Utilizzo di hardware specializzato per la sincronizzazione.
- Metodi di gestione degli interrupt.

COSTO DELL'ALGORITMO BSP

Per garantire la trasmissione di almeno **h** messaggi in un superstep, deve valere la relazione $L \geq hg$ dove **L** è la periodicità e **hg** rappresenta il tempo necessario affinché un processore invii **h** messaggi di dimensione unitaria.

Mantenere un valore basso di **g** è essenziale per evitare un aumento significativo del tempo di comunicazione.

Il costo totale di un superstep **s** è dato da $T_s = w_s + h_s g + L$ dove w_s è il costo computazionale totale del superstep.

Dato **S** il numero totali di superstep, il costo totale dell'algoritmo BSP è dato da:

$$T = \sum_{1 \leq s \leq S} T_s = \sum_{1 \leq s \leq S} (w_s + h_s g + L) = W + Hg + SL$$

dove **W** è il costo totale della computazione e **H** è il costo totale della comunicazione

BSP BASATO SU MEMORIA CONDIVISA

Il modello BSP **non supporta direttamente** la memoria condivisa, il broadcasting o l'aggregazione. Tuttavia, queste funzionalità possono essere emulate utilizzando una **Parallel Random Access Machine (PRAM)** su un computer BSP.

In un sistema PRAM esiste un **numero infinito di processori** connessi a una memoria condivisa di **capacità illimitata**. La comunicazione tra processori avviene esclusivamente attraverso la memoria condivisa e i calcoli sono completamente **sincroni**.

		WRITE	
		exclusive	concurrent
READ	exclusive	EREW	ERCW
	concurrent	CREW	CRCW

a memory cell can be accessed by not more than one processor at a time
a memory cell can be written by more than one processor at a time
a memory cell can be read by more than one processor at a time
a memory cell can be both read and written by more than one processor at a time

Fig. 3.9. Classical variations of a parallel random access machine.

Il Bulk-Synchronous PPRAM (BSPRAM) è stato introdotto da Alexandre Tiskin nel 1998 per facilitare la programmazione in stile memoria condivisa. Il BSPRAM è costituito da **p processori** con memoria locale veloce, **un'unica memoria principale condivisa**. Esegue computazioni in **superstep**, come il modello BSP ms ogni superstep include tre fasi, **input, computazione locale e output**, durante il quale i processori interagiscono con la memoria principale.

La sincronizzazione avviene **tra i superstep**, mentre il calcolo all'interno di un superstep è asincrono.

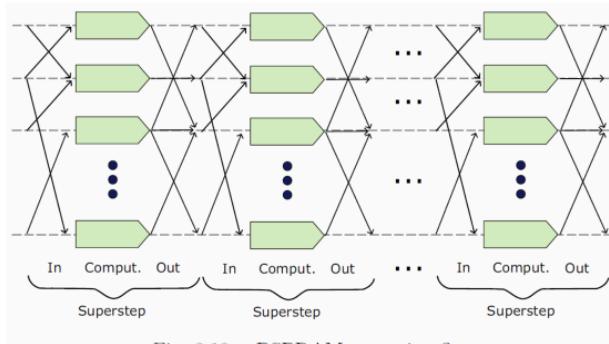


Fig. 3.10. BSPRAM execution flow.

MODELLI : SQL-like E PGAS

MODELLO SQL-like

Uno dei problemi principali dei tradizionali database relazionali è la loro incapacità di scalare orizzontalmente su più computer, limitandone l'efficacia quando si tratta di gestire enormi volumi di dati.

Per superare questo limite, è emerso l'approccio **NoSQL**, che offre un'alternativa non relazionale e consente la scalabilità orizzontale per le operazioni di lettura e scrittura del DB su più server.

Mentre i database relazionali si basano sul modello **ACID** (Atomicità, Coerenza, Isolamento, Durabilità), i database NoSQL aderiscono al modello **BASE** (Basic Availability, Soft-state, Eventual Consistency).

Quest'ultimo elimina il vincolo della consistenza immediata dopo ogni transazione, favorendo invece l'elaborazione parallela su più server, anche se ciò può comportare una consistenza solo eventuale dei dati.

Sebbene i database NoSQL siano ottimi per la scalabilità, spesso risultano inadatti per l'analisi dei dati, che è invece un punto di forza delle soluzioni **SQL-like**. Queste combinano l'efficienza del modello **MapReduce** con la semplicità di un linguaggio simile a SQL.

MapReduce è utile per la scalabilità e riduce i tempi di interrogazione, ma è complesso per utenti meno esperti.

I sistemi SQL-like semplificano le operazioni comuni come aggregazioni, selezioni e conteggi, mantenendo al contempo velocità e scalabilità.

Spesso questi sistemi ottimizzano automaticamente le query utilizzando MapReduce dietro le quinte.

Uno degli strumenti più noti per abilitare query SQL-like su big data è **Apache Hive**, che migliora le capacità di interrogazione dei sistemi basati su MapReduce. Hive consente lo sviluppo di applicazioni di analisi dati utilizzando un linguaggio simile a SQL, facilitando il lavoro degli sviluppatori.

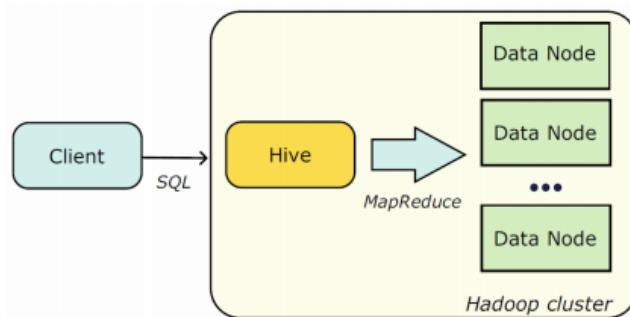


Fig. 3.11. Developing SQL on big data engines.

PERCHE' USARE SQL CON I BIG DATA?

SQL è ampiamente utilizzato da sviluppatori, amministratori di database e data scientist grazie ai suoi numerosi vantaggi:

- **Linguaggio dichiarativo**, che permette di descrivere trasformazioni di dati senza dover specificare il flusso di controllo.
- **Interoperabilità**, poiché SQL è standardizzato e diverse piattaforme possono implementarlo con sintassi compressibile da parte degli utilizzatori garantendo compatibilità.
- **Approccio data-driven**, ideale per applicazioni che richiedono elaborazioni complesse su dataset di grandi dimensioni. Le operazioni SQL riflettono trasformazioni e modifiche dei set di dati di input, rendendolo un modello di programmazione conveniente per applicazioni incentrate sui dati in ambienti tradizionali e big data

Un altro aspetto fondamentale nell'uso di SQL per i big data è la tecnica della **query-in-place**, che consente di eseguire query direttamente sui dati senza bisogno di spostarli in un database analitico separato. La tecnica query-in-place offre un cambio di paradigma nell'analisi dei big data, fornendo un mezzo potente, efficiente e conveniente per estrarre intuizioni fruibili direttamente da enormi set di dati. Questo approccio:

- **Evita la duplicazione e il trasferimento di dati**, riducendo la complessità e i costi operativi.
- **Garantisce un accesso rapido ai dati**, migliorando la latenza per interrogazioni SQL ad hoc su dataset di grandi dimensioni, offrendo disponibilità immediata dei dati e riducendo i costi operativi.

PARTIZIONAMENTO DEI DATI PER LE QUERY

Il **partizionamento dei dati** è cruciale per ottimizzare le query su big data. Il partizionamento divide una tabella in più porzioni basate su specifici valori di colonna, creando file o directory separate.

Riduce il numero di dati letti inutilmente, abbassando i costi di I/O. Migliora la velocità di esecuzione delle query. Tuttavia, un eccesso di partizioni può sovraccaricare il nodo master, che deve mantenere in memoria tutti i metadati.

MODELLO PGAS

Il PGAS è un paradigma di programmazione parallela pensato per massimizzare la produttività dei programmatore mantenendo alte prestazioni. L'idea centrale è quella di sfruttare un **indirizzamento globale dello spazio di memoria**, che offre un compromesso tra la semplicità di programmazione e l'efficienza nell'accesso ai dati, implementando anche una separazione tra accessi ai dati **locali** e **remoti**. Questa separazione nell'accesso ai dati è fondamentale per ottenere miglioramenti delle prestazioni e garantire la scalabilità su architetture parallele su larga scala.

Nel modello PGAS, il programma è eseguito da più **processi concorrenti**, ognuno operante su nodi diversi. Ogni processo ha un **rank**, che corrisponde all'indice del nodo su cui è in esecuzione. I processi accedono a una **memoria globale condivisa**, che è suddivisa in **spazi locali e remoti**. Gli indirizzi locali sono accessibili direttamente, mentre per accedere a indirizzi remoti servono chiamate API specifiche.

Un thread o un processo può ottenere un puntatore a dati che si trovano ovunque nel sistema e può leggere o scrivere dati remoti locali ad altri thread. I linguaggi PGAS distinguono tra **memoria condivisa** (accessibile a tutti i thread) e **memoria privata** (accessibile solo al thread proprietario). Ogni thread ha la sua porzione di spazio privato e una sezione di spazio condiviso.

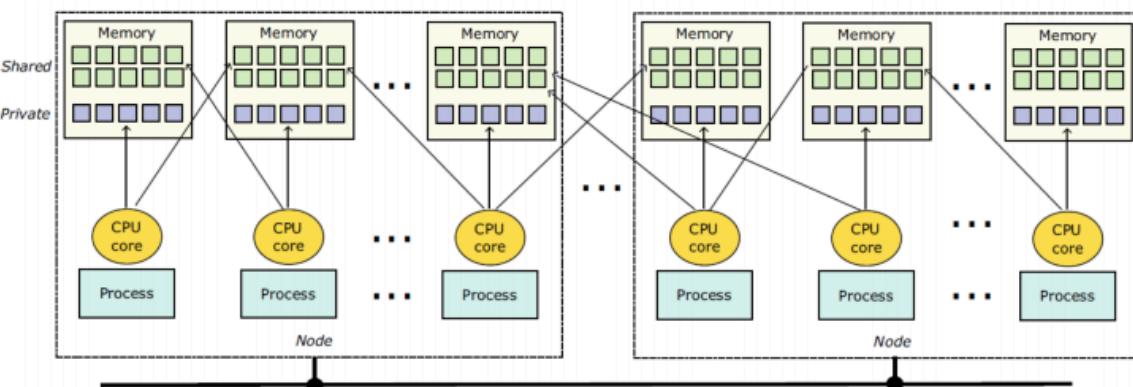


Fig. 3.12. Example of the PGAS model.

Il modello PGAS supporta tre approcci principali per l'esecuzione parallela:

1. **Single Program Multiple Data (SPMD)**: Un numero fisso di thread viene avviato all'inizio del programma e ciascuno esegue lo stesso codice.
2. **Asynchronous PGAS (APGAS)**: All'avvio del programma, un singolo thread avvia l'esecuzione al punto di ingresso del programma. Successivamente, nuovi thread possono essere generati dinamicamente per operare all'interno delle stesse partizioni dello spazio di indirizzamento o di quelle remote. Ogni thread generato può eseguire un codice diverso.
3. **Parallelismo implicito**: Non c'è alcun parallelismo visibile nel codice, come costrutti o direttive parallele, e il programma sembra descrivere un singolo thread di controllo. Tuttavia, più thread di controllo possono essere generati dinamicamente durante il runtime per velocizzare il calcolo

MEMORIA E COSTO DI PGAS

Nel modello PGAS, la memoria è suddivisa in **places**, ovvero nodi di calcolo associati a un processo o thread specifico. Un thread può accedere alla memoria del proprio **place** con un costo basso e uniforme, mentre l'accesso alla memoria di un altro place comporta costi più elevati.

Questo approccio sfrutta il modello **NUMA (Non-Uniform Memory Access)**, che distingue tra accessi **economici** (dati vicini) e **costosi** (dati lontani).

- le posizioni di memoria vicine alla fonte della richiesta di accesso sono considerate economiche, mentre le posizioni di memoria distanti sono considerate costose.
- due fattori contribuiscono al calcolo del costo: il luogo di origine della richiesta di accesso e il luogo in cui si trovano i dati richiesti

I linguaggi basati su PGAS adottano diversi schemi di distribuzione dei dati:

1. **Ciclico**: i dati vengono suddivisi in blocchi consecutivi distribuiti ciclicamente tra i nodi.
2. **Blocco**: i dati vengono suddivisi in blocchi di dimensione fissa, assegnati ai diversi nodi.
3. **Blocco-ciclico**: i dati sono divisi in blocchi di dimensione parametrizzabile e distribuiti in modo ciclico.

MODELLI: SISTEMA EXASCALE

I sistemi exascale rappresentano un'opportunità promettente, ma la loro progettazione e implementazione sono complesse a causa di sfide come scalabilità, latenza di rete, affidabilità e robustezza delle operazioni sui dati. La gestione efficiente di enormi volumi di dati richiede algoritmi scalabili capaci di partizionare e analizzare i dati attraverso milioni di operazioni parallele. I moderni sistemi HPC (High-Performance Computing) necessitano di modelli di programmazione scalabili per ottenere prestazioni ottimali, supportando i programmatore nella gestione della complessità di milioni o miliardi di thread concorrenti.

Un modello di programmazione scalabile per sistemi exascale dovrebbe includere i seguenti meccanismi:

- **Accesso parallelo ai dati**, per migliorare la larghezza di banda accedendo contemporaneamente a diversi elementi.
- **Resilienza ai guasti**, per gestire i fallimenti durante le comunicazioni non locali.
- **Comunicazione locale basata sui dati**, per limitare lo scambio di dati.
- **Elaborazione dei dati su gruppi limitati di core** in specifiche macchine exascale.
- **Sincronizzazione vicino ai dati**, riducendo il sovraccarico generato dalla sincronizzazione tra core distanti.
- **Analisi in-memory**, per diminuire i tempi di reazione memorizzando i dati nella RAM dei nodi di elaborazione.
- **Selezione dei dati basata sulla località**, per ridurre la latenza mantenendo un sottoinsieme di dati localmente disponibile.

LIMITAZIONI ODIERNE

Le soluzioni tradizionalmente utilizzate nei sistemi HPC (come **MPI**, **OpenMP** e **MapReduce**) non sono sufficienti o adatte per la programmazione di software destinato a sistemi exascale.

Le **proprietà essenziali** dei modelli di programmazione influenzate dalla transizione all'exascale sono:

- Pianificazione dei thread
- Comunicazione
- Sincronizzazione
- Distribuzione dei dati
- Controllo visivo

I sistemi exascale attuali mirano al **parallelismo con memoria distribuita**, quindi si prevede un'adozione parziale dell'**architettura a passaggio di messaggi**.

Sebbene l'**MPI (Message Passing Interface)** abbia dimostrato efficacia con milioni di core in scenari specifici, presenta alcune sfide:

- Richiede agli utenti di gestire aspetti della parallelizzazione come distribuzione di dati e lavoro, comunicazione e sincronizzazione.
- È progettato principalmente per una distribuzione statica dei dati, risultando inadatto al **bilanciamento dinamico del carico**.
- Problemi di **scalabilità** derivano dalla comunicazione **many-to-many** (molti-a-molti) nel passaggio di messaggi, che presuppone una rete completamente connessa con modelli di comunicazione densi.
- L'**I/O diventa un collo di bottiglia** nei sistemi basati su MPI, suggerendo la necessità di rivedere il modello attuale.

I sistemi exascale dovranno supportare **centinaia di core su una singola CPU o GPU**. L'uso di sistemi paralleli a **memoria condivisa di medie dimensioni** rappresenta un'alternativa praticabile al passaggio di messaggi, poiché trasferisce la responsabilità della parallelizzazione dal programmatore al compilatore. I modelli di programmazione a memoria condivisa spesso adottano un modello di controllo del parallelismo che **non gestisce la distribuzione dei dati** e impiega meccanismi di sincronizzazione **non scalabili**, come lock o sezioni atomiche. La **visione globale dei dati** promuove la sincronizzazione congiunta degli accessi remoti ai dati da parte di tutti i thread, rendendola comparabile agli accessi locali, con un impatto negativo sull'efficienza della programmazione.

I cluster composti da **nodi eterogenei**, che combinano CPU multi-core e GPU, sono sempre più utilizzati nei sistemi HPC grazie ai benefici in termini di prestazioni di picco ed efficienza energetica. Per sfruttare appieno queste piattaforme, gli sviluppatori spesso utilizzano una **combinazione di paradigmi di programmazione parallela**. Tuttavia, questa programmazione eterogenea introduce nuove sfide nella gestione di ambienti di esecuzione e modelli di programmazione differenti. Data la **propensione agli errori** nella programmazione su queste piattaforme, è necessaria l'introduzione di **nuove astrazioni, modelli di programmazione e strumenti** per affrontare queste sfide.

LINGUAGGI PER IL MODELLO EXASCALE

Le applicazioni parallele sui sistemi exascale devono gestire **milionи di thread** in esecuzione su una vasta gamma di core. Implementare strategie per **minimizzare la sincronizzazione**, ridurre la comunicazione e l'uso della memoria remota e infine affrontare eventuali **guasti software e hardware**.

Alcuni modelli di programmazione proposti per gli ambienti exascale includono:

- **Legion**

Legion è un modello di programmazione a memoria distribuita progettato per alte prestazioni su **architetture parallele eterogenee**. L'organizzazione dei dati si basa sull'uso di **regioni logiche**, che possono essere allocate, rimosse e utilizzate per memorizzare gruppi di oggetti in strutture dati. Le **regioni** possono essere fornite come input a funzioni specifiche, chiamate **task**, che leggono dati in regioni specifiche e forniscono informazioni sulla località. Le regioni logiche possono essere suddivise in sotto-regioni **distinte o sovrapposte**, fornendo informazioni cruciali per valutare l'indipendenza dei calcoli.

- **Charm++**

Charm++ è un modello di programmazione a memoria distribuita in cui un programma definisce collezioni di oggetti interagenti mappati dinamicamente sui processori dal sistema di runtime. Utilizza un approccio asincrono, basato su messaggi e task, con oggetti mobili. Gli oggetti possono essere **migrati tra processori**, permettendo alle operazioni di inviare dati a oggetti logici anziché a processori fisici. Charm++ sfrutta l'*overdecomposition*, dividendo le

applicazioni in molti piccoli oggetti che rappresentano unità di lavoro o dati, spesso superando il numero di processori disponibili.

- **DCEx**

DCEx è un modello di programmazione basato su PGAS per applicazioni parallele su larga scala nei sistemi exascale. Costruito su **operazioni di base data-aware**, permette l'uso scalabile di un **numero massivo di elementi di elaborazione**. Riduce lo scambio di dati tra thread concorrenti e impiega la sincronizzazione vicino ai dati, consentendo ai thread di eseguire calcoli in prossimità dei dati stessi. Un programma DCEx è strutturato in **blocchi di dati paralleli**, che fungono da unità di memoria per calcolo parallelo, comunicazione e migrazione.

- **X10**

X10 è un modello basato su **APGAS** che introduce il concetto di **location** come astrazione del contesto computazionale. Ogni **location** offre una **vista localmente sincrona** della memoria condivisa. Le computazioni in **X10** si svolgono in più **places**, ognuno contenente dati ed eseguendo **task** (thread leggeri) che possono essere create dinamicamente. Le attività possono **accedere sincronicamente** a una o più regioni di memoria all'interno della loro **place** di appartenenza.

- **Chapel**

Chapel è un modello di programmazione basato su **APGAS**, che utilizza astrazioni di alto livello per la programmazione parallela generale. Fornisce **strutture dati con vista globale** e una **visione globale del controllo**, migliorando il livello di astrazione sia per i dati che per il flusso di controllo. Le **strutture dati con vista globale** includono array e altre aggregazioni di dati, la cui dimensione e i cui indici vengono rappresentati globalmente, anche se l'implementazione è distribuita su più nodi del sistema parallelo. Un **local** in Chapel rappresenta un'astrazione di un'unità di memoria uniforme nell'architettura di destinazione, garantendo che tutti i thread all'interno di un locale abbiano tempi di accesso simili a una determinata locazione di memoria. La **visione globale del controllo** significa che un'applicazione inizia

con un singolo thread logico di esecuzione, introducendo parallelismo attraverso concetti specifici del linguaggio.

- **UPC++**

UPC++ è una libreria per C++ progettata per la programmazione basata su **PGAS**, che fornisce strumenti per descrivere le dipendenze tra calcoli asincroni e il trasferimento di dati. La libreria supporta una comunicazione **one-sided** efficiente, permettendo di spostare il calcolo vicino ai dati attraverso chiamate di procedura remota (**RPC**), facilitando l'implementazione di strutture dati distribuite complesse.

UPC++ si basa su tre concetti principali di programmazione:

- **Puntatori globali**, che consentono un utilizzo efficiente della località dei dati.
- **Programmazione asincrona basata su RPC**, che permette lo sviluppo efficace di programmi asincroni.
- **Futures**, per gestire la disponibilità dei dati generati dai calcoli.

TOOLS: APACHE HADOOP



Apache Hadoop è il framework open-source più popolare per implementare il modello di programmazione MapReduce

Hadoop è progettato per sviluppare applicazioni data-intensive e scalabili in diversi linguaggi di programmazione (Java, Python) per essere eseguite in parallelo su sistemi distribuiti

L'approccio di programmazione in Hadoop consente l'astrazione dai classici problemi di elaborazione distribuita, tra cui la località dei dati, il bilanciamento del

carico di lavoro, la tolleranza agli errori e il risparmio di larghezza di banda della rete.

Esistono anche altre implementazioni minori del modello MapReduce. Tra queste troviamo:

- **Phoenix++**: basato su C++ utilizza chip multi-core e multi-processori a memoria condivisa. Il suo runtime gestisce la creazione di thread, il partizionamento dei dati, la pianificazione dinamica delle attività e la tolleranza agli errori.
- **Sailfish** : framework MapReduce che sfrutta la trasmissione in batch dai mapper ai reducer. Utilizza un'astrazione chiamata *I-file* per supportare l'aggregazione dei dati, raggruppando in modo efficiente i dati scritti e letti da più nodi.

CARATTERISTICHE E MODULI

Apache Hadoop è un framework comunemente utilizzato per l'elaborazione batch, ma è inefficiente per le applicazioni altamente interattive con l'utente a causa dell'elaborazione su disco nel file system distribuito

Il progetto Hadoop è supportato da una vasta comunità open source, che fornisce aggiornamenti costanti e correzioni di bug.

Hadoop fornisce un basso livello di astrazione (**low-level abstraction**): i programmatore definiscono le applicazioni utilizzando API potenti ma non user-friendly, che richiedono una comprensione di basso livello del sistema. Lo sviluppo in Hadoop richiede più impegno e codice rispetto ai sistemi di astrazione di livello superiore (ad esempio, Pig o Hive), ma il codice è generalmente più efficiente in quanto può essere completamente ottimizzato.

Hadoop è progettato per sfruttare il parallelismo dei dati (**data-parallelism**), poiché i dati di input sono partizionati in blocchi ed elaborati in parallelo da macchine diverse.

Il framework garantisce anche un'elevata tolleranza ai guasti grazie a checkpoint e meccanismi di ripristino

Il progetto Hadoop include molti altri moduli, come:

- **Hadoop Distributed File System (HDFS)**: un file system distribuito che offre tolleranza ai guasti con ripristino automatico, portabilità su hardware e sistemi operativi eterogenei e a basso costo.
- **Yet Another Resource Negotiator (YARN)**: un framework per la gestione delle risorse del cluster e lo scheduling dei Job.
- **Hadoop Common**: librerie e altri strumenti utili che supportano gli altri moduli Hadoop.

Nel corso degli anni, Hadoop si è evoluto in una piattaforma versatile che supporta molti sistemi di programmazione, come, **Storm** per l'analisi dei dati in streaming, **Hive** per l'interrogazione di grandi set di dati, **Giraph** per l'elaborazione iterativa dei grafici e **Ambari** per il provisioning e il monitoraggio del cluster

SOFTWARE STACK DI HADOOP

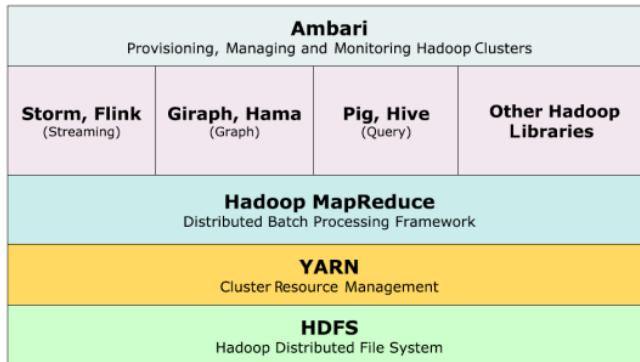


Fig. 4.1. The Hadoop software stack.

HDFS

Il file system distribuito Hadoop (HDFS) è stato progettato per archiviare grandi volumi di dati, garantendo allo stesso tempo una lettura veloce e tolleranza ai guasti.

I file in HDFS vengono distribuiti e replicati su diversi nodi di archiviazione, supportando l'organizzazione gerarchica dei file come i file system tradizionali.

Un cluster HDFS ha un'architettura master-worker ed è costituito da:

- un **namenode** (master) che gestisce il file system distribuito, mantenendo l'albero del file system e archiviando nomi e metadati;
- un insieme di **datanode** (worker) che archiviano e recuperano blocchi di dati, comunicando periodicamente con il namenode;
- un **namenode secondario** opzionale per la tolleranza ai guasti, che archivia lo stato del file system in caso di errori del namenode.

HDFS memorizza i file come una sequenza di blocchi di dati (**data blocks**), ognuno dei quali rappresenta la quantità minima di dati per la lettura o la scrittura. La dimensione predefinita del blocco è 128 MB, ma può essere configurata. Per la tolleranza agli errori, ogni blocco viene replicato tra i nodi dati con un fattore di replicazione noto, configurabile durante la creazione e la modifica del file.

Hadoop sfrutta **la località dei dati** per l'efficienza quando distribuisce i lavori di elaborazione tra i worker, riducendo al minimo il trasferimento di dati sulla rete, riducendo la congestione e aumentando la produttività complessiva del sistema

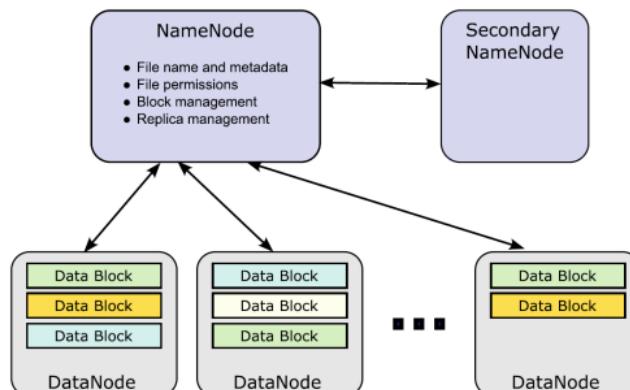


Fig. 4.2. The HDFS architecture.

FLUSSO DI ESECUZIONE

Il flusso di esecuzione in Hadoop coinvolge diversi componenti:

- **Input Data:** i file di input per un job MapReduce, in genere archiviati su HDFS.
- **InputFormat:** definisce come i dati di input vengono suddivisi e letti per creare suddivisioni di input (*input splits*).
- **InputSplit:** rappresenta la porzione di dati che verrà elaborata da una singola istanza di un mapper. Ogni suddivisione viene suddivisa in *record* prima di

essere elaborata

- **RecordReader:** converte una suddivisione di input in coppie chiave-valore adatte a essere lette ed elaborate dal mapper.
- **Mapper:** questo componente applica a ogni coppia chiave-valore una funzione di mappatura che produce un elenco di coppie chiave-valore come output.
- **Combiner:** esegue l'aggregazione locale dell'output del mapper, mirando a minimizzare il trasferimento di dati intermedi tra mapper e reducer.
- **Partitioner:** partiziona l'output dal combiner utilizzando una funzione di hashing in modo che le tuple con la stessa chiave vadano nella stessa partizione.
- **Shuffle and Sorting:** ogni partizione generata dal partitioner viene trasferita attraverso la rete ai nodi reducer (shuffling). Tuttavia, prima di inviare i dati, il framework Hadoop esegue su di essi l'ordinamento per chiave.
- **Reducer:** esegue l'aggregazione finale applicando una funzione di riduzione sui dati.
- **RecordWriter:** è responsabile della scrittura delle coppie chiave-valore di output dalla fase di riduzione nei file di output. Un componente, denominato **OutputFormat**, definisce come le coppie chiave-valore di output vengono scritte nei file di output dal record writer

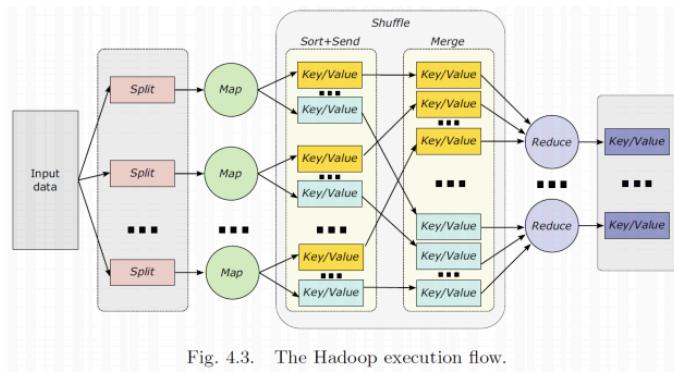


Fig. 4.3. The Hadoop execution flow.

PROGRAMMAZIONE DEL FRAMEWORK

Un programma MapReduce è costituito da almeno tre parti:

- un **mapper**, che estende la classe Mapper per fornire un'implementazione personalizzata del metodo map;
- un **reducer**, che estende la classe Reducer per fornire un'implementazione personalizzata del metodo reduce;
- un **driver**, che configura il job MapReduce e contiene la parte principale del programma.

CLASSE MAPPER

La classe **Mapper** è definita nel seguente modo:

classMapper < keyin, valuein, keyout, valueout >

La classe include i seguenti metodi che possono essere sovrascritti (override):

- **void setup(Context context)**, che viene chiamato una volta all'inizio del task.
- **void map(KEYIN key, VALUEIN value, Context context)**, che è il metodo map chiamato una volta per ogni coppia chiave-valore nella suddivisione di input.
- **void cleanup()**, che viene chiamato alla fine dell'attività map.

CLASSE REDUCER

La classe **Reducer** è definita nel seguente modo:

classReducer < keyin, valuein, keyout, valueout >

La classe include i seguenti metodi che possono essere sovrascritti (override):

- **void setup(Context context)**, che viene chiamato una volta all'inizio del task.
- **void reduce(KEYIN key, iterable<VALUEIN> values, Context context)**, che è il metodo reduce chiamato una volta per ogni chiave per elaborare tutti i valori associati ad essa.
- **void cleanup()**, che viene chiamato alla fine dell'attività reduce.

Da aggiungere PowerPoint 09 Tools - HDFS overview

CLASSE DRIVER

La **classe Driver** è responsabile della configurazione di vari aspetti del job MapReduce da eseguire in Hadoop, tra cui, il nome, i tipi di dati di input/output, le classi mapper e reducer e altri parametri.

L'oggetto

Context consente sia al mapper che al reduce di interagire con il resto del sistema Hadoop, consentendo loro di accedere ai dati di configurazione per il job MapReduce e di emettere output

SECONDARY SORT

Hadoop ordina le tuple chiave-valore intermedie in base alla chiave prima di inviarle al reducer.

L'

ordinamento secondario è una tecnica che consente di controllare l'ordinamento utilizzando una chiave composita *<primary_key, secondary_key>* per le tuple intermedie. In particolare:

1. viene definito un partizionatore personalizzato per assegnare tutte le tuple con la stessa chiave primaria a un singolo nodo reducer;
2. viene utilizzato un comparatore di ordinamento personalizzato per ordinare le tuple utilizzando l'intera chiave composita;
3. infine, utilizzando un comparatore di gruppo personalizzato, le tuple ordinate vengono raggruppate in base alla chiave primaria prima di essere inviate alla chiamata del metodo reduce.

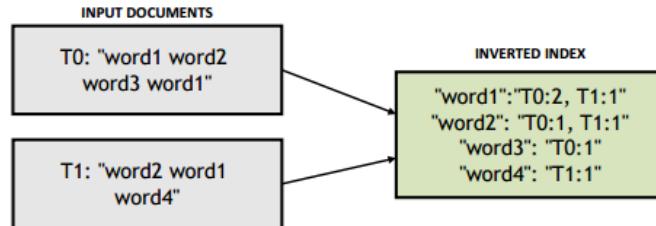
Un esempio applicativo è il seguente. Utilizzando una chiave composita *<user_id, timestamp>*, le tuple possono essere partizionate in base all'ID utente, ordinate utilizzando la chiave composita e raggruppate in base all'ID utente prima di essere elaborate dal reducer.

ESEMPIO DI APPLICAZIONE HADOOP

Hadoop può essere utilizzato per creare un inverted index per un ampio set di documenti web, che è un componente fondamentale dei sistemi di indicizzazione dei motori di ricerca.

Un'inverted index è una struttura dati contenente un set di parole (termini di

indice) e che specifica, per ogni parola, gli ID di tutti i documenti che la contengono e il numero di occorrenze.



Il flusso di esecuzione e i componenti principali (Mapper, Reducer, Combiner) per un'applicazione inverted index sviluppata con il framework Hadoop

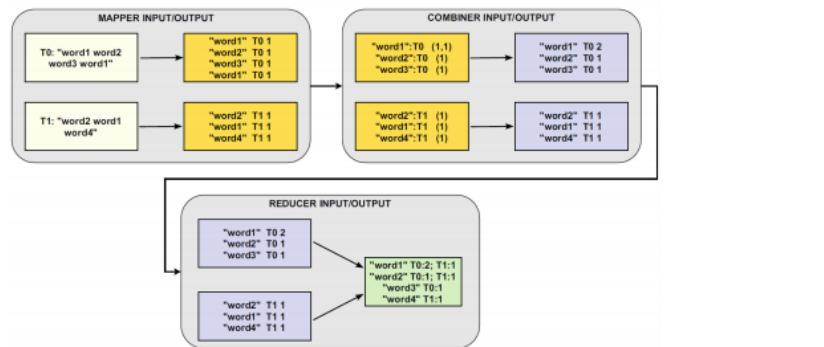


Fig. 4.4. Execution flow of the proposed Hadoop application.

La classe **MapTask** implementa il mapper, che riceve un elenco di documenti da elaborare come input. Il metodo map della classe deve:

1. Catturare il *documentID*, che è il nome file del documento attualmente elaborato.
2. Analizzare le righe di testo provenienti dai documenti di input ed emettere una coppia

<word, documentID : numberOfOccurrences>, dove *numberOfOccurrences* = 1.

Ogni parola può essere preelaborata con comuni funzioni di elaborazione del testo, come

rimozione della punteggiatura, lemmatizzazione e stemming. Hadoop usa Text e

IntWritable invece di String e Integer per ottenere una gestione più leggera della serializzazione degli oggetti.

```
public class MapTask extends Mapper<Object, Text, Text, Text> {
    private final Text keyContent = new Text();
    private final Text valueContent = new Text();
    private final static IntWritable one = new IntWritable(1);
    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        // Extract the filename from the current input split
        FileSplit fileSplit = (FileSplit) context.getInputSplit();
        String filename = fileSplit.getPath().getName();
        StringTokenizer it = new StringTokenizer(value.toString());
        while (it.hasMoreTokens()) {
            // Remove punctuation, apply lemmatization and stemming
            String word = process(it.nextToken());
            keyContent.set(word);
            valueContent.set(filename + ":" + one);
            context.write(keyContent, valueContent);
        }
    }
}
```

Listing 4.1: Inverted index Mapper.

La classe **CombineTask** implementa il combiner, un reducer utilizzato per aggregare i dati intermedi prodotti dai mapper. Il metodo reduce implementa la logica del combiner sommando tutte le occorrenze di ogni parola che compaiono più volte in un documento ed emette un elenco di coppie *<word, documentID : sumNumberOfOccurrences>*

```
public class CombineTask extends Reducer<Text, Text, Text, Text>
{
    private final Text sumContent = new Text();

    @Override
    protected void reduce(Text key, Iterable<Text> values,
        Context context) throws IOException, InterruptedException
    {
        // Sum all the occurrences of a word in the document
        HashMap<String, Integer> sumMap = new HashMap<>();
        for (Text value : values) {
            String[] parts = value.toString().split(":");
            sumMap.merge(parts[0], 1, Integer::sum);
        }
        for (Map.Entry<String, Integer> e : sumMap.entrySet()) {
            sumContent.set(e.getKey() + ":" + e.getValue());
            context.write(key, sumContent);
        }
    }
}
```

Listing 4.2: Inverted index Combiner.

La classe **ReduceTask** implementa la classe Reducer che, per ogni parola, produce l'elenco di tutti i documenti che la contengono e il numero di occorrenze in ogni documento

<word, List<documentID : numberOfOccurrences>>

L'insieme di tutte le coppie di output generate dalla funzione reduce forma l'inverted index per i documenti di input

```
public class ReduceTask extends Reducer<Text, Text, Text, Text>
{
    private final Text result = new Text();

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context
        context) throws IOException, InterruptedException {
        StringBuilder fileList = new StringBuilder();
        HashMap<String, Integer> sumMap = new HashMap<>();
        for (Text value : values) {
            String[] parts = value.toString().split(":");
            sumMap.merge(parts[0], Integer.parseInt(parts[1]),
                Integer::sum);
        }
        for (Map.Entry<String, Integer> e : sumMap.entrySet()) {
            fileList.append(e.getKey() + ":" + e.getValue())
                .append(";");
        }
        result.set(fileList.toString());
        context.write(key, result);
    }
}
```

Listing 4.3: Inverted index Reducer.

L'elenco mostra la classe driver utilizzata per impostare ed eseguire l'applicazione Hadoop. Il job viene configurato specificando: le classi da utilizzare come mapper, combiner e reducer, i formati chiave/valore di input e output utilizzati da queste classi, i percorsi di input/output dei dati

```

public class InvertedIndexJob extends Configured implements
    Tool {

    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(this.getClass());
        job.setMapperClass(MapTask.class);
        job.setCombinerClass(CombineTask.class);
        job.setReducerClass(ReduceTask.class);
        // Set the output class of key and value for the
        // mapper
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        // Set the output class of key and value for the
        // reducer
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        // Specify input and output paths
        FileInputFormat.addInputPaths(job, "webPage1,
            webPage2,...");
        FileOutputFormat.setOutputPath(job,
            new Path(args[0]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

Listing 4.4: Inverted index job.

TOOLS: SISTEMI STREAMING

Un sistema di streaming è motore di elaborazione dati progettato per elaborare dati illimitati.

I Dati limitati sono set di dati di dimensioni finite, mentre i Dati illimitati sono costituiti da un set di dati di dimensioni (teoricamente) infinite.

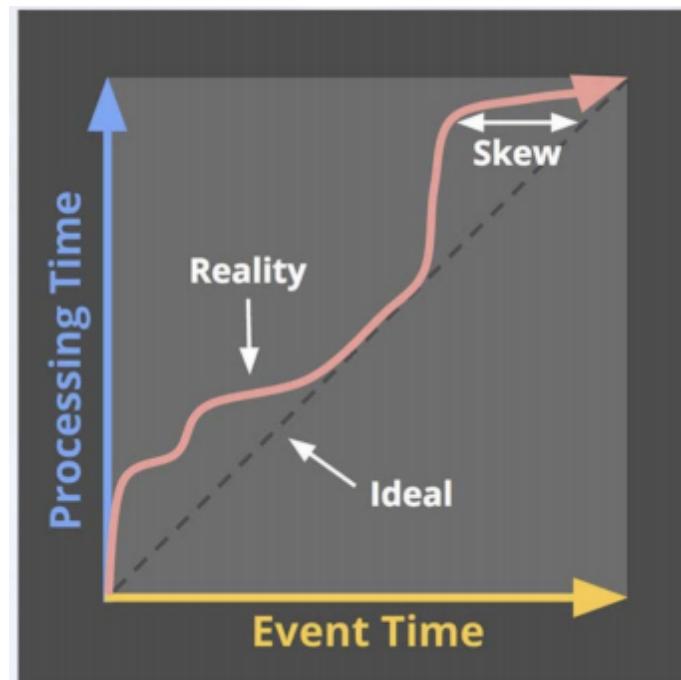
Concettualmente infinito, uno stream è costituito da dati illimitati in continua crescita di elementi o eventi. Praticamente consiste in un flusso di dati continuo che deve essere processato e analizzato

Diversi modello di processamento dei dati. Quando la produzione è controllata da una sorgente si parla di **Push Model**. Esistono anche altri modi alternativi tra cui il Publish/subscribe Model.

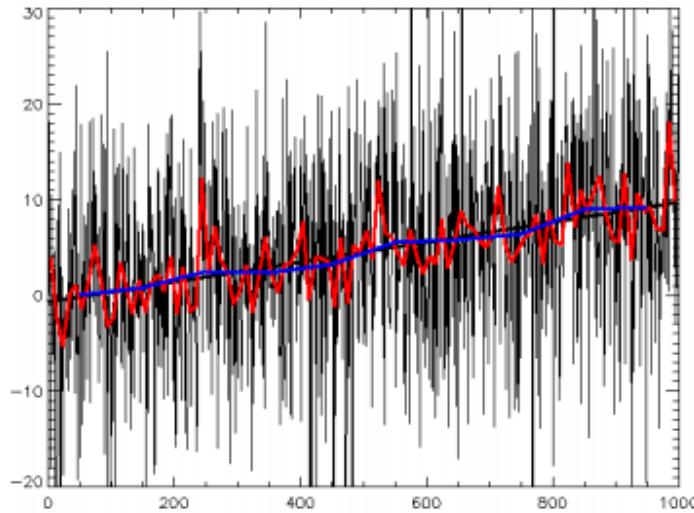
Molto importante nei sistemi di streaming è il concetto di **tempo**. Spesso è necessario capire quando i dati sono prodotti e quando i dati potranno essere

processati. Esistono diversi approcci basati sul tempo:

- Event time: Tempo necessario per produrre un singolo dato
- Ingestion time: Istante di tempo in cui il sistema riceve il dato da processare
- Processing time: Istante di tempo in cui il sistema ha terminato di elaborare il singolo dato



Una serie temporale è una serie di punti dati indicizzati in ordine temporale. Più comunemente, una serie temporale è una sequenza presa in punti successivi equamente distanziati nel tempo.



MODELLO REGISTRATORE DI CASSA vs TORNELLO

Dato un vettore $a = (a_1, \dots, a_n)$ aggiornato attraverso uno stream. Nella fase iniziale tutti $a_i = 0$. Esistono due approcci per aggiornare il vettore:

- **Registratore di cassa**

Ogni aggiornamento è nella forma $\langle i, c \rangle$ quindi gli a_i sono incrementati da un numero positivo pari a c

- **Tornello**

Ogni aggiornamento è nella forma $\langle i, c \rangle$ quindi gli a_i sono incrementati da un qualche numero (anche negativo) pari a c

ALGORITMI DI STREAMING

Sono algoritmi per l'elaborazione di flussi di dati in cui l'input viene presentato come una sequenza di elementi e può essere esaminato in pochi passaggi (in genere solo uno). Questi algoritmi possono avere accesso a una memoria limitata e ad un tempo di elaborazione limitato per elemento.

Esistono quattro approcci diversi per il processamento dei dati

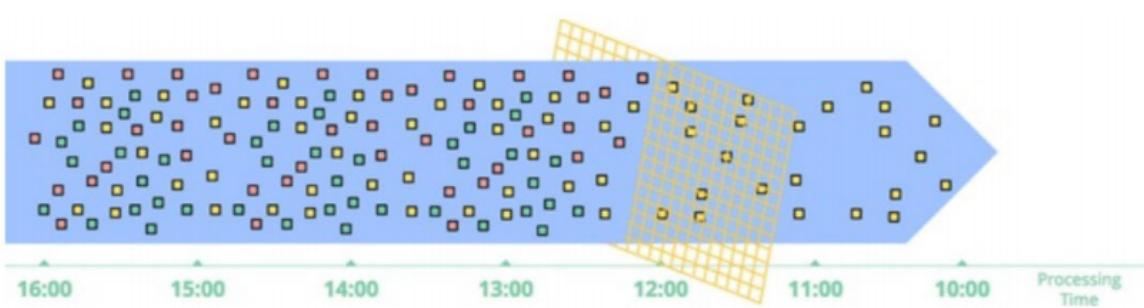
ELABORAZIONE INDIPENDENTE DAL TEMPO

L'elaborazione time-agnostic viene utilizzata nei casi in cui il tempo è essenzialmente irrilevante

Due esempi: Filtraggio, Inner Join

- **FILTRAGGIO**

Vogliamo elaborare i log del traffico web per filtrare tutto il traffico che non ha avuto origine da un dominio specifico. Possiamo esaminare ogni record quando arriva, vedere se appartiene al dominio di interesse e scartarlo in caso contrario. Dato che questo dipende da un singolo elemento in qualsiasi momento, il fatto che la fonte dati abbia una distorsione variabile del tempo dell'evento è irrilevante

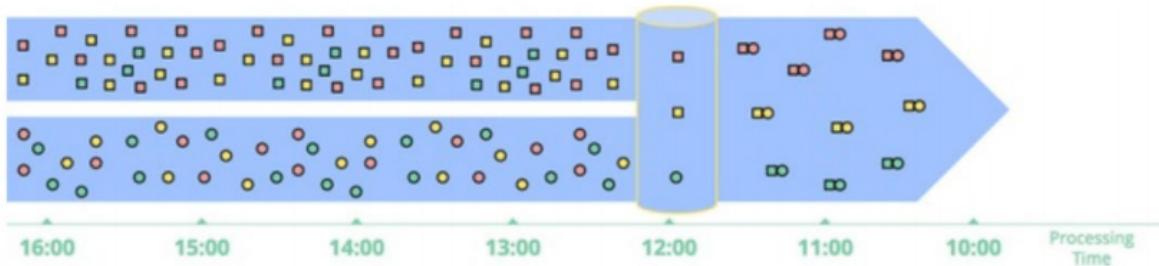


La figura mostra un esempio di filtraggio di dati illimitati: una raccolta di dati (che scorre da sinistra a destra) di vari tipi viene filtrata in una raccolta omogenea contenente un singolo tipo

- **INNER JOIN**

Vogliamo unire due fonti di dati illimitate. Quando vediamo un valore da una fonte, possiamo metterlo in buffer; dopo che arriva il secondo valore dall'altra fonte, emettiamo il record unito.

Dato che ci interessano solo i risultati di un join quando arriva un elemento da entrambe le fonti, non c'è alcun elemento temporale nella logica.



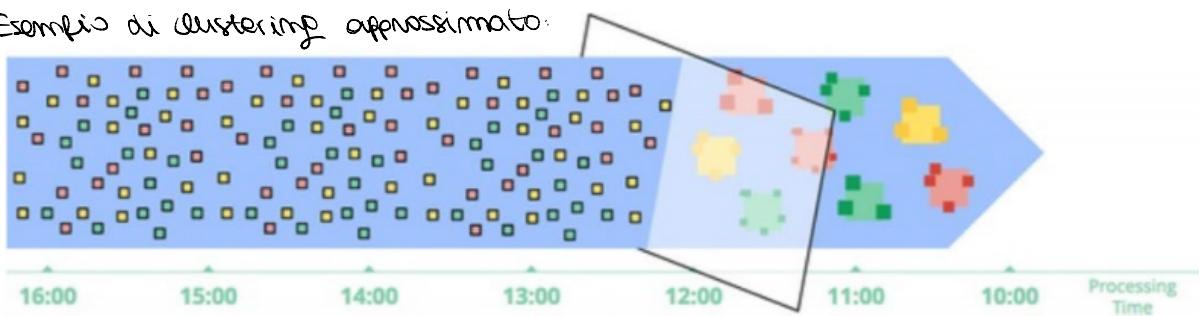
La figura mostra un esempio di esecuzione di un inner join su dati illimitati. I join vengono prodotti quando vengono osservati elementi di abbinamento da entrambe le fonti

ELABORAZIONE APPROXIMATA

L'elaborazione approssimativa si basa su algoritmi che producono una risposta approssimativa basata su un riepilogo o "schizzo" del flusso di dati. Alcuni esempi: Top-N approssimativo, streaming k-means

I dati un insieme di elementi trovare i top N dove N è un numero.

Esempio di clustering approssimato:

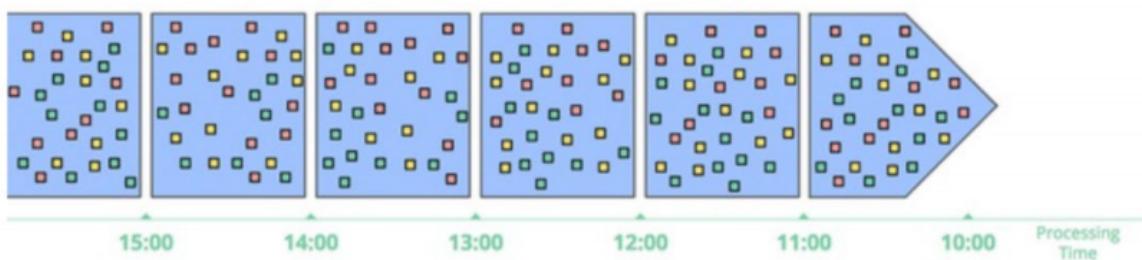


WINDOWING

La fonte dei dati (sia illimitata che limitata), viene suddivisa lungo i confini temporali in blocchi finiti per l'elaborazione. Tre tipi di finestra: Fissa, Slicing e Sessione

- **WINDOWING IN BASE AL TEMPO DI ELABORAZIONE**

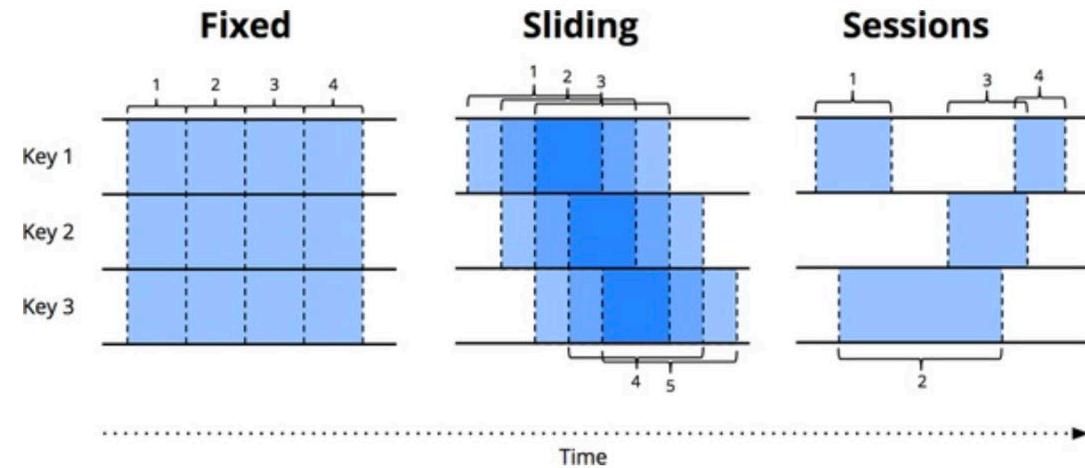
Il sistema memorizza i dati in arrivo in finestre finché non è trascorso un certo periodo di tempo di elaborazione. Esempio: memorizzazione dei dati per n minuti di tempo di elaborazione, dopodiché tutti i dati in quel intervallo di tempo vengono inviati per l'elaborazione



Windowing

- The data source (either unbounded or bounded), is chopped up along temporal boundaries into finite chunks for processing.

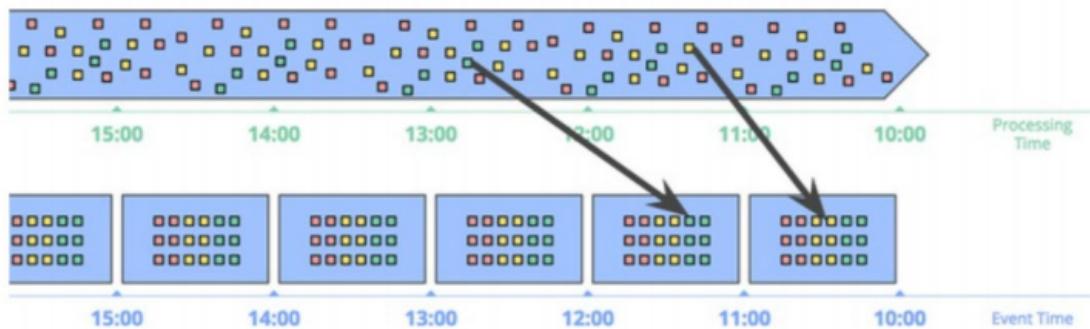
- Three patterns:
 - Fixed windows**
 - Sliding windows**
 - Sessions**



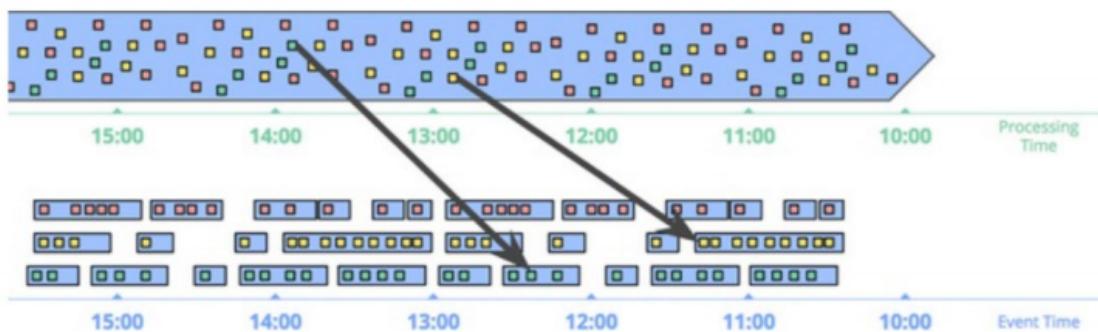
Questa figura mostra la suddivisione in finestre fisse in base al tempo di elaborazione: i dati vengono raccolti in finestre in base all'ordine in cui arrivano nella pipeline

- **WINDOWING IN BASE AGLI EVENTI**

Questo viene utilizzato quando dobbiamo osservare una fonte di dati in blocchi finiti che riflettono i tempi in cui tali eventi si sono effettivamente verificati. Più complesso del windowing in base al tempo di elaborazione (ad esempio, richiede un buffering maggiore dei dati). Spesso non abbiamo modo di sapere quando abbiamo visualizzato tutti i dati per una determinata finestra



I dati sono collezionati in finestre di dimensione fissata in base all'istante di tempo in cui vengono generati

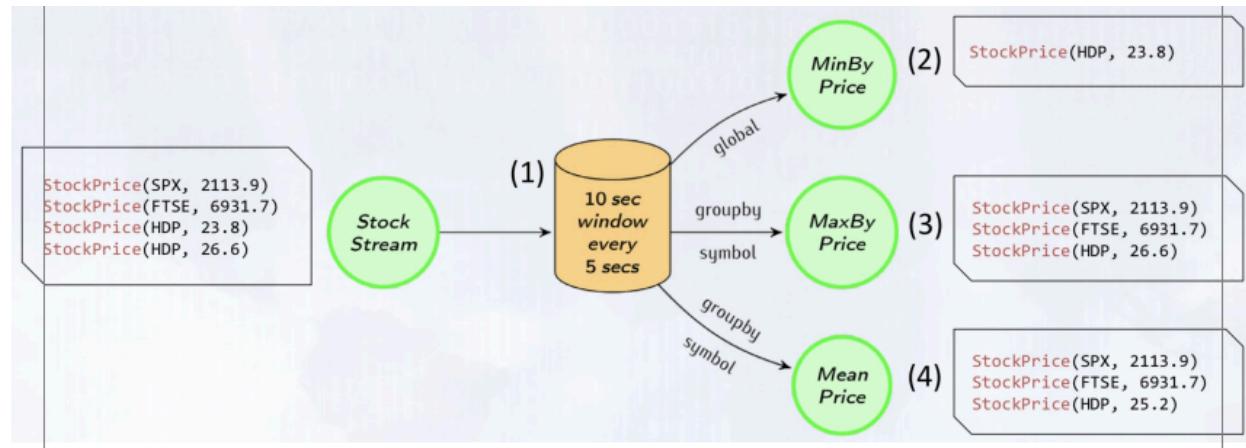


I dati sono collezionati in finestre di sessione basate sull'istante di tempo di apparizione del dato

OPERAZIONI DI BASE SUGLI STREAM

Le operazioni di base che si possono effettuare su uno stream sono di due nature diverse. Il primo consiste nell'aggregazione delle finestre mentre il secondo nell'unione delle finestre

ESEMPIO DI AGGREGAZIONE



OPERAZIONI COMPLESSE PER L'ELABORAZIONE AD EVENTI

Tra le operazioni complesse che possono essere effettuate sugli stream troviamo la ricerca di pattern significativi nel flusso. Gli eventi complessi possono essere definiti usando la logica e le condizioni temporali. Per gli eventi complessi, invece, si sceglie di modellare il sistema con un NFA

BIG DATA STREAMING

REQUISITI

- Mantenere i dati in movimento (Architettura dello stream)
- Accesso dichiarativo (StreamSQL)
- Gestione delle imperfezioni (ritardi, mancanze, elementi non ordinati)
- Risultati prevedibili (consistenza, eventi temporali)
- Integrazione della memoria e flusso dati (sistemi ibridi stream/batch)
- Sicurezza e disponibilità dei dati (tolleranza ai guasti, stato durevole)

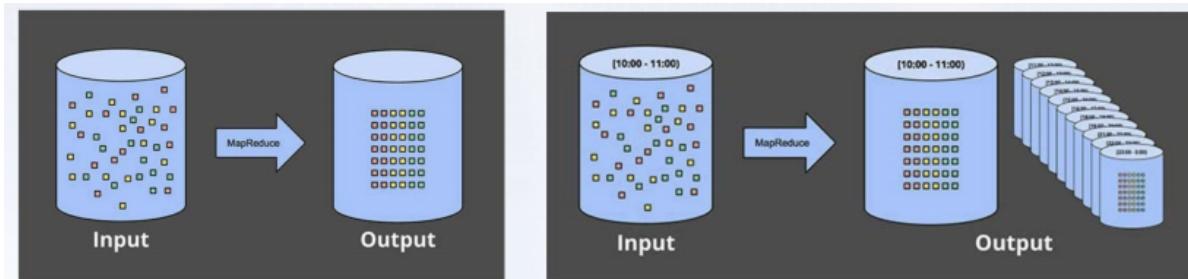
- Scalabilità e partizionamento automatico
- Processamento istantaneo e risposta rapida

ELABORAZIONE DI STREAM BIG DATA

I database possono processare una grande quantità di dati, però non possono essere utilizzati perché i Big data non sono completamente strutturati. L'idea è capire i dati attraverso operazioni di selezione, proiezione e unione.

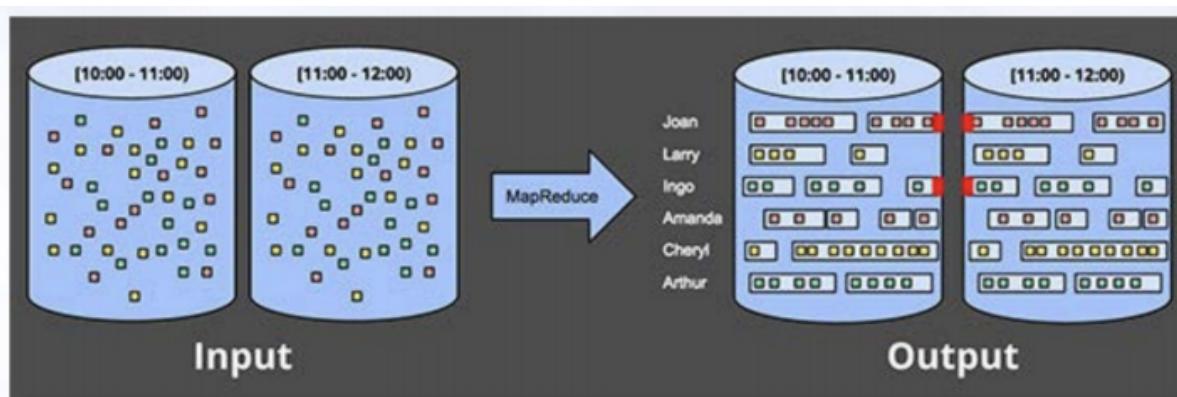
- **MAP REDUCE**

Buono per grandi quantità di dati statici. Per gli stream invece si adatta bene solo con finestre molto grandi. Inoltre i dati non si muovono quindi si ha alta latenza e bassa efficienza



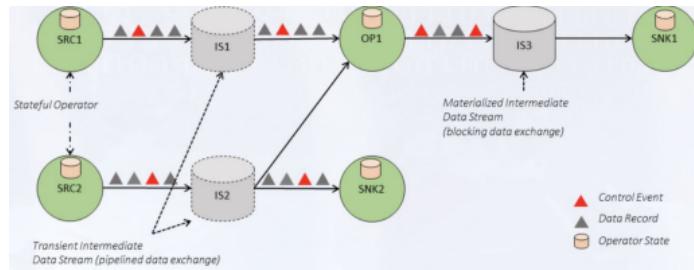
- **MINI-BATCH**

Facile da implementare con ottimi risultati sulla consistenza dei dati e sulla tolleranza ai guasti. Non è applicabile per l'elaborazione ad eventi o a sessione



ARCHITETTURA DEI SISTEMI STREAMING

Il programma può essere descritto da un DAG di operazione applicate su stream intermedi, realizzati con stream logici di record. Le operazioni sono sia computazioni sia passaggi di stato. Le possibili trasformazioni che si possono applicare ai dati sono: Map, Reduce, Filter, CoMap, Join, ecc...



WATERMARKS

I dati potrebbero giungere nel sistema in anticipo, in orario o in ritardo. Per ovviare a questa problematica si possono utilizzare i watermark.

1. Catturano il progresso della completezza del tempo-evento man mano che il tempo di elaborazione avanza.
2. Possono essere definiti come una funzione $F(P) \rightarrow E$, che prende un punto nel tempo di elaborazione e restituisce un punto nel tempo dell'evento.
3. Quel punto nel tempo dell'evento, E, è il punto fino al quale il sistema ritiene di aver osservato tutti gli input con tempi evento inferiori a E.
4. In altre parole, è un'affermazione secondo cui non verranno più visti dati con tempi evento inferiori a E.

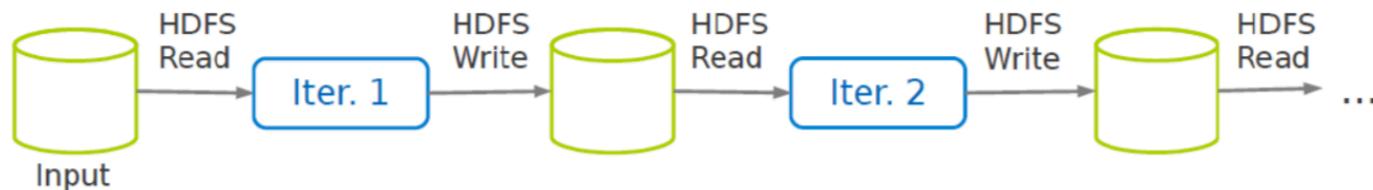
WATERMARK PERFETTI vs WATERMARK EURISTICI

Quando abbiamo una conoscenza perfetta dei dati di input, è possibile costruire una filigrana perfetta. In questo caso, tutti i dati sono in anticipo o puntuali. Le filigrane euristiche utilizzano qualsiasi informazione per fornire una stima del progresso che sia il più accurata possibile. Vengono utilizzate quando la conoscenza perfetta dei dati di input è impraticabile.

TOOLS: SPARK

MapReduce: weaknesses and limitations

- Programming model
 - Hard to implement everything as a MapReduce program
 - Multiple MapReduce steps needed even for simple operations
 - E.g., WordCount that also sorts words by their frequency
 - Lack of control, structures and data types
- No native support for iteration
 - Each iteration writes/reads data from disk: overhead
 - Need to design algorithms that minimize number of iterations



MapReduce: weaknesses and limitations

- Efficiency (recall HDFS)
 - High communication cost: computation (map), communication (shuffle), computation (reduce)
 - Frequent writing of output to disk
 - Limited exploitation of main memory
- Not feasible for real-time data stream processing
 - A MapReduce job requires to scan the entire input before processing it



Spark è un framework e un motore general-purpose, veloce per l'elaborazione dei Big data. Spark non è una versione modificata di Hadoop ma si integra perfettamente nel suo ecosistema. E' la piattaforma leader per SQL su larga scala, elaborazione batch, elaborazione di stream e anche negli ultimi tempo piattaforma dedicata al machine learning.

Caratteristica principale di Spark è **un unico motore analitico** per una grande quantità di elaborazioni dati.

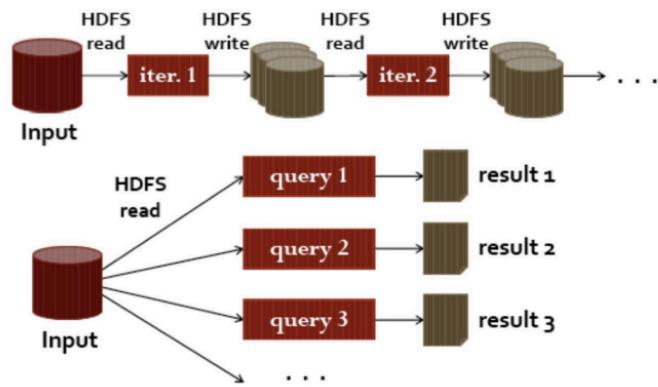
Il processamento dei dati avviene **in-memory** rendendolo molto veloce nelle elaborazioni iterative. La sua velocità è circa superiore di 10 volte (10x) rispetto ad Hadoop

Adatto anche per elaborazioni su grafi con potenti ottimizzazioni built-in.

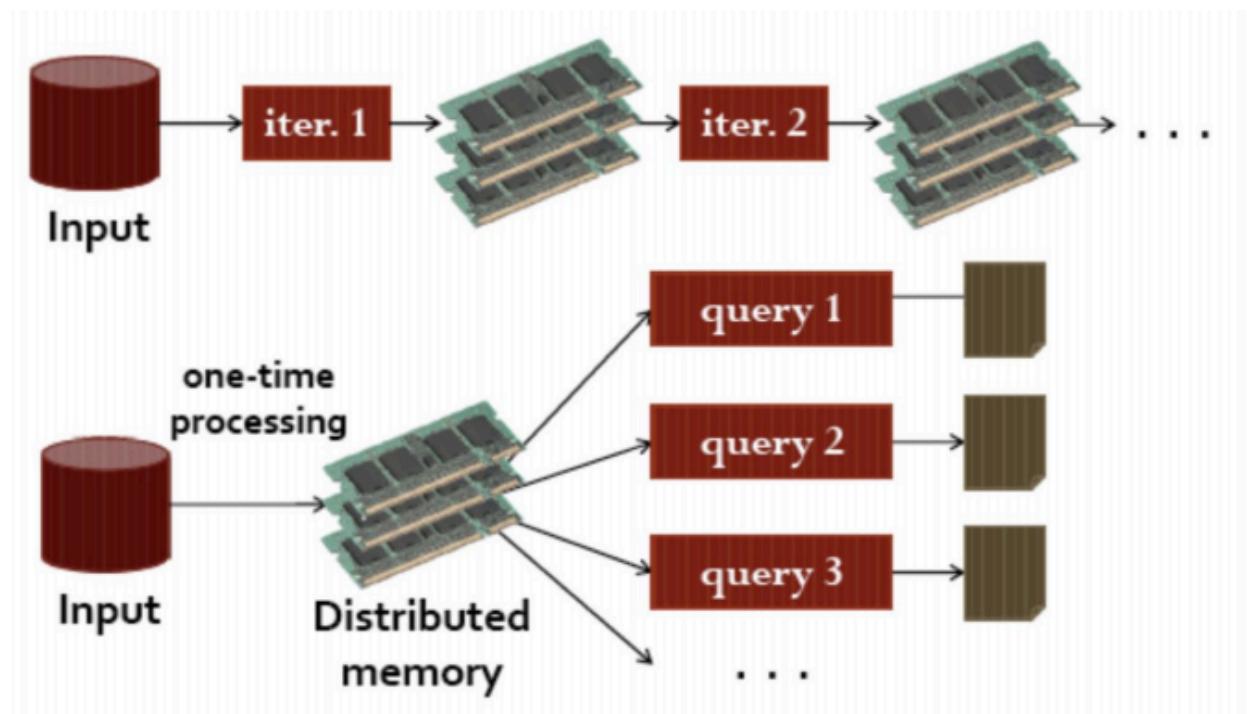
Un altro punto di forza di Spark è la sua compatibilità con le API dello storage Hadoop. Infatti si integra perfettamente con i sistemi supportati da Hadoop incluso il file system HDFS

SPARK vs HADOOP MAP-REDUCE

La condivisione dei dati in Hadoop è lenta a causa della replicazione, serializzazione e operazioni di I/O sul disco



In Spark invece si utilizza la RAM distribuita (distributed in-memory) che è circa 100x più veloce delle operazioni su disco e della rete



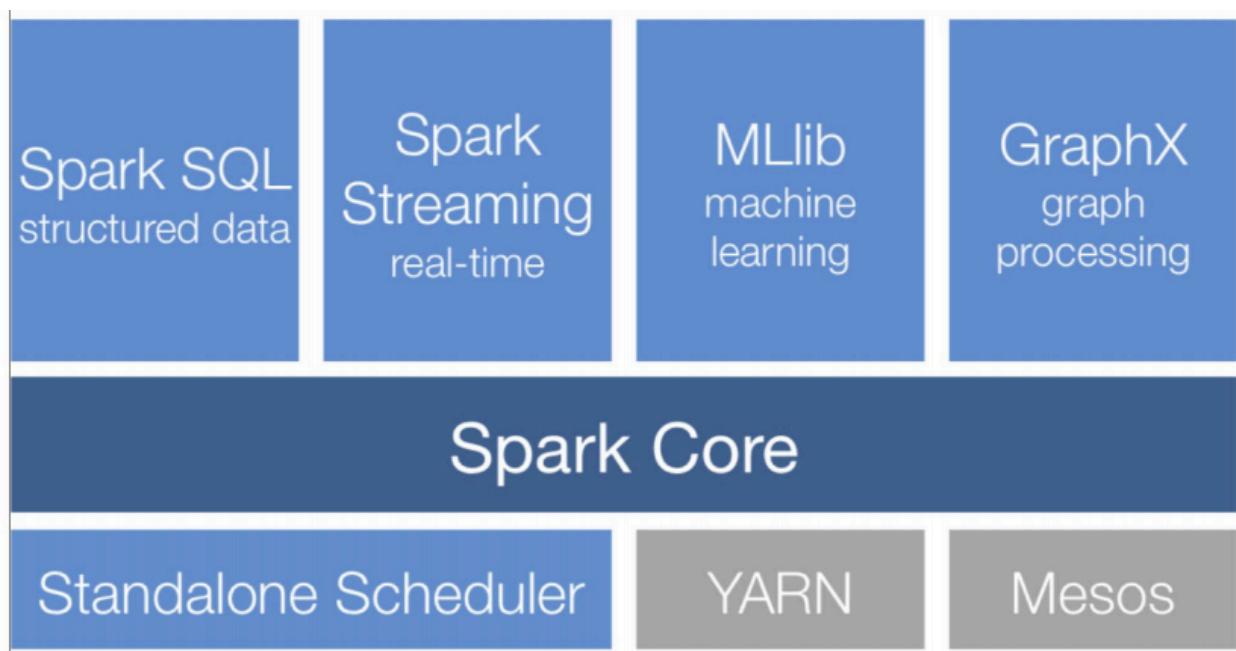
Paradigma di programmazione di base simile a MapReduce. Fondamentalmente "scatter-gather": dati e calcoli sparsi su più nodi del cluster che vengono eseguiti in parallelo su porzioni di dati; raccolta dei risultati finali.

Spark offre un modello di dati più generale tra cui RDD, DataSet, DataFrame

Spark offre un modello di programmazione più generale e intuitivo, la funzione map diventa una trasformazione in Spark, la funzione reduce invece diventa un'azione in Spark

Possono essere utilizzati anche diverse tipologie di database non solo HDFS, ma anche Cassandra, S3, file Parquet, ecc...

SPARK STACK



SPARK CORE E MOTORE UNICO

Fornisce funzionalità di base (tra cui pianificazione delle attività, gestione della memoria, ripristino degli errori, interazione con i sistemi di archiviazione) utilizzate da altri componenti. Fornisce un'astrazione dei dati chiamata **resilient distributed dataset** (RDD), una raccolta di elementi distribuiti su molti nodi di elaborazione che possono essere manipolati in parallelo. Spark Core fornisce molte API per la creazione e la manipolazione di queste raccolte, progettate nel linguaggio Scala ma perfettamente integrate con API per Java, Python e R.

Il motore unico consiste in un certo numero di moduli integrati di alto livello costruiti su Spark che possono essere combinati senza problemi nella stessa applicazione

- **Spark SQL**

Per lavorare con dati strutturati. Consente di interrogare i dati tramite SQL. Supporta molte fonti di dati (tabelle Hive, Parquet, JSON, ...). Estende l'API Spark RDD

- **Spark Streaming**

Per elaborare flussi di dati in tempo reale. Estende l'API Spark RDD

- **MLlib**

Libreria di machine learning scalabile. Integra un insieme di algoritmi distribuiti come classificazione, clustering, regressione, raccomandazione, estrazione delle feature, ecc...

- **GraphX**

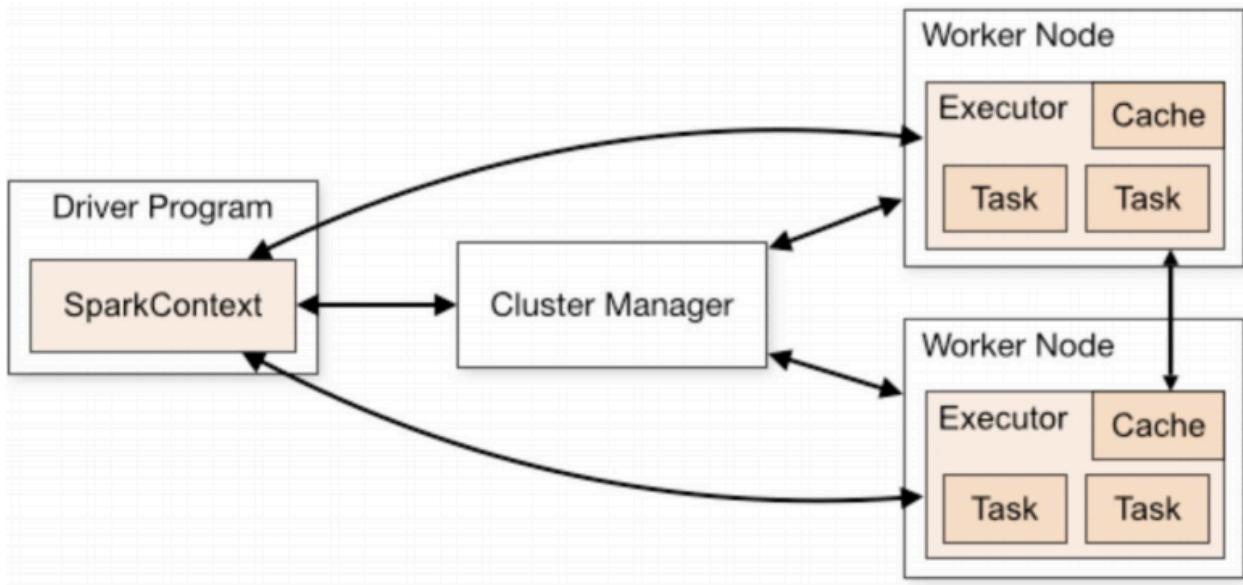
API per la manipolazione di grafi e elaborazione prestazionale parallela su grafi. Include anche algoritmi noti su grafi come il PageRank. Estende le Spark RDD API

Spark può sfruttare molti gestori di risorse di cluster per eseguire le sue applicazioni. Modalità autonoma di Spark in questo caso utilizza un semplice scheduler FIFO incluso in Spark. Può utilizzare anche Hadoop YARN, Mesos e Kubernetes

ARCHITETTURA SPARK

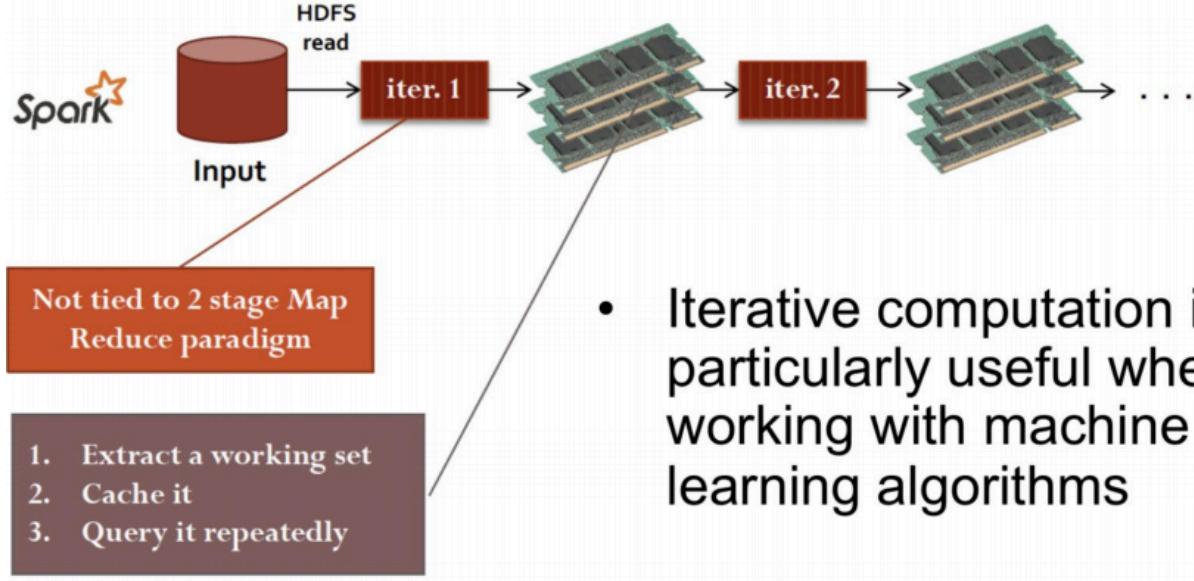
Ogni applicazione è composta da un **programma driver** e **esecutori** sul cluster. Il programma driver è un processo che esegue la funzione main() dell'applicazione e crea l'oggetto SparkContext. Ogni applicazione ottiene i propri esecutori, che sono processi che rimangono attivi per tutta la durata dell'intera applicazione ed eseguono attività in più thread. Per l'esecuzione sul cluster lo SparkContext si connette a un gestore del cluster, che alloca le risorse, una volta connesso, Spark acquisisce gli esecutori sui nodi del cluster e invia il codice dell'applicazione (ad esempio, jar) agli esecutori. Infine, SparkContext invia le attività agli esecutori per elaborare le informazioni

- Programma **Driver** che si interfaccia con il cluster manager
- Nodi **worker** dove vengono eseguiti gli **esecutori**

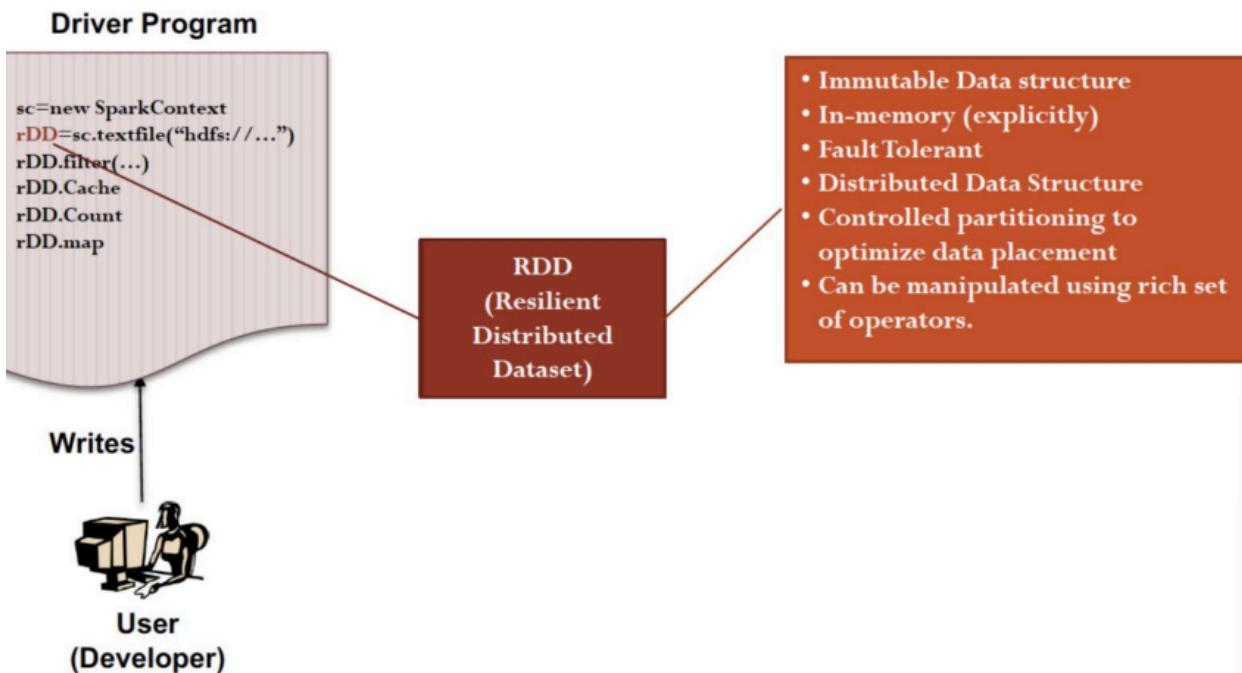
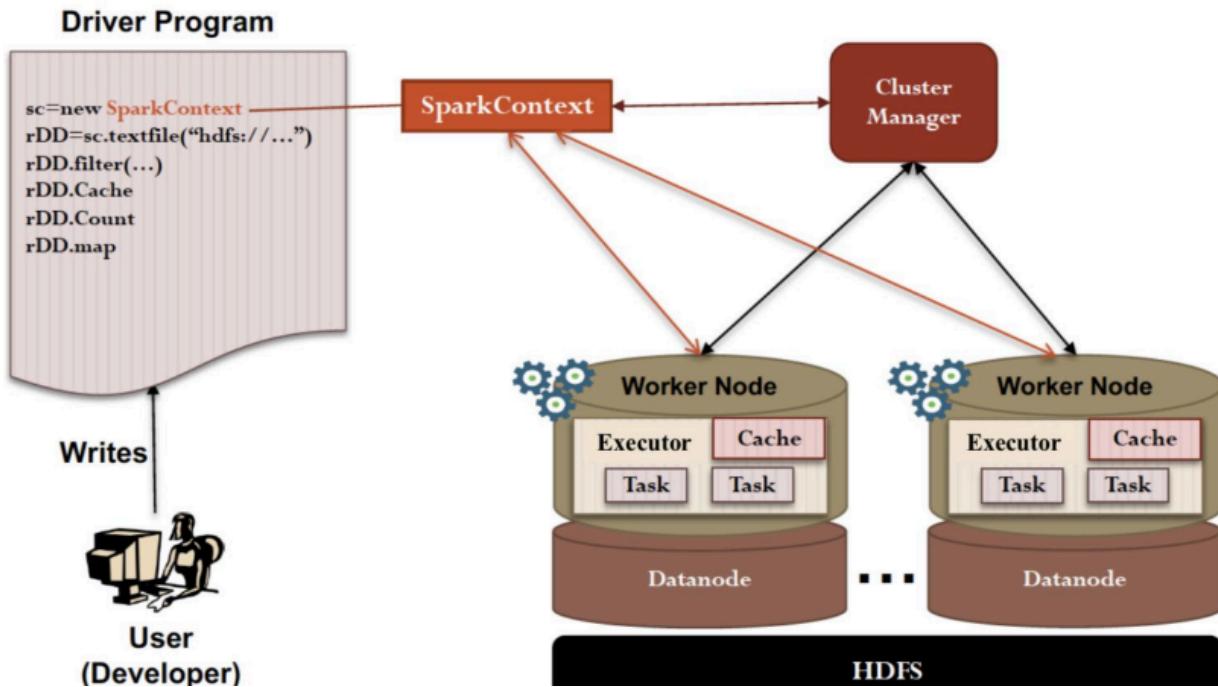


FLUSSO DATI

Si leggono i dati da input si fa un iterazione e si inserisce in memoria, poi si fa un'altra iterazione e si inserisce di nuovo in memoria e così via



MODELLO DI PROGRAMMAZIONE



RESILIENT DISTRIBUTED DATASET (RDDs)

Gli RDD sono l'astrazione di programmazione chiave in Spark: un'astrazione di memoria distribuita. Sono una raccolta immutabile, partizionata e tollerante agli

errori di elementi che possono essere manipolati in parallelo.

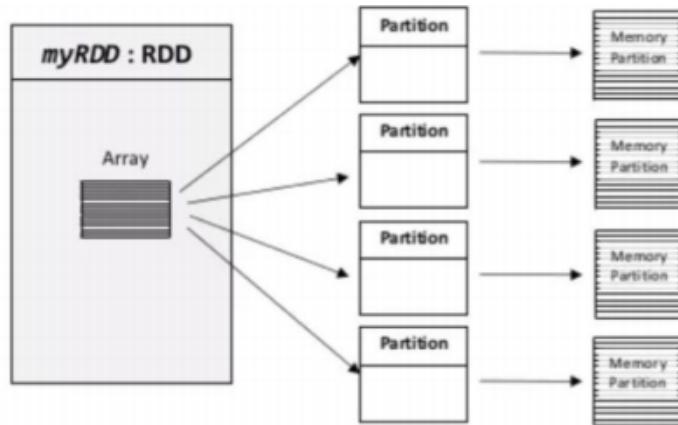
Come una `LinkedList <MyObjects>` memorizzati nella memoria principale nei nodi del cluster, ogni nodo del cluster utilizzato per eseguire un'applicazione contiene almeno una partizione degli RDD che sono definiti nell'applicazione.

Memorizzati nella memoria centrale degli esecutori in esecuzione nei nodi worker (quando possibile) o sul disco locale del nodo (se non c'è abbastanza memoria principale). Consentono di eseguire in parallelo il codice invocato su di essi. Ogni esecutore di un nodo worker esegue il codice specificato sulla sua partizione dell'RDD. Una partizione consiste in un blocco atomico di dati (una divisione logica dei dati) ed è l'unità di base del parallelismo. Le partizioni di un RDD possono essere memorizzate su diversi nodi del cluster.



Immutabili una volta costruiti ciò implica che il contenuto di un RDD non può essere modificato. Vengono creati nuovi RDD basati su RDD esistenti. Ricostruiti automaticamente in caso di errore (senza replica) tracciano le informazioni della gerarchia in modo da ricalcolare in modo efficiente i dati mancanti o persi a causa di errori del nodo. Per ogni RDD, Spark sa come è stato costruito e può ricostruirlo se si verifica un errore. Queste informazioni sono rappresentate tramite RDD lineage DAG che collega i dati di input e gli RDD.

Spark gestisce la suddivisione di RDD in partizioni e assegna le partizioni di RDD ai nodi del cluster. Spark nasconde le complessità della tolleranza agli errori infatti gli RDD vengono automaticamente ricostruiti in caso di errore utilizzando il DAG gerarchico di RDD, che definisce il piano di esecuzione logico.



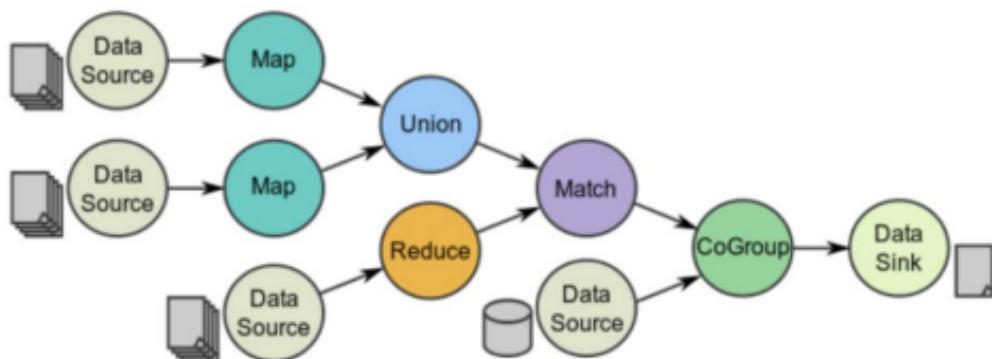
RDD API

I programmi Spark sono scritti in termini di operazioni su RDD. Gli RDD vengono creati da dati esterni o altri RDD tramite trasformazioni grossolane, che definiscono un nuovo set di dati basato su quelli precedenti come ad esempio map, filter, join, ...; ma anche manipolati attraverso azioni (conteggio, collezione, salvataggio) che avviano un lavoro da eseguire su un cluster

MODELLO DI PROGRAMMAZIONE

Basato su operatori parallelizzabili come funzioni di alto ordine che eseguono funzioni definite dall'utente in parallelo Il flusso di dati è composto da un numero qualsiasi di fonti di dati, operatori e sincronizzatori tra dati collegando i loro input e output

Il descrittore del Job è basato su un grafo aciclico diretto (DAG)



FUNZIONI DI ALTO ORDINE

Le funzioni di alto ordine sono gli operatori degli RDD. Ne esistono di due tipi, trasformazioni e azioni.

Le **trasformazioni** sono operazioni lazy che creano nuovi RDD. Il nuovo RDD che rappresenta il risultato di un calcolo non viene calcolato immediatamente ma viene materializzato su richiesta quando viene chiamata un'azione su di esso.

Le **azioni** sono operazioni che restituiscono un valore al programma driver dopo aver eseguito un calcolo sul set di dati o aver scritto dati sul disco

Le trasformazioni e le azioni disponibili in Spark

	Transformations	$map(f : T \Rightarrow U)$: $RDD[T] \Rightarrow RDD[U]$ $filter(f : T \Rightarrow Bool)$: $RDD[T] \Rightarrow RDD[T]$ $flatMap(f : T \Rightarrow Seq[U])$: $RDD[T] \Rightarrow RDD[U]$ $sample(fraction : Float)$: $RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling) $groupByKey()$: $RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$ $reduceByKey(f : (V, V) \Rightarrow V)$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $union()$: $(RDD[T], RDD[T]) \Rightarrow RDD[T]$ $join()$: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$ $cogroup()$: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$ $crossProduct()$: $(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$ $mapValues(f : V \Rightarrow W)$: $RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning) $sort(c : Comparator[K])$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $partitionBy(p : Partitioner[K])$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	Actions	$count()$: $RDD[T] \Rightarrow Long$ $collect()$: $RDD[T] \Rightarrow Seq[T]$ $reduce(f : (T, T) \Rightarrow T)$: $RDD[T] \Rightarrow T$ $lookup(k : K)$: $RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs) $save(path : String)$: Outputs RDD to a storage system, e.g., HDFS

COME CREARE UN RDD

RDD può essere creato tramite

- la parallelizzazione di raccolte esistenti del linguaggio di programmazione di hosting (ad esempio, raccolte ed elenchi di Scala, Java, Python o R) . Il numero di partizioni è specificato dall'utente. Il metodo nell'API RDD è *parallelize*
- Da file (di grandi dimensioni) archiviati in HDFS o in qualsiasi altro file system. Viene generata una partizione per ogni blocco HDFS. Il metodo nell'API RDD è *textFile*
- Trasformazione di un RDD esistente e il numero di partizioni dipende dal tipo di trasformazione le operazioni di trasformazione nell'API RDD sono *map*, *filter*,

flatMap

Trasforma una raccolta esistente in un RDD

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

sc è la variabile dello **SparkContext**. Un parametro importante è il numero di partizioni in cui dividere il set di dati. Spark eseguirà un'attività per ogni partizione del cluster (impostazione tipica: 2-4 partizioni per ogni CPU nel cluster). Spark tenta di impostare automaticamente il numero di partizioni ma si potrebbe impostarlo manualmente passandolo come secondo parametro per parallelizzare, ad esempio, `sc.parallelize(data, 10)`

Carica i dati dall'archivio (file system locale, HDFS o S3)

```
lines = sc.textFile("/path/to/README.md")
```

TRASFORMAZIONI DI UN RDD

- **Map**

prende in input una funzione che viene applicata ad ogni elemento dell'RDD.
Mappa ogni elemento in input in un altro oggetto

```
nums = sc.parallelize([1,2,3,4])
squares = nums.map(lambda x:x*x) #[1,4,9,16]
```

- **Filter**

genera un nuovo RDD filtrando il dataset in input utilizzando una specifica funzione

```
even = squares.filter(lambda num: num%2 == 0)
```

- **FlatMap**

prende in input una funzione che è applicata ad ogni elemento dell'RDD; può mappare ogni elemento in input con zero o più elementi

```
lines = sc.parallelize(["hello world","hi"])
words = lines.flatMap(lambda line:line.split(" "))
```

- **Join**

effettua una join sulle chiavi di due RDDs. In output ci sono solo le chiavi presenti in entrambi gli RDD. I candidati alla Join sono elaborati in maniera indipendente

```
users = sc.parallelize([(0,"Alex"),(1,"Bert"),(2,"Curt"),(3,"Don")])
hobbies = sc.parallelize([(0,"writing"),(0,"gym"),(1,"swimming")])
users.join(hobbies).collect()
#[(0,(Alex,writing)),(0,(Alex,gym)),(1,(Bert,swimming))]
```

- **ReduceByKey**

aggrega valori con chiave uguale usando una funzione specifica. Esegue in parallelo molte operazioni di riduzione, una per ogni chiave del dataset

```
x = sc.parallelize([("a",1),("b",1),("a",1),("a",1),("b",1),("b",1),("b",1)], 3)
y = x.reduceByKey(lambda accum, n: accum+n) #[('b',4),('a',3)]
```

AZIONI SU RDD

- **Collect**

restituisce tutti gli elementi di un RDD sottoforma di una lista

```
nums = sc.parallelize([1,2,3,4])
nums.collect() #[1,2,3,4]
```

- **Take**

restituisce un array con i primi n elementi di un RDD

```
nums.take(3) #[1,2,3]
```

- **count**

restituisce il numero di elementi dell'RDD

```
nums.count()
```

- **Reduce**

aggrega gli elementi di un RDD usando una funzione specifica

```
sum = nums.reduce(lambda x, y:x+y)
```

- **saveAsTextFile**

scrive gli elementi di un RDD come un file di testo nel file system locale oppure su HDFS

```
nums.saveAsTextFile("hdfs://file.txt")
```

TRANSFORMAZIONE LAZY

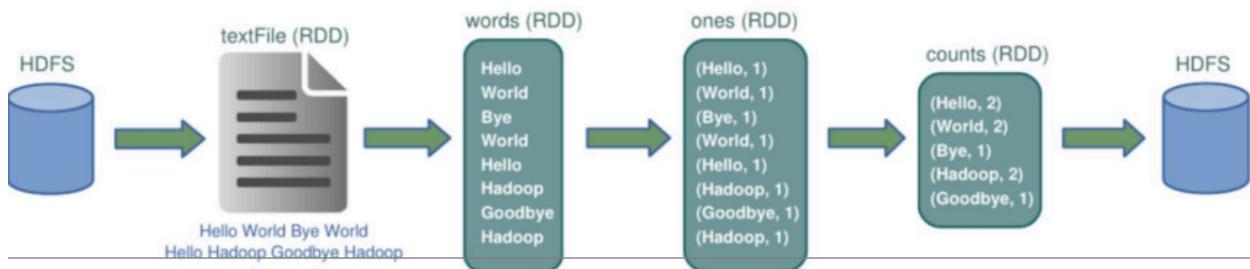
Le trasformazioni sono lazy cioè non vengono calcolate finché un'azione non richiede che un risultato venga restituito al programma driver. Questa progettazione consente a Spark di eseguire le operazioni in modo più efficiente poiché le operazioni possono essere raggruppate insieme. Ad esempio, se ci sono più operazioni di filter o map, Spark può fonderle in un unico passaggio. Un altro esempio se Spark sa che i dati sono partizionati, può evitare di spostarli sulla rete per groupBy

WORD COUNT IN SCALA

```

val textFile = sc.textFile("hdfs://...")
val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")

```



Le trasformazioni e le azioni possono essere concatenate insieme. Utilizziamo alcune trasformazioni per creare un set di dati di coppie (String, Int) chiamate conteggi e quindi salvarlo in un file

```

val textFile = sc.textFile("hdfs://...")

val counts = textFile.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")

```

INIZIALIZZAZIONE DI SPARK: SparkContext

Prima parte nel programma Spark: creare l'oggetto `SparkContext`, che è il punto di ingresso principale per le funzionalità Spark. Rappresenta la connessione al cluster Spark, può essere utilizzato per creare RDD su quel cluster. Disponibile anche nella shell, nella variabile chiamata `sc`. Può essere attivo solo uno `SparkContext` per JVM. Utilizzare `stop()` sullo `SparkContext` attivo prima di crearne

uno nuovo. Oggetto SparkConf: configurazione per un'applicazione Spark Utilizzato per impostare vari parametri Spark come coppie chiave-valore.

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

PERSISTENZA DEGLI RDD

Per impostazione predefinita, ogni RDD trasformato può essere ricalcolato ogni volta che viene eseguita un'azione su di esso. Spark supporta anche la persistenza (o memorizzazione nella cache) di RDD in memoria tra le operazioni per un rapido riutilizzo. Quando RDD viene reso persistente, ogni nodo memorizza tutte le sue partizioni che calcola in memoria e le riutilizza in altre azioni su quel set di dati (o set di dati derivati da esso). Ciò consente alle azioni future di essere molto più veloci (anche 100 volte). Per rendere persistente RDD, utilizzare i metodi *persist()* o *cache()* su di esso. La cache di Spark è fault-tolerant: una partizione RDD persa viene ricalcolata automaticamente utilizzando le trasformazioni che l'hanno creata originariamente. E' uno strumento chiave per algoritmi iterativi e un rapido utilizzo interattivo

STORAGE LEVEL

Utilizzando *persist()* puoi specificare il livello di archiviazione per persistenza di un RDD. Invocare *cache()* equivale a chiamare *persist()* con il livello di archiviazione predefinito (*MEMORY_ONLY*)

Livelli di archiviazione per *persist()*:

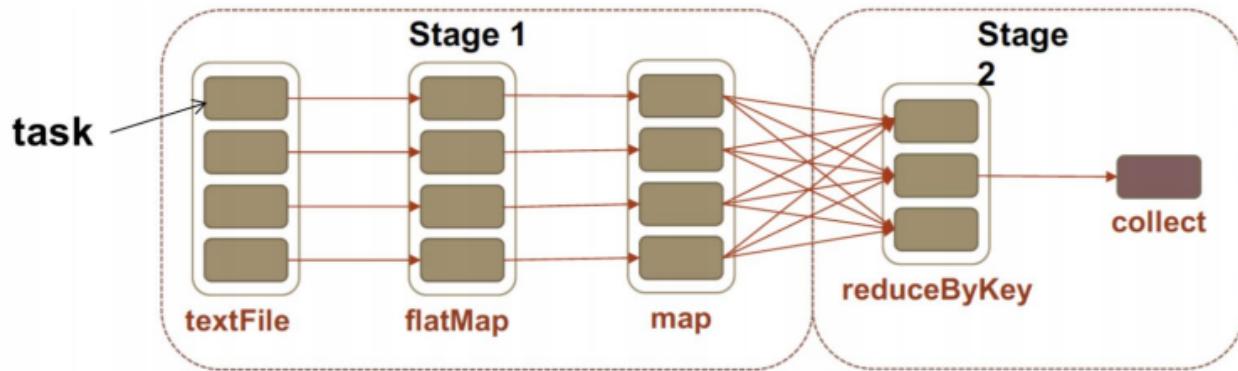
- *MEMORY_ONLY*
- *MEMORY_AND_DISK*
- *MEMORY_ONLY_SER*, *MEMORY_AND_DISK_SER* (Java e Scala)
- *DISK_ONLY*,...

Quale livello di archiviazione è il migliore? Alcune cose da considerare: cerca di mantenere in memoria il più possibile, La serializzazione rende gli oggetti molto

più efficienti in termini di spazio, ma seleziona una libreria di serializzazione veloce (ad esempio, la libreria Kryo) cerca di non salvare su disco a meno che le funzioni che hanno calcolato i tuoi set di dati non siano costose (ad esempio, filtrano una grande quantità di dati). Utilizza livelli di archiviazione replicati solo se desideri un rapido ripristino degli errori.

COME SPARK LAVORA A RUNTIME

L'applicazione crea RDD, li trasforma ed esegue azioni. Ciò si traduce in un DAG di operatori. Il DAG è compilato in fasi, esse sono sequenze di RDD senza shuffle intermedie. Ogni fase viene eseguita come una serie di attività (una attività per ogni partizione)



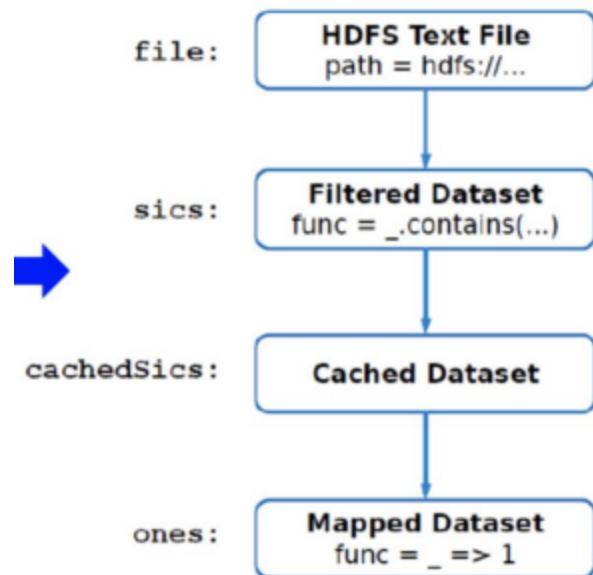
Spark crea un task per ogni partizione nel nuovo RDD e lo assegna ad un nodo worker. Tutto questo avviene internamente senza che il progettista si preoccupi di nulla

COMPONENTI SPARK

- RDD: dataset parallelo con le partizioni
- DAG: grafo logico di operazioni su RDD
- Stage: insieme di task da eseguire in parallelo
- Task: unità fondamentale dell'esecuzione di Spark

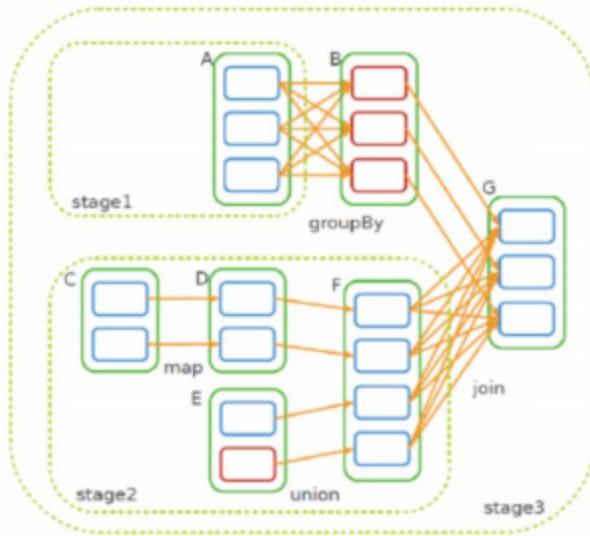
FAULT TOLERANCE IN SPARK

Gli RDD tengono traccia della serie di trasformazioni utilizzate per costruirli (gerarchia). L'informazioni della gerarchia vengono utilizzati per calcolare i dati persi. Gli RDD sono salvati come una catena di oggetti che cattura la gerarchia di ogni RDD



JOB SCHEDULING IN SPARK

Il Job scheduling di Spark tiene conto di quali partizioni di RDD persistenti sono disponibili in memoria. Quando un utente esegue un'azione su un RDD, lo scheduler crea un DAG di stage dal grafo della gerarchia di RDD. Uno stage contiene tante trasformazioni pipeline con dipendenze ristrette.



Lo scheduler avvia le attività per calcolare le partizioni mancanti da ogni fase finché non calcola l'RDD di destinazione. Le attività vengono assegnate alle macchine in base alla **località dei dati**, se un'attività necessita di una partizione, che è disponibile nella memoria di un nodo, l'attività viene inviata a quel nodo

DATAFRAME E DATASET APIs

Nelle evoluzioni delle API Spark ci sono i DataFrame e Dataset. Come gli RDD, DataFrame e Dataset sono raccolte di dati distribuite immutabili, Spark valuta in maniera lazy sia i DataFrame che i Dataset.

I DataFrame (da Spark 1.3) introducono il concetto di schema per descrivere i dati. A differenza degli RDD, i dati sono organizzati in colonne denominate, come una tabella in un database relazionale infatti funzionano solo su dati strutturati e semi-strutturati. Spark SQL fornisce API per eseguire query SQL su DataFrame con una semplice sintassi simile a SQL. Da Spark 2.0 i DataFrame sono implementati come un caso speciale di Dataset

I Dataset (da Spark 1.6) estendono i DataFrame fornendo un'interfaccia di programmazione OO sicura per i tipi quali una raccolta di dati strutturata ma tipizzata, DataFrame può essere visto come una raccolta di tipo generico Dataset[Row], dove Row è un oggetto JVM generico e non tipizzato e dataset, al contrario, è una raccolta di oggetti JVM fortemente tipizzati

La classe SparkSession diventa il punto di ingresso per entrambe le API

Offrono i vantaggi degli RDD (tipizzazione forte, capacità di utilizzare funzioni lambda) con quelli del motore di esecuzione ottimizzato di Spark SQL (ottimizzatore Catalyst). I dataset sono disponibile in Scala e Java ma non in Python ed R. Può essere costruito da oggetti JVM, essere manipolato utilizzando trasformazioni funzionali (map, filter, flatMap, ...). Hanno una valutazione lazy, ovvero il calcolo viene attivato solo quando viene invocata un'azione. Internamente, un piano logico che descrive il calcolo richiesto per produrre dati. Quando viene invocata un'azione, l'ottimizzatore di query di Spark ottimizza il piano logico e genera un piano fisico per un'esecuzione efficiente in modo parallelo e distribuito

SPARK STREAMING

Spark Streaming è un'estensione che consente di analizzare dati in streaming. Vengono inseriti e analizzati in micro-batch Utilizza un'astrazione di alto livello chiamata Dstream (discretized stream) che rappresenta un flusso continuo di dati che corrisponde ad una sequenza di RDD



SPARK MLlib

Fornisce molti algoritmi ML distribuiti tra cui Classificazione (ad esempio, regressione logistica), regressione, clustering (ad esempio, K-means), raccomandazione, alberi decisionali, randomForest e altro. Fornisce anche metodi di utilità per il machine learning come trasformazioni di funzionalità, valutazione del modello e ottimizzazione degli iperparametri. Adotta DataFrame per supportare una varietà di tipi di dati

ESEMPIO DI REGRESSIONE LOGISTICA

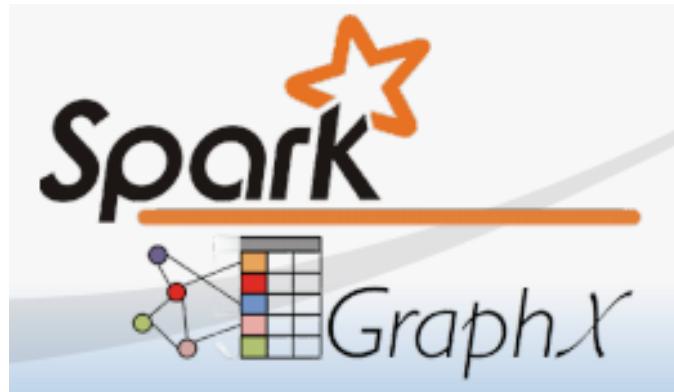
```
# Every record of this DataFrame contains the label and
# features represented by a vector.
df = sqlContext.createDataFrame(data, ["label", "features"])

# Set parameters for the algorithm.
# Here, we limit the number of iterations to 10.
lr = LogisticRegression(maxIter=10)

# Fit the model to the data.
model = lr.fit(df)

# Given a dataset, predict each point's label, and show the results.
model.transform(df).show()
```

TOOLS: GRAPHX



ANALISI SU GRAFI

Un grafo è una struttura dati composta da un insieme di vertici (noti anche come nodi) collegati da archi. I grafi sono adatti a rappresentare relazioni non lineari tra oggetti, il che ha portato alla loro applicazione in diversi domini applicativi:

- Analisi dei social network
- Rappresentazione dei dati e della conoscenza
- Ottimizzazione e routing

- Sistemi di raccomandazione
- Modellazione della diffusione delle malattie

La modellazione di queste relazioni ci consente di ottenere utili approfondimenti sui modelli sottostanti, creando rappresentazioni molto più accurate dei fenomeni analizzati. Diverse tipologie di dati possono essere naturalmente modellati come un grafo

Man mano che i dataset aumentano in dimensioni e complessità, gli strumenti tradizionali di elaborazione dei grafi diventano inefficienti. I framework di elaborazione Big Data, come Hadoop o Spark, non sono la scelta migliore quando si ha a che fare con i grafi perché:

- Non considerano la struttura del grafo sottostante ai dati.
- Il calcolo può portare a un eccessivo spostamento dei dati e a un degrado delle prestazioni.

Ciò comporta la necessità di soluzioni ad hoc, appositamente progettate per un calcolo efficiente di grafi paralleli. **Pregel** è un framework di elaborazione dei grafi sviluppato da Google, progettato per elaborare in modo efficiente grafi su larga scala su cluster di elaborazione distribuiti.

- È molto adatto per esprimere algoritmi altamente iterativi di grafi paralleli.
- Si basa sul modello **Bulk Synchronous Parallel** (BSP).

Il framework Google Pregel, progettato per supportare l'elaborazione distribuita scalabile di grafi su larga scala, si basa su due modelli computazionali principali:

- **BSP**: i vertici eseguono calcoli locali, inviano messaggi ad altri vertici e si sincronizzano tra i superstep.
- **Programmazione incentrata sui vertici**: i grafi vengono elaborati tramite funzioni che operano su singoli vertici e sui loro archi associati.

Il modello BSP fornisce un framework strutturato per il calcolo parallelo, la tolleranza ai guasti e la scalabilità. Il modello di programmazione incentrato sui vertici semplifica lo sviluppo di algoritmi su grafi consentendo ai programmati di esprimere la logica a livello di vertice, con conseguente aumento di chiarezza ed efficienza. Sebbene l'implementazione di Pregel di Google non sia disponibile al

pubblico, esistono alternative open source e framework simili, come Apache Giraph, la Gelly API di Apache Flink e GraphX di Apache Spark.

SPARK GraphX

Apache Spark GraphX è una libreria di elaborazione di grafi che fa parte del progetto Apache Spark e fornisce un framework distribuito per l'elaborazione scalabile ed efficiente di strutture di dati a grafi su larga scala. Le caratteristiche principali sono:

- Grafi distribuiti resilienti (RDG): GraphX estende gli RDD di Spark con un'astrazione a grafo, l'RDG, progettata per partizionare in modo efficiente i dati dei grafi su un cluster di macchine.
- Algoritmi di grafi: GraphX include una raccolta di algoritmi di grafi integrati, come PageRank, componenti connessi e conteggio di triangoli, semplificando l'esecuzione di comuni attività di analisi dei grafi
- Operatori di grafi: GraphX fornisce un set di operatori per la trasformazione e la manipolazione dei grafi, che possono essere utilizzati per eseguire operazioni di mappa, creare sottografi, invertire la direzione degli archi o calcolare una versione mascherata di un grafo.
- Integrazione con Spark: GraphX si integra perfettamente con altri componenti Spark, consentendo di combinare l'elaborazione di grafi con l'elaborazione dei dati e le attività di machine learning all'interno della stessa applicazione Spark.

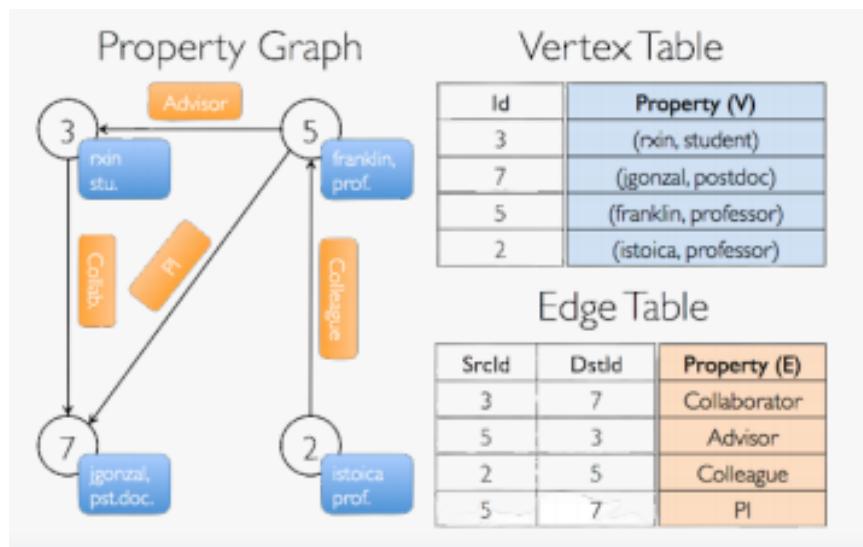
GRAFI DISTRIBUITI E RESILIENTI

GraphX estende Spark RDD introducendo il Resilient Distributed Graph, una nuova astrazione del grafo che consiste in un multigrafo diretto con proprietà associate a ciascun nodo e arco.

Questa astrazione fornisce un'interfaccia unificata per rappresentare i dati considerando la struttura del grafo sottostante, mantenendo l'efficienza degli Spark RDD. Infatti, consente di sfruttare:

- concetti di grafo e primitive efficienti per l'elaborazione
- operazioni distribuite di dati paralleli tipiche di Spark.

Un grafo contiene due RDD distinti, uno per i nodi e uno per i vertici.

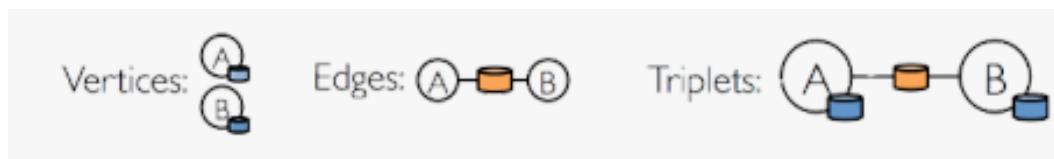


EDGETRIPLET

GraphX fornisce un'ulteriore visualizzazione della struttura del grafo sottostante, tramite il concetto di EdgeTriplet. Estende le informazioni fornite dagli edge RDD aggiungendo le proprietà dei vertici di origine e destinazione.

- Edge RDD consiste in un set di tuple (*src id, dest id, attr*) per ogni edge.
- Vertex RDD consiste in set di tuple (*vertex id, attr*) per ogni vertice del grafo.

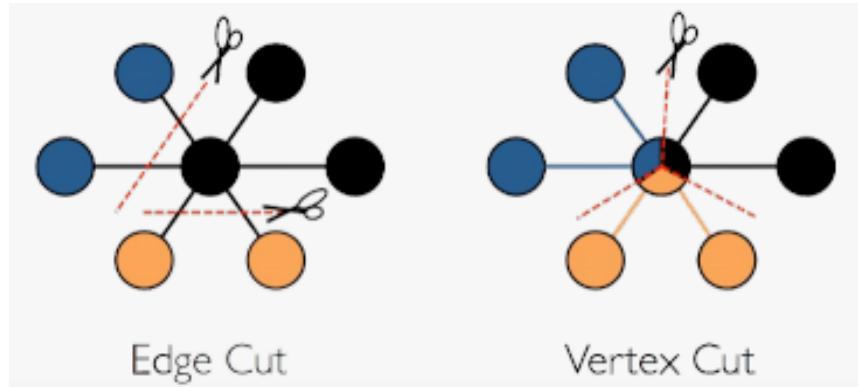
La tripletta di edge fornisce direttamente le informazioni complete su due vertici interconnessi, ovvero la tupla (*src id, dest id, src attr, edge attr, dst attr*)



PARTIZIONAMENTO VERTEX-CUT

I grafi generati in GraphX vengono partizionati per consentire lo sviluppo di applicazioni distribuite parallele a grafi scalabili. Viene sfruttato un approccio basato sui vertici per il partizionamento dei grafi, appositamente progettato per ridurre i costi di comunicazione e archiviazione. Il processo consiste nel partizionare il grafico lungo i suoi nodi, il che corrisponde all'assegnazione di archi alle macchine e all'estensione dei vertici lungo più macchine. L'operatore

`Graph.partitionBy` consente agli utenti di selezionare tra una varietà di algoritmi di partizionamento forniti da GraphX.



OPERAZIONI DI BASE

Informazioni sul grafo di input

```
// Information about the Graph
val numEdges: Long
val numVertices: Long
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]
```

Informazioni sulla topologia e sulla rappresentazione basata su RDD

```
// Views of the graph as collections
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
val triplets: RDD[EdgeTriplet[VD, ED]]
```

Trasformare gli attributi associati ai nodi, agli archi o alle triplette tramite una UDF

```
// Transform vertex and edge attributes
def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2]): Graph[VD, ED2]
```

Unione dei dati da un RDD esterno in un grafo

```
// Join RDDs with the graph
def joinVertices[U](table: RDD[(VertexId, U)])(mapFunc: (VertexId, VD, U) => VD): Graph[VD, ED]
def outerJoinVertices[U, VD2](other: RDD[(VertexId, U)])(mapFunc: (VertexId, VD, Option[U]) => VD2)
  : Graph[VD2, ED]
```

PREGEL API

GraphX fornisce un'implementazione del modello di calcolo **vertex-centric** di **Pregel**, mirato a consentire l'implementazione di applicazioni di grafi su larga scala altamente parallele.

Durante un superstep Pregel, il framework richiama una funzione definita dall'utente (UDF) per ogni vertice, che specifica il suo comportamento e viene eseguita in parallelo. Ogni vertice può modificare il suo stato e leggere i messaggi inviati nel superstep precedente o inviarne di nuovi, che verranno recapitati nel superstep successivo.

Differenze con il Pregel standard:

- GraphX esegue il calcolo dei messaggi in parallelo in funzione della edgeTriplets ed ha accesso sia al dato del nodo di origine che di destinazione.
- I nodi possono inviare messaggi solo ai loro vicini e quelli che non ricevono un messaggio all'interno di un superstep vengono saltati.

L'operatore Pregel accetta due set di parametri di input, il primo specifica il messaggio iniziale, il numero di iterazioni e la direzione dell'arco; il secondo si aspetta tre funzioni definite dall'utente:

- **vprog: (VertexId, VD, A) ⇒ VD**: codifica il comportamento del vertice. Questa UDF viene invocata su ogni vertice che riceve un messaggio e calcola il valore del vertice aggiornato.
- **sendMsg: EdgeTriplet[VD, ED] ⇒ Iterator[(VertexId, A)]**: questa UDF viene applicata ai archi in uscita dei vertici che hanno ricevuto messaggi nell'iterazione corrente.
- **mergeMsg: (A, A) ⇒ A**: specifica come due messaggi ricevuti da un vertice debbano essere uniti in un singolo messaggio dello stesso tipo. Questa UDF deve implementare una funzione commutativa e associativa.

Alla fine del calcolo, il grafo risultante viene restituito in output

```

def pregeT[A](
  initialMsg: A,
  maxIterations: Int, activeDirection: EdgeDirection)(
  vprog: (vertexId, VD, A) => VD,
  sendMsg: EdgeTriplet[VD, ED] => Iterator[(vertexId, A)],
  mergeMsg: (A, A) => A
  : Graph[VD, ED]

```

ESEMPIO DI PROGRAMMAZIONE

PageRank è un algoritmo iterativo sviluppato da Larry Page e Sergey Brin, utilizzato da Google Search per classificare le pagine web nei risultati del suo motore di ricerca. Si basa sull'idea che un collegamento da una pagina all'altra può essere visto come un voto di fiducia: le pagine con più collegamenti in entrata da fonti affidabili sono considerate più importanti o autorevoli.

PageRank modella il processo che porta un utente a una determinata pagina:

- L'utente generalmente arriva a quella pagina tramite una sequenza di collegamenti casuali, seguendo un percorso attraverso più pagine.
- L'utente può eventualmente smettere di cliccare sui collegamenti in uscita e cercare un URL diverso che non sia direttamente raggiungibile dalla pagina corrente (ad esempio, il salto casuale).

La probabilità che un utente continui a cliccare sui link in uscita è il fattore di smorzamento, generalmente impostato su $d = 0.85$, mentre la probabilità di salto casuale è $1 - d = 0.15$

Il PageRank della pagina p_i rappresenta la probabilità che un utente, che si trova su una pagina p_j , arrivi a p_i . È dato dalla somma di due probabilità:

- Il primo termine esprime la probabilità che un utente raggiunga p_i tramite un salto casuale, smettendo di cliccare sui link presenti in p_j . Si presume che la probabilità di arrivare su ciascuna delle N pagine disponibili sia equamente distribuita.
- Il secondo termine fornisce la probabilità che l'utente arrivi a pagina p_i seguendo un link esistente in p_j , supponendo che la probabilità dell'utente di seguire ogni link in p_j sia equamente distribuita.

$$PR(p_i) = \frac{1-d}{N} + d \left(\sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \right)$$

- d : è il fattore di smorzamento
- $1 - d$: probabilità di salto casuale
- N : numero di pagine disponibili
- $M(p_i)$: l'insieme di pagine che collegano a p_i
- $L(p_j)$: il numero di link nella pagina p_j

Tecniche di summarization:

- Riassunto estrattivo: è un tipo di riassunto del testo che comporta l'identificazione e l'estrazione delle informazioni più importanti (ad esempio, le frasi principali) da un documento e la loro presentazione in un riassunto conciso.
- Riassunto astrattivo: le tecniche in questa categoria sono generalmente più complesse, poiché comportano la comprensione del significato e del contesto del testo originale e la creazione di un riepilogo simile a quello umano tramite tecniche NLP come l'uso di LLM.

Applicazioni principali:

- Riepiloghi di notizie, in cui una grande quantità di testo deve essere riassunta rapidamente e accuratamente per i lettori.
- Motori di ricerca, che spesso utilizzano il riepilogo estrattivo per visualizzare brevi frammenti di pagine Web nei risultati di ricerca.
- E-learning, in cui le tecniche di riepilogo possono essere utilizzate nella tecnologia educativa per creare riepiloghi concisi e coinvolgenti di materiali didattici, come i libri di testo.
- Ricerca accademica, per aiutare i ricercatori a comprendere rapidamente le principali scoperte e i contributi di un articolo

TEXTRANK

TextRank è un algoritmo di riepilogo estrattivo proposto da Mihalcea e Tarau nel 2004. Rappresenta il testo di input come un grafo pesato di frasi semanticamente connesse ed estrae un riassunto identificando le prime k frasi più rappresentative in base a PageRank.

Formalmente, dato un testo T da riassumere, l'algoritmo crea un grafico $G = \langle S, E \rangle$:

- L'insieme $S = s_1, \dots, s_n$ contiene le n frasi presenti in T.
- L'insieme E contiene gli archi del grafo ed è creato collegando ogni coppia di frasi in S, dove
ogni connessione è associata a un peso
 $w_{i,j}$ che rappresenta la somiglianza testuale tra $s_i s_j$.

La somiglianza tra una coppia di frasi è calcolata come segue

$$w_{i,j} = \frac{|s_i \cap s_j|}{\log(|s_i + 1|) + \log(|s_j + 1|) + 1}$$

IMPLEMENTAZIONE DI TEXTRANK IN SCALA

L'operatore Pregel verrà sfruttato per realizzare una versione ponderata di PageRank, in cui la forza della connessione tra due nodi (ad esempio, frasi) è considerata

1. Definiamo la Spark session ed una funzione per calcolare il punteggio di similarità

```

//Create Spark session
val spark = SparkSession
  .builder.master("local")
  .appName("Spark-GraphX-TextRank")
  .getOrCreate()
val sc: SparkContext = spark.sparkContext

// define sentence similarity score
def sentenceSimilarity(s1: String, s2: String): Double = {
  val words1 = s1.split("[\\s,.;?!]+").map(_.toLowerCase).toSet
  val words2 = s2.split("[\\s,.;?!]+").map(_.toLowerCase).toSet
  val commonWords = words1.intersect(words2).size
  val den = log(words1.size + 1) + log(words2.size + 1) + 1
  commonWords.toDouble / den
}

```

2. Creiamo un grafo connesso di frasi dal file di testo in input. Leggi il file ed estrai le diverse frasi in esso contenute. Associa ogni frase al suo indice e calcola tutte le possibili coppie di frasi come l'autoprodotto cartesiano dell'insieme di frasi estratte. Ogni coppia ottenuta sarà un arco del grafo di frasi completamente connesso, ponderato con la similarità tra di esse.

```

def buildGraph(input_path: String): Graph[String, Double] = {
  // get sentence from textual file to be summarized
  val input_sentences = sc.textFile(input_path)
    .flatMap(line => line.split('.'))
  // each sentence is a vertex
  val vertices = input_sentences.zipWithIndex.map {
    case (sentence, index) => (index, sentence)}
  // each pair of sentences is an edge weighted with pairwise
  // similarity
  val pairs = vertices.cartesian(vertices)
    .filter { case ((i1, _), (i2, _)) => i1 != i2 }
    .map { case ((i1, s1), (i2, s2)) => (i1, i2,
      sentenceSimilarity(s1, s2)) }
    .map { case (i1, i2, sim) => Edge(i1, i2, sim) }
  Graph(vertices, pairs)
}

```

3. Una volta creato, il grafo delle frasi è pronto per l'algoritmo pageRank. I pesi degli archi sono normalizzati in modo che, per ogni nodo, la somma dei pesi degli archi in uscita sia uguale a uno. La normalizzazione viene eseguita creando una mappa contenente, per ogni nodo, la somma dei pesi dei suoi archi in uscita ottenuti sfruttando il metodo *aggregateMessages* fornito da

GraphX. Successivamente, viene utilizzata una funzione di mappa per normalizzare ogni arco in base al suo nodo sorgente e al valore corrispondente memorizzato nella mappa. Infine, a ogni nodo viene assegnata la stima iniziale tramite un'operazione *mapVertices*.

```
// normalize edge weights and set initial guess
def prepareGraph(graph: Graph[String, Double]): Graph[Double,
  Double] = {
  // compute a map of normalization factors
  val outgoingEdgesSum = graph.aggregateMessages[Double]{
    ctx => ctx.sendToSrc(ctx.attr),
    (a, b) => a + b,
    TripletFields.EdgeOnly
  }.collect.toMap
  // normalize edges
  val normEdges = graph.edges.map(e => {
    val srcSum = outgoingEdgesSum.getOrElse(e.srcId, 1.0)
    val newWeight = e.attr / srcSum
    Edge(e.srcId, e.dstId, newWeight)
  })
  val initialGuess = 1.0
  val vertices = graph.vertices
  // assign each vertex an initial guess
  Graph(vertices, normEdges).mapVertices((_, attr) =>
    initialGuess)
}
```

4. Definiamo la UDF del modulo Preleg per il PageRank. L'UDF *vertexProgram* implementa la formula PageRank data in input la somma dei contributi normalizzati di PageRank delle pagine in-neighbors. L'UDF *sendMessage* codifica il modo in cui una pagina condivide il suo PageRank tra i suoi out-neighbors. L'UDF *messageCombiner* implementa la somma dei contributi che saranno utilizzati in *vertexProgram* per aggiornare il rank della pagina che riceve tali contributi

```
val d = 0.85
val numVertices = preparedGraph.numVertices
def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double =
  (1 - d) / numVertices + d * msgSum
def sendMessage(edge: EdgeTriplet[Double, Double]): Iterator[(VertexId, Double)] =
  Iterator((edge.dstId, edge.srcAttr * edge.attr))
def messageCombiner(a: Double, b: Double): Double = a + b
```

5. Definiamo com'è costruita la somma. La funzione *buildSummary* riceve *sentenceGraph* e *rankGraph* per creare un riepilogo. *PhraseGraph* è il grafo di input per l'operatore Pregel, mentre *rankGraph* è il risultato del calcolo PageRank BSP. Il riepilogo è composto dalle prime k frasi per PageRank, ordinate in base a come appaiono nel testo originale. L'ordinamento originale viene mantenuto sfruttando l'ID nodo nel grafico delle frasi

```
def buildSummary(sentenceGraph: Graph[String, Double], rankGraph: Graph[Double, Double], k: Int): String = {
    // select the top-k sentences by PageRank, ordered according to
    // how they appear in the original text
    sentenceGraph.vertices.join(rankGraph.vertices)
        .map { case (id, (sent, rank)) => (id, sent, rank) }
        .top(k)(Ordering.by(_.-_3)).sortBy(_.-_1)
        .map { case (_, sent, _) => sent }
        .mkString("\n") + "."
}
```

6. Inseriamo tutto nel metodo main. Costruisci il grafo dal file di testo di input. Prepara il grafo per il calcolo del PageRank ponderato. Definisci l'UDF di Pregel per PageRank. Esegui l'operatore Pregel per 50 iterazioni. Alla fine del calcolo BSP, viene restituito un *rankGraph*, in cui a ogni nodo viene assegnato il suo PageRank. Utilizza *rankGraph* e il grafico originale nella funzione *buildSummary* per ottenere il riassunto finale dalle prime 3 frasi.

```

def main(args: Array[String]) = {
    // build sentence graph
    val path = "src/main/scala/graphX_apps/ANN.txt"
    val sentenceGraph = buildGraph(path)
    // prepare graph for PageRank computation
    val preparedGraph = prepareGraph(sentenceGraph)
    // Define Pregel's UDFs for PageRank
    val d = 0.85
    val numVertices = preparedGraph.numVertices
    def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double =
        (1 - d) / numVertices + d * msgSum
    def sendMessage(edge: EdgeTriplet[Double, Double]): Iterator[(VertexId, Double)] =
        Iterator((edge.dstId, edge.srcAttr * edge.attr))
    def messageCombiner(a: Double, b: Double): Double = a + b
    // Execute Pregel for a fixed number of iterations.
    val rankGraph = Pregel(graph = preparedGraph, initialMsg = 0.0,
                           maxIterations = 50)(vprog = vertexProgram, sendMsg =
                           sendMessage, mergeMsg = messageCombiner)
    // Extract summary
    val top_k = 3
    val summary = buildSummary(sentenceGraph, rankGraph, top_k)
    println("Summary:\n" + summary)
}

```

TOOLS: STORM



Apache Storm è un sistema di elaborazione distribuito in tempo reale che consente di elaborare flussi di dati illimitati in modo affidabile. Prima di Storm, i sistemi di elaborazione in tempo reale venivano sviluppati utilizzando *queues* per scrivere dati e *worker* per leggere ed elaborare tali dati. La maggior parte della logica dell'applicazione aveva a che fare con dove inviare/ricevere messaggi, come serializzare/deserializzare messaggi e assicurarsi che le code e i worker

fossero sempre attivi.

Storm si è dimostrato estremamente scalabile, facile da usare e in grado di elaborare dati con bassa latenza.

Storm fornisce un **livello medio di astrazione** poiché i programmatori possono facilmente definire un'applicazione utilizzando astrazioni di base (ad esempio, spout, stream, bolt e topologie) e testarla in modalità locale senza doverla eseguire su un cluster.

È scritto in **Clojure**, un dialetto di Lisp, ma fornisce API anche in Java.

Diversi linguaggi di programmazione sono supportati tramite il protocollo multi-linguaggio che consente l'implementazione di spout e bolt con altri linguaggi, incluso Python.

Il suo runtime supporta il **parallelismo dei dati** quando molti thread eseguono in parallelo lo stesso codice su blocchi diversi e il **parallelismo delle attività** quando diversi spout e bolt vengono eseguiti in parallelo.

DATI E ASTRAZIONE DELL'ELABORAZIONE

Il paradigma di programmazione offerto da Storm si basa su 5 astrazione per i dati e l'elaborazione

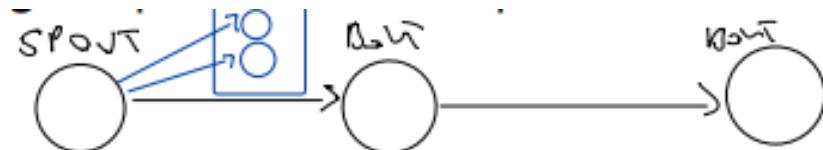
1. **Tupla:** è l'unità di base dei dati che può essere elaborata. Una tupla è composta da un elenco di campi di vari tipi (ad esempio, byte, char, integer, long).
2. **Stream:** rappresenta una sequenza illimitata di tuple, che viene creata o elaborata in parallelo. Gli stream possono essere creati utilizzando serializzatori standard (interi, double) o personalizzati.
3. **Spout:** è la fonte dati di un flusso. I dati possono essere letti da diverse fonti esterne, come API di social network, reti di sensori e sistemi di accodamento (ad esempio, Java Message Service, Kafka, Redis). Quindi, vengono immessi nell' applicazione.
4. **Bolt:** rappresenta l'entità di elaborazione. In particolare, può eseguire qualsiasi tipo di attività o algoritmo (ad esempio, pulizia dei dati, join, query).

5. **Topologia:** rappresenta un job. Una topologia generica è configurata come un DAG, in cui spout e bolt rappresentano i vertici del grafo e i flussi fungono da loro archi. Può essere eseguito all'infinito finché non viene arrestato.

STREAM E RAGGRUPPAMENTO

Gli stream vengono creati ed elaborati in parallelo. Uno spout crea nuovi flussi, mentre un bolt accetta i flussi come input e può generare flussi come output. Parte della creazione di una topologia è definire quali flussi ogni bolt dovrebbe ricevere come input tramite **un raggruppamento di flussi** che determina come il flusso dovrebbe essere suddiviso tra i bolt:

- **raggruppamento shuffle:** le tuple vengono suddivise casualmente tra i bolt in modo che ogni bolt riceva una uguale quantità di tuple.
- **raggruppamento di campi:** le tuple vengono suddivise in base al campo fornito nel raggruppamento (ad esempio, le tuple con lo stesso campo vengono assegnate alla stessa attività, mentre le tuple con campi diversi possono essere elaborate da attività diverse).
- **raggruppamento diretto:** il produttore della tupla determina quale attività del consumatore otterrà questa tupla

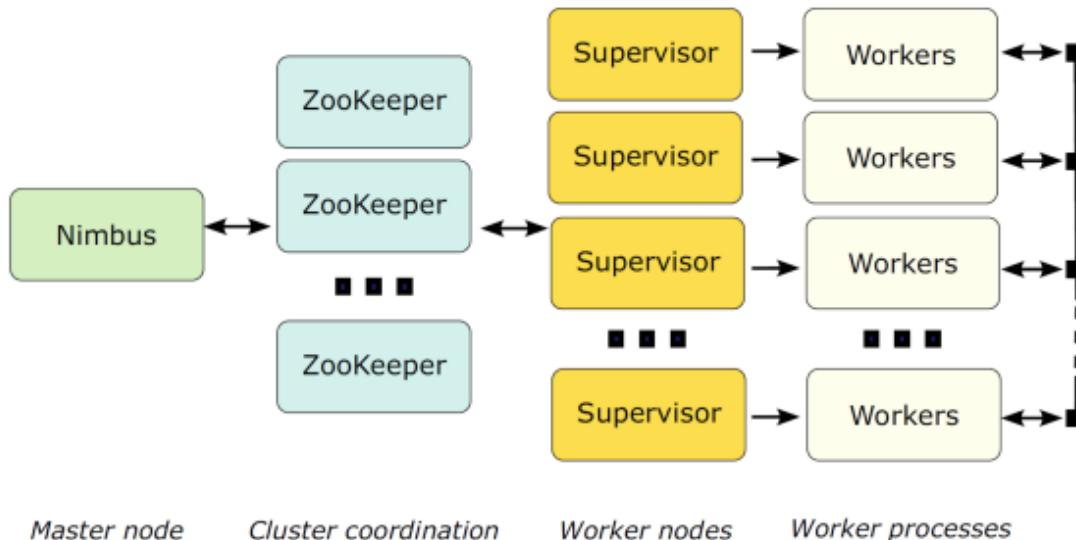


Storm assicura un'elaborazione affidabile dei messaggi utilizzando una serie di task (tracker task) per tracciare il DAG di tuple generate dagli spout. Questi task riconoscono lo spout che ha prodotto la tupla. Ogni tracker utilizza una mappa che collega un ID tupla spout a una coppia di valori, ovvero l'ID task che genera la tupla spout, che viene utilizzato per il riconoscimento, e un intero a 64 bit, chiamato "ack val", che rappresenta lo stato complessivo dell'albero di tuple (derivato dall'XOR di tutti gli ID tupla nell'albero che sono stati generati e/o riconosciuti).

Quando un task acker verifica che un "ack val" sia diventato zero, sa che l'albero di tuple è completato. Questo meccanismo, noto come elaborazione "

almeno una volta", assicura una semantica di elaborazione senza stato garantendo che tutti i messaggi verranno elaborati, sebbene alcuni possano essere elaborati più volte in caso di guasti del sistema.

ARCHITETTURA



Storm è progettato per essere un sistema fault-tolerant:

- Se un worker muore, il Supervisor lo riavvia semplicemente;
- Se non può essere riavviato (ad esempio, gli heartbeat di Nimbus falliscono), il worker verrà riassegnato a un'altra macchina da Nimbus.

I demoni Nimbus e Supervisor sono progettati per essere stateless, ovvero tutti gli stati vengono mantenuti in un componente chiamato **ZooKeeper**.

- Se i Nimbus o Supervisor muoiono, si riavviano semplicemente e i worker non sono interessati dal loro guasto. In questo modo, senza Nimbus, i worker continueranno comunque a lavorare e i Supervisor continueranno a riavviarli se necessario.
- Tuttavia, in assenza di Nimbus, i worker non verranno automaticamente riassegnati a macchine alternative quando necessario.
- La proprietà stateless di Nimbus e Supervisors è garantita da ZooKeeper, che coordina, condivide le informazioni di configurazione tra le applicazioni con

tecniche di sincronizzazione robuste e memorizza tutti gli stati associati al cluster e alle attività.

Per riassumere, un cluster Storm dovrebbe avere **un nodo Nimbus, più Supervisors** e un'istanza **ZooKeeper per ogni nodo**. L'intero flusso di un'applicazione Storm segue questi passaggi:

1. Il Nimbus attende l'invio della topologia.
2. Quando viene inviata, il Nimbus raccoglierà tutti i task della topologia e il loro ordine di esecuzione e distribuirà le attività tra tutti i Supervisor in esecuzione.
3. Tutti i Supervisor inviano heartbeat al Nimbus a intervalli regolari per confermare che sono ancora attivi. Qualsiasi errore (ad esempio, Supervisori o Nimbus) non influirà sull'applicazione in esecuzione.
4. Quando tutte le attività sono state eseguite, il supervisor attenderà una nuova attività dal Nimbus.
5. Quando tutte le topologie sono state elaborate, il Nimbus attende che una nuova topologia venga inviata dall'utente

BASI DI PROGRAMMAZIONE

Una tipica applicazione Storm consiste di 3 componenti principali

- **spout**, che implementa l'interfaccia **IRichSpout** per fornire un'implementazione personalizzata del metodo *nextTuple()*.
- **bolt**, che implementa l'interfaccia **IRichBolt** per fornire un'implementazione personalizzata del metodo *execute()*.
- **main**, che definisce la topologia e i nodi del DAG utilizzando i metodi **setSpout** e **setBolt**

SPOUT

La classe Spout include i seguenti metodi sovrascrivibili:

- **void open():** chiamato quando un task viene inizializzato all'interno di un worker sul cluster.

- `void nextTuple()`: utilizzato per emettere tuple nella topologia tramite un collector. Questo metodo dovrebbe essere non bloccante: se lo spout non ha tuple da emettere, dovrebbe return.
- `void ack()` e `void fail()`: chiamati quando Storm rileva che una tupla emessa dallo spout viene completata correttamente o non riesce a essere completata.
- `void declareOutputFields()`: dichiara lo schema di output per tutti i flussi della topologia.

BOLT

La classe Bolt include i seguenti metodi sovrascrivibili:

- `void prepare()`: fornisce al bolt una collezione in output che viene utilizzato per emettere tuple.
- `void execute()`: riceve una tupla da uno degli input del bolt e applica la logica di elaborazione.
- `void cleanup()`: chiamato quando un bolt viene arrestato e dovrebbe ripulire tutte le risorse che sono state aperte.
- `void declarationOutputFields()`: dichiara lo schema di output per tutti i flussi della topologia.

COLLECTOR

L'oggetto collector, utilizzato sia da spout che da bolt, espone l'API per emettere tuple nella topologia. La differenza principale tra il collector per l'interfaccia IRichSpout, vale a dire SpoutOutputCollector, e OutputCollector per IRichBolt, è che gli spout possono contrassegnare i messaggi con ID in modo che possano essere riscontrati o falliti

ESEMPIO DI APPLICAZIONE

L'esempio seguente analizza un flusso di tweet per estrarre il numero di occorrenze di ogni hashtag. La topologia è composta da uno spout e due bolt:

- **TweetSpout** è l'unica fonte di dati. Questo spout si connette a Twitter tramite le sue API ed emette nella topologia un flusso di tweet generati a diversi timestamp.

- **SplitHashtag** riceve le tuple dallo spout ed esegue la pre-elaborazione. In particolare, estraе gli hashtag da ogni tweet e li emette nella topologia per l'elaborazione da parte del bolt successivo.
- **HashtagCounter** esegue l'operazione di conteggio utilizzando una mappa interna come struttura dati e stampa il numero corrente di occorrenze di ogni hashtag incontrato nel flusso di tuple.

```

public class TweetSpout extends BaseRichSpout {
    public static final String consumerKey = "...";
    public static final String consumerSecret = "...";
    public static final String accessToken = "...";
    public static final String accessTokenSecret = "...";

    private SpoutOutputCollector collector;
    private TwitterStream ts;
    private LinkedBlockingQueue queue;
    public void nextTuple() {
        Object s = queue .poll();
        if(s == null) {
            Utils.sleep(50);
        } else {
            collector.emit(new Values(s));
        }
    }

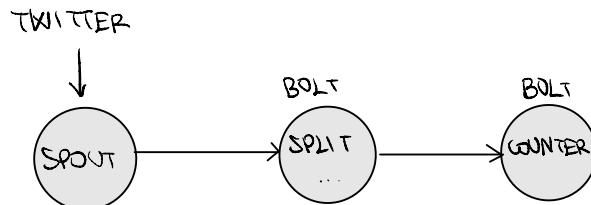
    public void declareOutputFields(OutputFieldsDeclarer
        outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("tweet"));
    }
}

```

Listing 4.9: Tweet Spout.

→ permette una coppia che ha come chiave tweet e come valore il valore

cosa fa l'applicazione:



Quando nel main istanziamo oggetti dobbiamo dire quante copie vogliamo istanziare.

```

public class SplitHashtag extends BaseRichBolt {
    private OutputCollector _collector;

    @Override
    public void prepare(Map conf, TopologyContext context,
        OutputCollector collector) {
        _collector = collector;
    }

    @Override
    public void execute(Tuple tuple) {
        String tweet = tuple.getString(0);
        StringTokenizer st = new StringTokenizer(tweet);
        while(st.hasMoreElements()) {
            String tmp = (String) st.nextElement();
            if(StringUtil.startsWith(tmp, "#")) {
                _collector.emit(new Values(tmp));
            }
        }
        _collector.ack(tuple);
    }

    @Override
    public void declareOutputFields
        (OutputFieldsDeclarer declarer){
        declarer.declare(new Fields("hashtag"));
    }
}

```

Metodo principale

} UNA Specie di
costruttore

Specifica per ogni
tupla che riceve che
operazioni deve fare.

Tutte le parole che non sono
“hashtag” non vengono
considerate.

\uparrow chiave tupla

Listing 4.10: SplitHashtag bolt.

```

public class HashtagCounter extends BaseRichBolt {
    Map<String, Integer> _counts; → per contare le occorrenze
    OutputCollector _collector;

    @Override
    public void prepare(Map conf, TopologyContext context,
        OutputCollector collector) {
        _counts = new HashMap<String, Integer>();
        _collector = collector;
    }

    @Override
    public void execute(Tuple tuple) {
        String hashtag = tuple.getString(0);
        Integer count = _counts.get(hashtag);
        if (count == null)
            count = 0;
        count++;
        _counts.put(hashtag, count);
        _collector.ack(tuple);
    }
}

```

Per ogni tupla che riceve
estrae l’hashtag e prende dalla
mappa il valore attuale dei
conteggi della mappa. Se
restituisce map vuol dire che
per quella chiave ancora non
c’è valore. Se incontro il valore
lo incremento.

```

@Override
public void cleanup() {
    for(Map.Entry<String, Integer> entry: _counts.
        entrySet()){
        System.out.println(entry.getKey()+" : " + entry.
            getValue());
    }
}

@Override
public void declareOutputFields(OutputFieldsDeclarer
    outputFieldsDeclarer) {
    outputFieldsDeclarer.declare(new Fields("hashtag"));
}

```

Viene invocato quando l'applicazione termina.

NON SERVE
ma non da errore in fase di compilazione

Listing 4.11: HashtagCounter bolt.

```

public class HashtagCountTopology {
    public static void main(String[] args) throws Exception{

        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new TweetSpout, 3);
        builder.setBolt("split", new SplitHashtag(), 3).
            shuffleGrouping("spout");
        builder.setBolt("count", new HashtagCounter()).
            fieldsGrouping("split", new Fields("hashtag"));
        Config conf = new Config();
        String topologyName = "hashtagCounter";
        StormSubmitter.submitTopology(topologyName, conf,
            builder.createTopology());
    }
}

```

Listing 4.12: Topology.

TOOLS: APACHE AIRFLOW



Apache Airflow è una piattaforma open source progettata per sviluppare, programmare e monitorare i workflow; progetto di primo livello della Apache

Software Foundation dal 2019.

Può essere utilizzata per creare applicazioni di elaborazione dati come DAG di attività. I DAG sono definiti come codice Python, che offre la possibilità di:

- memorizzare i flussi di lavoro nel controllo delle versioni e di ripristinare le versioni precedenti
- sviluppare flussi di lavoro da più persone contemporaneamente
- scrivere test per convalidare le funzionalità

Airflow presenta un alto livello di astrazione poiché i programmatore possono facilmente creare flussi di lavoro combinando un set di attività e specificando dipendenze tra di esse.

Lo scheduler Airflow esegue le attività su una matrice di worker tenendo in considerazione le dipendenze specificate dal DAG. Il runtime supporta entrambi:

- parallelismo dei dati quando molte attività eseguono in parallelo lo stesso codice su diversi blocchi di dati
- parallelismo dei task quando diverse attività (o operatori) vengono eseguite in parallelo

ARCHITETTURA

I principi sui cui si fonda Airflow sono:

- **Dinamicità:** le pipeline di Airflow sono configurate come codice Python, consentendo la generazione dinamica di pipeline.
- **Estensibilità:** Airflow contiene operatori per connettersi a molte tecnologie e tutti i componenti sono estensibili per adattarsi facilmente agli ambienti degli utenti.
- **Flessibilità:** Airflow facilita la parametrizzazione dei flussi di lavoro utilizzando il motore di template Jinja*.

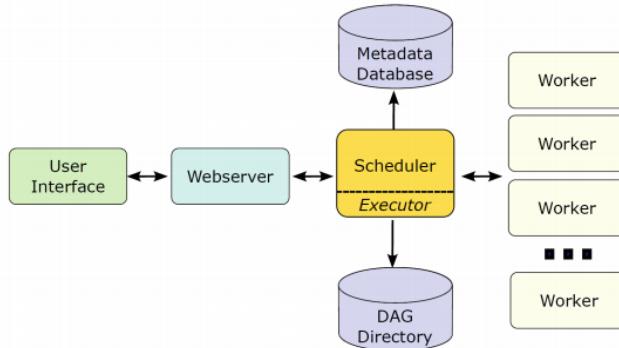


Fig. 4.12: Architecture of an Airflow installation.

- **Scheduler:** gestisce sia l'attivazione dei flussi di lavoro pianificati sia l'invio di attività all'Executor.
- **Executor:** gestisce l'esecuzione delle attività. Gli Executor a livello di produzione possono inviare l'esecuzione delle attività a un set di Worker.
- **Webserver:** fornisce un'interfaccia utente per ispezionare, attivare e correggere il comportamento di DAG e attività.
- **Directory DAG:** contiene i file DAG letti da Scheduler, Executor e Worker.
- **Database metadati:** utilizzato da Scheduler, Executor e Webserver per memorizzare lo stato

BASI DI PROGRAMMAZIONE

Un DAG definisce le dipendenze delle attività, il loro ordine di esecuzione e il comportamento di ripetizione in Airflow. Le attività stesse descrivono le azioni da intraprendere, che si tratti di recuperare dati, eseguire analisi, attivare altri sistemi o eseguire altre operazioni.

L'esempio seguente definisce un DAG denominato "backup giornaliero", che inizia il 1° gennaio 2023 ed è eseguito una volta al giorno. Sono definite quattro attività, tutte che eseguono uno script bash diverso. Una relazione tra le attività è indicata dal simbolo `>>` per stabilire una dipendenza e determinare la sequenza di esecuzione delle attività.

```

# DAG declaration
with DAG(dag_id="daily_backup", start_date=datetime(2023, 1, 1),
         schedule="0 0 * * *") as dag:

    # Definition of four tasks performing bash commands
    task_A = BashOperator(task_id="task_A", bash_command="mv /
        backup/*.tgz /backup/old")
    task_B = BashOperator(task_id="task_B", bash_command="tar czf /
        backup/http_log.tgz /var/log/http")
    task_C = BashOperator(task_id="task_C", bash_command="tar czf /
        backup/mail_log.tgz /var/log/mail")
    task_D = BashOperator(task_id="task_D", bash_command="echo
        backup completed")

    # Definition of task dependencies
    task_A >> [task_B, task_C]
    [task_B, task_C] >> task_D

```

Listing 4.13: Example of DAG declaration.

TIPOLOGIE DI TASK

- **Operatori:** task predefiniti che i programmati possono collegare insieme rapidamente per creare la maggior parte delle parti di un DAG.
- **Sensori:** una sottoclasse speciale di operatori specificamente progettata per il solo scopo di attendere che si verifichi un evento esterno.
- **Attività TaskFlow:** funzioni Python personalizzate confezionate come task

OPERATORI

Un Operatore è concettualmente un modello per un'attività predefinita, che può essere definita in modo dichiarativo all'interno di un DAG. Airflow offre un set completo di operatori disponibili, con alcuni integrati nel core o provider preinstallati. Alcuni operatori popolari del core includono:

- BashOperator: esegue un comando bash
- PythonOperator: richiama una funzione Python arbitraria
- EmailOperator: invia un'e-mail

Di seguito viene presentato un esempio di come utilizzare gli operatori. Un SimpleHttpOperator esegue una richiesta GET verso l'endpoint specificato, utilizzando una stringa di query specificata, per ottenere informazioni sulle condizioni meteorologiche attuali nella città di Cosenza, Italia. Un PythonOperator prende la risposta HTTP ottenuta da SimpleHttpOperator e la formatta correttamente per creare il corpo di un messaggio, che viene quindi inserito in un'e-mail inviata da

EmailOperator

```

endpoint = "https://api.open-meteo.com/v1/forecast"
latitude = 39.30
longitude = 16.25
parameters = ["temperature_2m_max", "temperature_2m_min",
              "precipitation_sum", "sunrise", "sunset", "windspeed_10m_max",
              "winddirection_10m_dominant"]
timezone = "Europe/Berlin"
today = pendulum.now().strftime("%Y-%m-%d")
weather_query = f"{endpoint}?latitude={latitude}&longitude={longitude}&daily={', '.join(parameters)}&timezone={timezone}&start_date={today}&end_date={today}"

def build_body(**context):
    query_result = context['ti'].xcom_pull('submit_query')
    weather_info = json.loads(query_result)
    daily_info = weather_info["daily"].items()
    units = weather_info["daily_units"].values()
    list_info = [f"[{k}]:{v[0]} {unit}" for (k,v),unit in zip(
        daily_info, units)]
    body_mail = ", ".join(list_info)
    return body_mail

with DAG(dag_id="weather_mail", start_date=datetime(2023, 1, 1),
         schedule="0 0 * * *") as dag_weather:
    submit_query = SimpleHttpOperator(task_id="submit_query",
                                       http_conn_id='', endpoint=weather_query, method="GET",
                                       headers={})
    prepare_email = PythonOperator(task_id='prepare_email',
                                   python_callable=build_body, dag=dag_weather)
    send_email = EmailOperator(task_id="send_email", to="user@example.com", subject="Weather today in Cosenza",
                               html_content="{{ti.xcom_pull('prepare_email')}}")

    submit_query >> prepare_email >> send_email

```

Listing 4.14: Example of Operators.

SENSORI

I sensori sono un tipo speciale di Operatore progettato per attendere che qualcosa si verifichi.

I sensori hanno due diverse modalità di esecuzione:

- **poke**: il sensore occupa uno slot worker per l'intera durata.
- **reschedule**: il sensore occupa uno slot worker solo quando esegue il controllo e dorme per una durata impostata tra i controlli.

Le modalità **poke** e **reschedule** possono essere configurate direttamente quando il sensore viene istanziato. Un'operazione che esegue il controllo ogni secondo dovrebbe essere in modalità **poke**, mentre qualcosa che esegue il controllo ogni minuto dovrebbe essere in modalità **reschedule**.

ATTIVITA' TASKFLOW

Per i programmatore che scrivono la maggior parte dei loro DAG utilizzando codice Python semplice anziché Operatori, l'API TaskFlow semplifica il processo per gli sviluppatori di creare DAG ben organizzati utilizzando il decoratore @task.

L'esempio dalla documentazione di Airflow mostrato successivamente utilizza tre attività: get_ip, compose_email, send_email. Le prime due attività vengono dichiarate utilizzando TaskFlow e passano automaticamente il valore di ritorno di get_ip in compose_email, dichiarando automaticamente che compose_email è a valle di get_ip.

La terza attività è un operatore più tradizionale, ma anche questa può utilizzare il valore di ritorno di compose_email per impostare i suoi parametri e calcolare automaticamente che deve essere a valle di compose_email

```
from airflow.decorators import task
from airflow.operators.email import EmailOperator

@task
def get_ip():
    return my_ip_service.get_main_ip()

@task
def prepare_email(external_ip):
    return {
        'subject': f'Server connected from {external_ip}',
        'body': f'Your server executing Airflow is connected from
                the external IP {external_ip}<br>'
    }

email_info = prepare_email(get_ip())

EmailOperator(
    task_id='send_email',
    to='example@example.com',
    subject=email_info['subject'],
    html_content=email_info['body']
)
```

Listing 4.15: Example showcasing the utilization of TaskFlow APIs.

ESEMPIO DI APPLICAZIONE

Mostriamo come le API TaskFlow di Airflow possono essere utilizzate per implementare un'applicazione di apprendimento di ensemble. Tra i diversi modi in cui è possibile ottenere un modello di ensemble, utilizziamo la tecnica di voto:

- Il set di dati di input viene suddiviso, utilizzando un partizionatore, in un `trainingSet` e `testSet`.
- Il `trainingSet` viene fornito in input a n algoritmi di classificazione che vengono eseguiti in parallelo per creare n modelli di classificazione indipendenti.
- Uno strumento di voto esegue una classificazione di ensemble assegnando a ciascuna istanza del `testSet` la classe prevista dalla maggior parte degli n modelli generati nel passaggio precedente

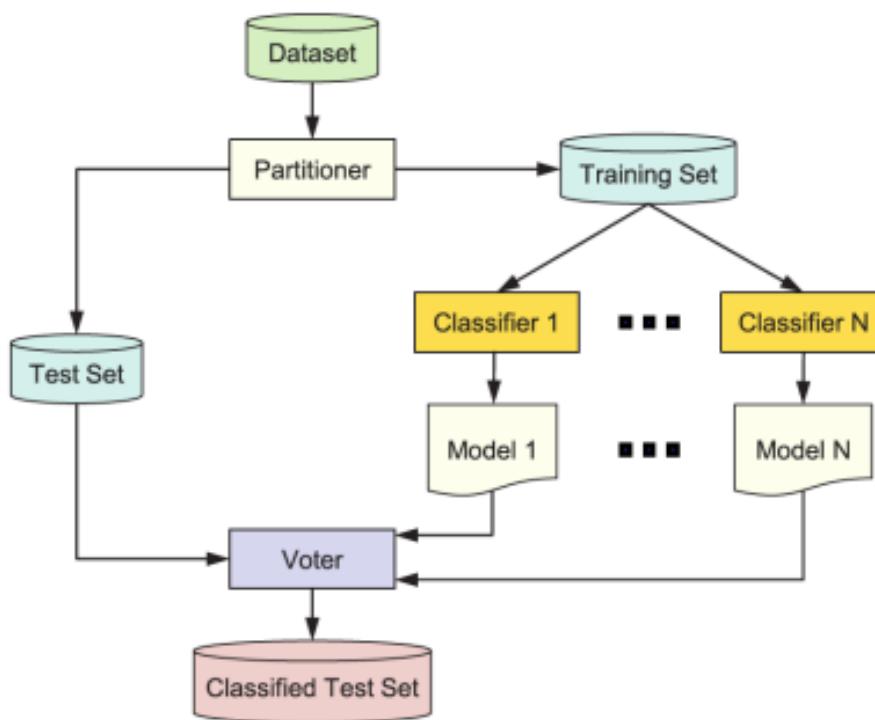


Fig. 4.13: Ensemble learning application.

Il workflow è composto dai seguenti task (funzioni Python):

- **load:** carica il dataset usando la funzione `load_breast_cancer` di `sklearn.datasets`; dati e target vengono restituiti in una tupla.
- **partition:** riceve il dataset caricato dall'attività di caricamento e lo partiziona in train e test set; restituisce i dati partizionati come un dizionario.
- **train:** riceve i dati di training e un'istanza di uno stimatore `sklearn`, ed esegue l'operazione di fitting; il modello adattato viene restituito come output.

- **vote**: riceve i dati di test e un elenco di modelli adattati; utilizza i modelli per calcolare un elenco di previsioni; in seguito, le previsioni vengono aggregate tramite votazione, per ottenere la classificazione dell'ensemble.

```

# instantiate DAG
@dag(
    schedule=None,
    start_date=pendulum.now(),
    catchup=False,
    tags=["example"],
)
def ensemble_taskflow():
    # load and partition dataset into train and test
    @task(multiple_outputs=True)
    def partition():
        X,y = load_dataset(return_X_y=True)
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.2, random_state=13
        )
        train_data = (X_train.tolist(), y_train.tolist())
        test_data = (X_test.tolist(), y_test.tolist())
        return {"train": train_data, "test": test_data}

    # fit a given classification model on train data
    @task
    def train(model:sklearn.base.BaseEstimator, train_data:tuple):
        X_train, y_train = train_data
        # fit the model and serialize it
        model.fit(X_train, y_train)
        model_bytes = pickle.dumps(model)
        model_str = model_bytes.decode("latin1")
        return model_str

    # perform ensemble classification on test data
    @task
    def vote(test_data:tuple, models:list):
        X_test, y_test = test_data
        pred_sum = np.array([0]*len(X_test))
        for model_str in models:
            # deserialize the model and predict
            model_bytes = model_str.encode("latin1")
            model = pickle.loads(model_bytes)
            pred_sum += model.predict(X_test)
        # prediction is set equal to the majority class
        n_models = len(models)
        threshold = np.ceil(n_models/2)
        preds = [int(s>=threshold) for s in pred_sum]
        print(f"Accuracy is: {accuracy_score(y_test, preds):.2f}")

    ### main flow ####
    # load and partition dataset
    partitioned_dataset = partition()
    train_data = partitioned_dataset["train"]
    test_data = partitioned_dataset["test"]
    # train in parallel 5 independent classifiers
    m1 = train(GaussianNB(), train_data)
    m2 = train(LogisticRegression(), train_data)
    m3 = train(DecisionTreeClassifier(), train_data)
    m4 = train(SVC(), train_data)
    m5 = train(KNeighborsClassifier(), train_data)
    # compute voting accuracy on test data
    vote(test_data, [m1, m2, m3, m4, m5])

    # start DAG
    ensemble_taskflow()

```

Listing 4.16: Ensemble learning in Airflow.

TOOLS: HIVE

I sistemi simili a SQL tentano di combinare l'efficacia di Hadoop con la facilità d'uso del linguaggio simile a SQL, per consentire lo sviluppo di applicazioni di analisi dei dati semplici ed efficienti.

- **Apache Hive**, un software di data warehouse basato su Hadoop per la lettura, la scrittura e la gestione dei dati in infrastrutture su larga scala, è uno dei sistemi più utilizzati in questo contesto.
- **Apache Pig** è un altro framework basato su Hadoop che sfrutta un linguaggio simile a SQL per l'esecuzione di applicazioni di flusso di dati in infrastrutture su larga scala.
- **Apache Impala** è un motore di query parallele massicce che fornisce bassa latenza e alta concorrenza per query analitiche su Hadoop, offrendo al contempo un'esperienza simile a RDBMS.



Apache Hive è un sistema di data warehouse basato su Hadoop, che consente agli utenti di scrivere query utilizzando un linguaggio dichiarativo simile a SQL, chiamato **HiveQL**, che vengono poi compilate in job MapReduce ed eseguite su Hadoop. Hive può essere visto come un motore SQL in grado di compilare automaticamente una query simile a SQL in un set di job MapReduce eseguiti su un cluster Hadoop, con funzionalità aggiuntive per la gestione di dati e metadati. Le ragioni alla base dello sviluppo di Hive si basano sul fatto che sebbene MapReduce sia un paradigma molto flessibile, è di livello troppo basso per le attività di analisi dei dati di routine.

Hive supporta una vasta porzione dello standard SQL, oltre a diverse estensioni progettate per semplificare le interazioni con la piattaforma Hadoop sottostante. A

differenza dei database tradizionali, Hive non richiede di definire la struttura della tabella prima di importare i dati, perché consente di proiettare la struttura tabulare nei dati sottostanti durante l'esecuzione della query.

Questa funzionalità è nota come

schema-on-read, il che significa che i dati vengono controllati rispetto allo schema quando viene eseguita una query su di essi.

Hive consente l'esecuzione della stessa query su diverse porzioni di dati, supportando il **parallelismo dei dati**. Fornisce inoltre un alto livello di astrazione perché un programmatore può sviluppare un'applicazione di elaborazione dati utilizzando HiveQL, che si basa su concetti tradizionali di database relazionali. Apache Hive è comunemente utilizzato dagli analisti di dati per query di dati e report su grandi set di dati.

CONCETTI PRINCIPALI E ARCHITETTURA

Hive offre una gamma completa di operazioni di **Data Definition Language (DDL)** e **Data Manipulation Language (DML)**:

- Creazione, modifica, esplorazione ed eliminazione di tabelle.
- Caricamento, inserimento, aggiornamento, eliminazione e unione di dati sul file system.

Le astrazioni utilizzate dal linguaggio HiveQL si basano sui concetti tradizionali di database relazionali (tabella, riga, colonna). Queste capacità fanno sì che Hive agisca come un adattatore tra la piattaforma Hadoop e il vasto ecosistema di strumenti di analisi dei dati basati su database relazionali.

Hive è specificamente progettato per l'elaborazione analitica online (OLAP) piuttosto che per l'elaborazione delle transazioni online (OLTP).

A differenza di SQL Server, Hive non fornisce accesso in tempo reale ai dati. Hive fornisce tre diversi tipi di funzioni per la manipolazione dei dati:

- funzioni definite dall'utente (UDF)
- funzioni aggregate definite dall'utente (UDAF)
- funzioni di generazione di tabelle definite dall'utente (UDTF)

Tali funzioni semplificano notevolmente la scrittura di funzioni personalizzate in diversi linguaggi.

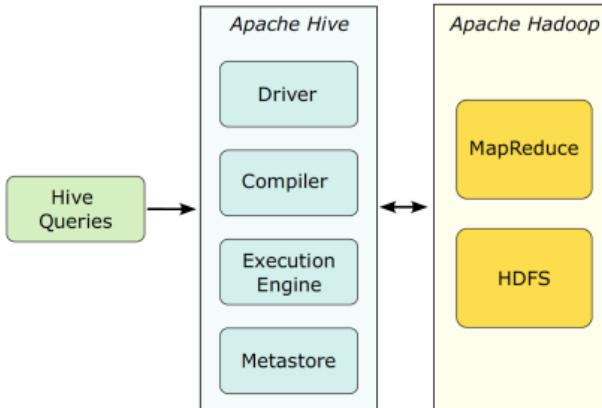


Fig. 4.17. The architecture of Apache Hive.

- **Interfaccia utente (UI):** funge da punto di ingresso per gli utenti per inviare query ed eseguire altre operazioni all'interno del sistema tramite un'interfaccia utente Web o un'interfaccia a riga di comando (CLI).
- **Driver:** questo componente riceve query e implementa handle di sessione, fornendo API di esecuzione e recupero modellate sulle interfacce JDBC/ODBC.
- **Compilatore:** è responsabile dell'analisi delle query, dell'esecuzione di analisi semantiche sulle espressioni di query e della generazione di un piano di esecuzione. Converte la query in un albero sintattico astratto (AST) e quindi, dopo aver verificato la presenza di errori di compilazione, l'AST viene convertito in un DAG. Il compilatore utilizza metadati di tabella e partizione ottenuti da Metastore.
- **Metastore:** utilizza un RDBMS per archiviare informazioni strutturali su varie tavole e partizioni nello store, come i metadati di entità relazionali persistenti e come sono mappate su HDFS. Queste informazioni includono anche dettagli di colonna, tipi di colonna, serializzatori/deserializzatori per la lettura/scrittura dei dati e le posizioni dei file HDFS corrispondenti.
- **Motore di esecuzione:** esegue il piano di esecuzione generato dal compilatore sotto forma di un DAG. Il motore di esecuzione gestisce le dipendenze tra le fasi del DAG e le esegue sui componenti di sistema appropriati.

- **HDFS:** è il file system distribuito sottostante utilizzato per l'archiviazione dei dati in Hadoop.

FLUSSO DI ESECUZIONE

L'esecuzione di una query HiveQL segue gli step sottostanti:

1. L'interfaccia utente avvia l'interfaccia di esecuzione al driver, che crea un handle di sessione per la query e lo invia al compilatore per la generazione di un piano di esecuzione.
2. Il compilatore ottiene i metadati necessari dal Metastore, utilizzandoli per le espressioni di controllo del tipo e applicando un set di ottimizzazioni all'AST (ad esempio, potatura delle partizioni in base ai predicati della query).
3. L'albero degli operatori viene quindi convertito in un DAG di più processi MapReduce, che vengono inviati al motore MapReduce sottostante per la valutazione.
4. Una libreria di serializzazione-deserializzazione, denominata SerDe, viene utilizzata per serializzare e deserializzare i dati per un formato di file specifico quando gli utenti scrivono file su HDFS.
5. Dopo che tutti i processi MapReduce sono stati completati, il driver restituirà i risultati della query all'utente.

BASI DI PROGRAMMAZIONE

Hive supporta le comuni operazioni DDL e DML tramite HiveQL. La DDL è una sintassi per creare e modificare oggetti di database, come tabelle e indici. Ad esempio, per creare una tabella, i programmatori possono utilizzare la seguente istruzione:

```
CREATE [REMOTE] (SCHEMA|DATABASE) [IF NOT EXISTS] database_name
    [LOCATION hdfs_path] [ROW FORMAT row_format] [FIELDS
        TERMINATED BY char];
```

Hive conserva lo schema delle tabelle nel Metastore, che viene utilizzato per archiviare tutte le informazioni su tabelle e partizioni. Il Metastore predefinito è il database Apache Derby, che può essere espresso in modo esplicito tramite l'opzione SCHEMA (o

DATABASE).

Con una sintassi simile, Hive consente all'utente di modificare e rimuovere righe o eliminare un'intera tabella (SHOW, ALTER, DESCRIBE, TRUNCATE, DELETE,...).

Istruzioni DML HiveQL: DML è una sintassi utilizzata per inserire, eliminare e aggiornare i dati in un database. Ad esempio, per caricare i dati in una tabella, i programmati possano utilizzare la seguente istruzione:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE  
tablename [PARTITION (partcol1=val1, partcol2=val2,"...")];
```

Quando si importano dati in tabelle, Hive non esegue alcuna trasformazione, mentre le operazioni di caricamento sono solo operazioni di copia/spostamento che spostano i dati nelle posizioni della tabella Hive. I risultati di una query possono essere inseriti in tabelle utilizzando una sintassi simile a quella precedente. L'opzione OVERWRITE sovrascriverà tutti i dati esistenti nella tabella o nella partizione. Altri modi per modificare i dati in Hive sono UPDATE e DELETE.

```
INSERT OVERWRITE TABLE tablename [IF NOT EXISTS]  
select_statement FROM from_statement;
```

ESEMPIO DI APPLICAZIONE

L'esempio seguente mostra come Hive può essere utilizzato per archiviare dati sulle valutazioni fornite dagli utenti a un set di film e per eseguire alcune query utilizzando HiveQL. Come primo passaggio, dobbiamo creare una tabella in cui verranno archiviati i dati, specificandone lo schema, ovvero quattro colonne denominate *userid*, *movieid*, *rating* e *timestamp*

```
CREATE TABLE data (  
    userid INT,  
    movieid INT,  
    rating INT,  
    timestamp DATE)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY '\t';
```

Listing 4.21: Creating a Hive table.

Una volta creata la tabella, i dati vengono recuperati da un file di testo archiviato nel file system locale (o HDFS), sovrascrivendo qualsiasi contenuto della tabella.

```
LOAD DATA LOCAL INPATH '<path>/data'
OVERWRITE INTO TABLE data;
```

Listing 4.22: Loading data into a Hive table.

Possiamo eseguire alcune analisi statiche

```
SELECT COUNT(*)
FROM data;
```

Listing 4.23: Counting rows.

Utilizzando le clausole SQL tradizionali, come SELECT, FROM, GROUP BY e ORDER BY, è possibile eseguire alcune semplici query, come trovare i film più apprezzati.

```
SELECT movieid, COUNT(rating) AS num_ratings
FROM data
GROUP BY movieid
ORDER BY num_ratings DESC
```

Listing 4.24: Most highly rated movies.

Hive consente agli utenti di eseguire analisi di dati più complesse utilizzando altri linguaggi di programmazione. Ad esempio, possiamo definire uno script Python, denominato week_mapper.py, che esegue alcuni passaggi di elaborazione sulle righe. Questo script mappa il timestamp di ogni recensione di film alla settimana dell'anno in cui è stato prodotto e aggrega i risultati. La funzione *isocalendar()* restituisce una tripla con (anno, settimana, giorno); pertanto, *isocalendar()[1]* restituisce il numero della settimana.

```
import sys
import datetime

for line in sys.stdin:
    line = line.strip()
    userid, movieid, rating, timestamp = line.split('\t')
    week = datetime.datetime.fromtimestamp(float(timestamp)).isocalendar()[1]
    print '\t'.join([userid, movieid, rating, str(week)])
```

Listing 4.25: Python script.

Lo script Python può essere facilmente incorporato nelle istruzioni HiveQL e utilizzato come una funzione tramite la clausola TRANSFORM. Per impostazione predefinita, le colonne saranno trasformate in stringhe e delimitate da tabulazioni prima di essere inviate allo script utente. Infine, viene eseguita una query COUNT raggruppando i nuovi dati per settimana dell'anno

```
add FILE week_mapper.py;

INSERT OVERWRITE TABLE data_new
SELECT TRANSFORM
    (userid, movieid, rating, timestamp) USING
    'python week_mapper.py'
    AS (userid, movieid, rating, week)
FROM data;

SELECT week, COUNT(*)
FROM data_new
GROUP BY week;
```

Listing 4.26: Hive query.

TOOLS: PIG



Apache Pig è un framework di flusso di dati di alto livello per l'esecuzione di programmi MapReduce su Hadoop utilizzando un linguaggio simile a SQL. Pig è stato proposto per colmare il divario tra le query dichiarative di alto livello di SQL e lo stile procedurale di basso livello del modello di programmazione MapReduce. Le query vengono scritte utilizzando un linguaggio personalizzato, denominato

Pig Latin, e vengono quindi convertite in piani di esecuzione che vengono eseguiti come job MapReduce su Hadoop.

Il sistema di programmazione Pig consente di comporre operazioni di manipolazione dei dati di alto livello utilizzando uno stile simile a SQL (ad esempio, operazioni di programmazione parallela, come FOREACH, FLATTEN e COGROUP) mantenendo le caratteristiche principali, i tipi di dati e i carichi di lavoro di MapReduce.

Pig sfrutta un sistema di esecuzione multi-query per elaborare un intero script o un batch di istruzioni contemporaneamente. Pertanto, supporta sia **il parallelismo dei dati**, che viene sfruttato suddividendo i dati in blocchi ed elaborandoli in parallelo, sia **il parallelismo delle attività**, quando più query vengono eseguite in parallelo sugli stessi dati.

Pig è comunemente utilizzato per sviluppare query di dati, analisi di dati semplici ed applicazioni di estrazione, trasformazione e caricamento (ETL), raccogliendo dati da diverse fonti, come flussi, HDFS o file.

Le aziende e le organizzazioni che utilizzano Pig in produzione includono LinkedIn, PayPal e Mendeley.

Grazie al linguaggio di scripting Pig Latin, Pig fornisce un livello medio di astrazione, il che significa che, rispetto ad altri sistemi come Hadoop, gli sviluppatori Pig non sono tenuti a scrivere codici complessi e lunghi.

CONCETTI FONDAMENTALI E ARCHITETTURA

Pig fornisce un modello dati nidificato, che consente di gestire dati complessi e non normalizzati. Supporta tipi scalari, come int, long, double, chararray (ad esempio string) e bytearray. Inoltre, fornisce tre modelli dati complessi:

- **mappa**: un array associativo, in cui una stringa è la chiave e il valore può essere di qualsiasi tipo.
- **Tupla**: un elenco ordinato di dati, chiamati anche campi, in cui ogni campo è un pezzo di dati. Gli elementi di una tupla possono essere di qualsiasi tipo, consentendo tipi complessi nidificati.
- **Bag**: una raccolta di tuple, simile a un database relazionale. Le tuple in un bag corrispondono alle righe di una tabella, sebbene, a differenza di una tabella

relazionale, i bag Pig non richiedono che ogni tupla contenga lo stesso numero di campi. Un bag è anche identificato come una **relazione**

Ottimizzazione delle query: ogni script Pig viene tradotto in un set di job MapReduce che vengono automaticamente ottimizzati dal motore Pig utilizzando diverse regole di ottimizzazione, come la riduzione delle istruzioni inutilizzate o l'applicazione di filtri durante il caricamento dei dati.

Questa ottimizzazione può essere logica o fisica:

- Le ottimizzazioni logiche riorganizzano il grafo del flusso di dati logico inviato dall'utente, generando un nuovo grafo che è semanticamente equivalente all'originale ma può essere valutato in modo più efficiente.
- Le ottimizzazioni fisiche riguardano il modo in cui il grafo del flusso di dati logico viene tradotto in un piano di esecuzione fisico, come una serie di processi MapReduce.

Viene creato un piano logico per ogni bag definita dall'utente. Quando il piano logico viene creato, non si verifica alcuna elaborazione, ma inizia solo quando l'utente richiama un comando STORE su una bag. A quel punto, il piano logico per quella bag viene trasformato in un piano fisico, che viene quindi eseguito. Questa esecuzione lazy è vantaggiosa poiché consente il pipelining in memoria e altre ottimizzazioni.

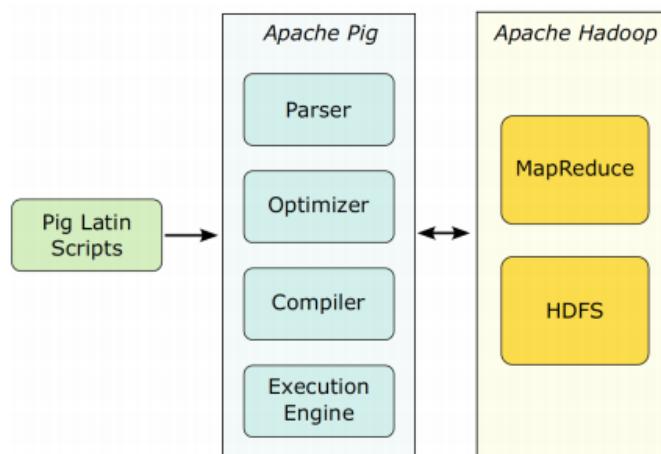


Fig. 4.18. The architecture of Apache Pig.

Pig consiste di 4 componenti principali:

- **Parser**, che gestisce tutte le istruzioni Pig Latin che verificano errori di sintassi e tipi di dati. Produce un DAG come output, che rappresenta gli operatori logici degli script come nodi e il flusso di dati come bordi.
- **Optimizer**, che applica operazioni di ottimizzazione sul DAG prodotto dal parser per migliorare la velocità di query, come split, merge, projection, pushdown, transform e reorder. Ad esempio, pushdown e projection omettono dati o colonne non necessari, riducendo la quantità di dati da elaborare.
- **Compiler**, che genera una sequenza di job MapReduce, a partire dall'output dell'ottimizzatore. Questo processo include altre ottimizzazioni, come la riorganizzazione dell'ordine di esecuzione.
- **Execution engine**, che esegue i job MapReduce generati dal compilatore sul runtime di Hadoop. L'output può essere visualizzato sullo schermo utilizzando il comando DUMP o salvato in HDFS utilizzando la funzione STORE.

BASI DI PROGRAMMAZIONE

Le istruzioni Pig Latin sono espresse tramite bag; un bag è una raccolta di tuple che possono essere create:

- utilizzando tipi di dati nativi supportati da Pig, sia tipi semplici (ad esempio, int, long, float, double, chararray e boolean) che tipi complessi (ad esempio, tuple, mappe o bag nidificati),
- oppure caricando dati dal file system.

Esempio di un'istruzione Pig Latin che carica alcuni dati degli studenti (nome ed età):

```
A = LOAD 'file' USING PigStorage() AS (name:chararray, age:int);
```

Altri comandi Pig Latin sono i seguenti:

- FILTER, che seleziona le tuple da una relazione in base a una condizione

```
alias = FILTER alias BY expression;
```

- JOIN (interno o esterno), che esegue un join interno/esterno di due o più relazioni in base a valori di campo comuni

```
alias = JOIN alias | left-alias BY { expression };
```

- FOREACH, che genera trasformazioni di dati in base a colonne di dati. Di solito, l'uso dell'operazione FOREACH è abbinato all'operazione GENERATE, che consente di lavorare con colonne

```
alias = FOREACH { block | nested_block };
```

```
X = FOREACH A GENERATE f1;
```

- STORE che salva il risultato nel file system

```
STORE alias INTO 'directory' [USING function];
```

- Poiché Pig supporta la definizione di UDF da parte del programmatore, consente di registrare un file JAR da utilizzare nello script e di assegnare alias alle UDF

```
REGISTER path;
DEFINE alias {function | ['command' [input] [output] [stderr]]};
```

ESEMPIO DI PROGRAMMAZIONE

L'esempio seguente implementa un analizzatore di sentiment basato su dizionario. Dato un dizionario di parole associate a sentiment positivo o negativo, il sentiment di un testo (ad esempio, una frase, una recensione, un tweet o un commento) viene calcolato sommando i punteggi delle parole positive e negative nel testo e calcolando la valutazione media.

Poiché Pig non fornisce una libreria integrata per l'analisi del sentiment, il sistema sfrutta dizionari esterni per associare le parole ai loro sentiment e determinare l'orientamento semantico delle parole di opinione



Fig. 4.19. Structure of the sentiment analyzer application.

Gli sviluppatori possono includere analisi avanzate in uno script definendo UDF. Ad esempio, l'UDF

PROCESS è finalizzato all'elaborazione di una tupla rimuovendo la punteggiatura come fase di pre-elaborazione. Altre funzionalità possono essere aggiunte al metodo exec, che è implementato in Java. L'UDF definito in Java è registrato con l'alias PROCESS

```

public class Processing extends EvalFunc<String> {
    @Override
    public String exec(Tuple tuple) throws IOException {
        if (tuple == null || tuple.size() == 0 ||
            tuple.get(0) == null)
            return null;
        String str = (String) tuple.get(0);
        // Remove punctuation and, if required, apply
        // lemmatization, stemming and other
        String clean = str.toLowerCase().replaceAll
            ("\\p{Punct}", " ");
        ...
        return clean;
    }
}

```

Listing 4.27: Java UDF processing.

```

REGISTER PigUDF.jar;
DEFINE PROCESS main.Processing;

```

Listing 4.28: Registering a Java UDF.

Successivamente, i dati relativi alle recensioni di testo vengono caricati da HDFS come file CSV

delimitato da tabulazione e, analogamente, viene caricato il dizionario dei sentimenti delle parole.

Quando si caricano dati da un file, l'utente può specificarne lo schema tramite colonne denominate.

```

-- Load data from HDFS
reviews = LOAD 'hdfs://hostname:port/pigdata/reviews.csv' USING
    PigStorage ('\t') AS (id:int, text:chararray);
-- Load the dictionary of word sentiment
dictionary = LOAD 'hdfs://hostname:port/pigdata/dictionary.txt'
    USING PigStorage('\t') AS (word:chararray,rating:int);

```

Listing 4.29: Loading review data and word sentiment dictionary.

Una volta caricati i dati, ogni riga viene tokenizzata ed elaborata. Tramite l'operatore FOREACH, ogni riga dei dati di input viene prima elaborata utilizzando l'UDF precedentemente registrata e poi tokenizzata, producendo un array di token come output. Questo array viene successivamente appiattito tramite l'operatore built-in FLATTEN. L'operatore GENERATE viene utilizzato in collaborazione con FOREACH per produrre come output triple nel formato <review id, text, word>, che vengono archiviate in un bag denominato words.

```
-- Tokenize and process the text of each review
words = FOREACH reviews GENERATE id, text, FLATTEN(TOKENIZE(
    PROCESS(text))) AS word;
```

Listing 4.30: Tokenization and preprocessing.

Il codice identifica innanzitutto tutte le corrispondenze tra le parole di una recensione e le parole del dizionario unendo il bag intermedio creato sopra e le parole nel dizionario. I risultati vengono salvati in un bag denominato matches, che viene poi iterato per assegnare il punteggio a ciascuna parola. Il risultato di questa operazione, salvato in un bag denominato matches_rating, è una tripla nella forma <id recensione, testo, valutazione>

```
-- Join each word with the dictionary and assign a score
sentiment
matches = JOIN words BY word, dictionary BY word;
matches_rating = FOREACH matches GENERATE words::id AS id,
    words::text AS text, dictionary::rating AS rate;
```

Listing 4.31: Joining words with the dictionary and assigning a score sentiment.

La coppia <id recensione, testo> viene utilizzata per eseguire un'operazione di raggruppamento e raccogliere tutte le valutazioni trovate nel dizionario (ad esempio, il bag denominato group_rating).

Dopo il raggruppamento, per ogni recensione, l'operatore AVG incorporato viene utilizzato per aggregare tutte le valutazioni delle parole e la valutazione finale di una recensione viene calcolata come la media dei punteggi dei suoi token.

Infine, il bag di output avg_ratings viene archiviato in un file sul file system HDFS.

```
-- Group and compute the average rating for a review
group_rating = GROUP matches_rating BY(id, text);
avg_ratings = FOREACH group_rating GENERATE group, AVG($1.$2) AS
    rate;
-- Store the results
STORE avg_ratings INTO 'ratings' USING PigStorage(',') '-schema');
```

Listing 4.32: Calculating average rating and storing results to HDFS.

TOOLS: UPC++

Il modello di programmazione Partitioned Global Address Space (PGAS) rappresenta un compromesso tra modelli di programmazione a memoria distribuita e memoria condivisa.

È stato progettato per implementare uno spazio di indirizzamento della memoria globale che è logicamente partizionato in porzioni locali per singoli processi.

Il suo obiettivo principale è limitare lo scambio di dati e isolare i guasti in sistemi su larga scala.

Il modello Asynchronous PGAS (APGAS) è una variante di PGAS che supporta sia la creazione di task asincrone locali che remote.

A differenza di PGAS, non richiede che tutti i processi vengano eseguiti su hardware omogeneo e

supporta la generazione dinamica di più attività. Negli ultimi anni, sono stati proposti diversi linguaggi basati su PGAS, come DASH, X10, Chapel, pPython e UPC.

UPC++ (Zheng et al., 2014) è una libreria basata su C++ che fornisce classi e funzioni volte a supportare la programmazione APGAS.

Oltre ad accedere alla memoria locale, nel modello APGAS ogni thread (indicato come rank) ha accesso a uno spazio di indirizzamento globale allocato in segmenti condivisi che sono distribuiti sui rank.

Questo modello di memoria rende UPC++ adatto per scrivere programmi paralleli che vengono eseguiti in modo efficiente e scalabili su computer paralleli con memoria distribuita composti da centinaia di migliaia di core

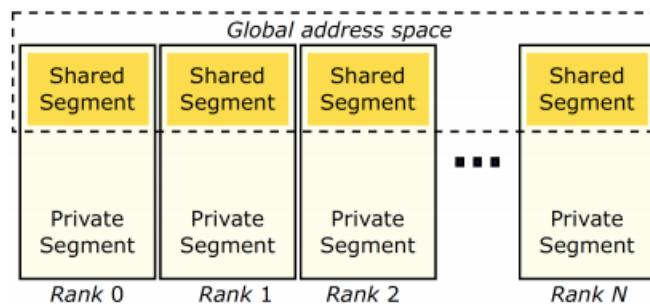


Fig. 4.20. APGAS memory model.

UPC++ utilizza il middleware Global-Address Space Networking (GASNet), un livello indipendente dal linguaggio che fornisce primitive di comunicazione indipendenti dalla rete, tra cui accesso remoto alla memoria (RMA) e messaggi attivi (AM).

In UPC++, tutte le operazioni di accesso remoto alla memoria sono asincrone per impostazione predefinita e le interfacce sono progettate per essere componibili e simili a quelle utilizzate nel C++ convenzionale.

Le principali astrazioni di programmazione in UPC++ comprendono puntatori globali, chiamate di procedura remota (RPC), future e oggetti condivisi, fornendo collettivamente un versatile toolkit per gli sviluppatori per strutturare applicazioni parallele.

UPC++ fornisce

un basso livello di astrazione, consentendo un controllo dettagliato sul parallelismo e l'utilizzo efficiente delle risorse di elaborazione nello sviluppo di applicazioni parallele iterative su larga scala.

Nonostante i suoi vantaggi, UPC++ può riscontrare colli di bottiglia nelle prestazioni a causa dell'uso esteso della comunicazione. Inoltre, l'assenza di costrutti di alto livello contribuisce ad aumentare la verbosità, rendendo necessario ai programmatore di gestire in modo esplicito lo scambio e la sincronizzazione dei dati.

BASI DI PROGRAMMAZIONE

Tutti i programmi UPC++ includono due operazioni fondamentali:

- **upcxx::init()** inizializza il runtime UPC++ e deve essere chiamato prima che vengano utilizzate le funzionalità UPC++.
- **upcxx::finalize()** arresta il runtime UPC++ e impedisce che vengano utilizzate le funzionalità UPC++ dopo di esso.

Un programma UPC++ viene eseguito con un numero fisso di thread, denominati rank, ciascuno dei quali esegue una copia del programma. Ogni rank ha un identificatore associato che va da 0 a N – 1, con N che indica il numero di rank, a cui si può accedere tramite l'operazione **upcxx::rank_me()**.

Il calcolo in un programma UPC++ può essere suddiviso in diversi rank, consentendo loro di eseguire le loro operazioni in parallelo. Il risultato finale viene

quindi raccolto da uno dei rank, il che implica la presenza di un punto di sincronizzazione in cui si attende il risultato di ciascun rank.

Per migliorare il livello di parallelismo, *il calcolo asincrono* viene sfruttato in UPC++, il che consente la sovrapposizione di comunicazione e calcolo, sfruttando attivamente il tempo di attesa. Ciò si ottiene utilizzando *future object*, che sono costrutti UPC++ caratterizzati da un valore e uno stato, che indicano se il valore è disponibile (pronto) o meno

Ad esempio, consideriamo l'operazione **upcxx::allreduce**, definita come segue:

```
template<typename T, typename BinaryOp>
upcxx::future<T> upcxx::allreduce(T &&value, BinaryOp &&op,
                                    upcxx::team &team = upcxx::world());
```

Questa operazione esegue una riduzione globale di un valore di tipo T su tutti i rank applicando una funzione binaria (ad esempio, una somma). È supportato l'uso di team, che sono insiemi ordinati di rank a cui possono essere applicate operazioni collettive. Attualmente, in UPC++, l'unico team supportato è **upxx::world()**, che comprende tutti i rank.

Il tipo di ritorno dell'operazione di riduzione è *future<T>*, che consente un calcolo asincrono. Un rank in attesa di un risultato può eseguire altre operazioni non correlate nel frattempo.

Questa logica è implementata dal metodo **wait()** di **upcxx::future**, che controlla lo stato dell'oggetto futuro, eseguendo un ciclo fino al completamento e alla disponibilità

OGGETTI CONDIVISI

UPC++ consente ai programmatore di lavorare con oggetti condivisi tra diversi rank.

Un oggetto condiviso viene allocato all'interno di un segmento di memoria condivisa ed è accessibile tramite un puntatore globale, definito come **upcxx::global_ptr<T> gptr**.

UPC++ definisce due diverse aree di memoria nello spazio di indirizzamento globale, in cui gli oggetti possono essere allocati tramite uno dei seguenti approcci:

- Utilizzando **upcxx::new_<T>**, un nuovo oggetto di tipo T viene allocato nel segmento condiviso del rank corrente. Ogni rank può fare riferimento a questo oggetto tramite un puntatore globale privato al suo segmento condiviso locale.
- Utilizzando la parola chiave standard C++ new, viene eseguita una tipica allocazione dinamica nella memoria locale privata del rank.

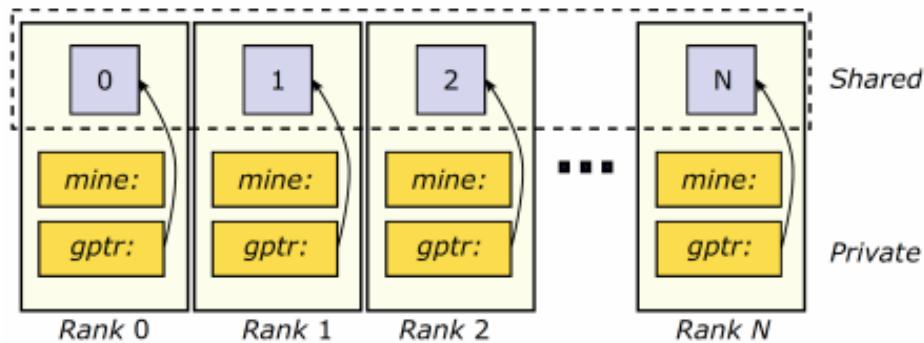


Fig. 4.21. UPC++ global pointers.

ESEMPIO DI PROGRAMMAZIONE

Il metodo Monte Carlo per la stima di π opera attraverso i seguenti passaggi:
Viene generato un gran numero N di punti in un piano 2D, le cui coordinate (x, y) sono variabili casuali uniformemente distribuite nell'intervallo [0, 1].

Tra tutti i punti generati casualmente, il metodo conta quanti di essi rientrano nel settore della circonferenza con raggio unitario centrato in (0, 0), ovvero il numero di punti M che soddisfano la seguente equazione: $x^2 + y^2 \leq 1$

Dato che le aree del quadrato unitario e della sezione sono

$$A_q = 1 \text{ e } A_s = \frac{4}{\pi}$$

il valore di π può essere stimato come segue:

$$\frac{M}{N} = \frac{A_s}{A_q} = \frac{\pi}{4} \rightarrow \pi = \frac{4M}{N}$$

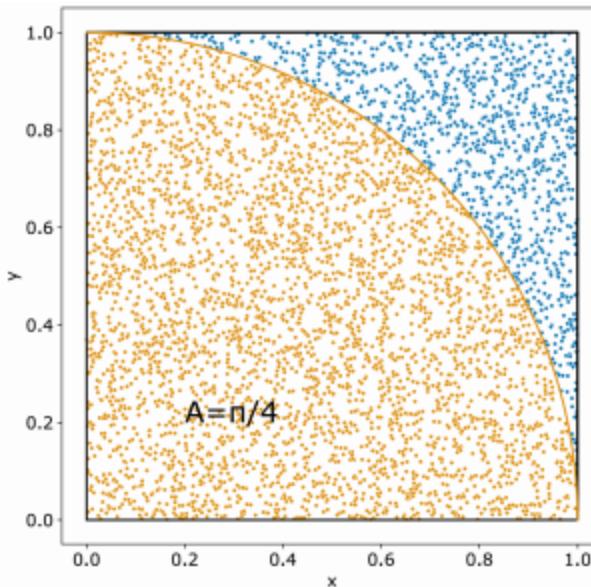


Fig. 4.22. Monte Carlo estimation of π .

Come prima operazione, vengono incluse tutte le librerie necessarie. Quindi viene definito il metodo **hit()** per verificare se un dato punto 2D generato rientra nel settore della circonferenza con raggio unitario centrato in $(0, 0)$. In particolare, il metodo campiona un punto 2D casuale nel quadrato unitario e quindi verifica se le sue coordinate soddisfano l'equazione $x^2 + y^2 \leq 1$

```

#include <iostream>
#include <cstdlib>
#include <random>
#include <upcxx/upcxx.hpp>

using namespace std;

int64_t hit()
{
    // generate random point coordinates
    double x = static_cast<double>(rand()) / RAND_MAX;
    double y = static_cast<double>(rand()) / RAND_MAX;
    // check whether the point falls within the circular
    // sector
    if (x*x + y*y <= 1.0) return 1;
    else return 0;
}

```

Il programma inizia invocando il metodo `upcxx::init()`, che imposta il runtime UPC++. Quindi, ogni rank, il cui identificatore è dato dal metodo `upcxx::rank_me()`,

inizializza un generatore casuale locale, che viene utilizzato per calcolare le coordinate dei diversi campioni (prove). Ogni rank viene eseguito in parallelo ed esegue 100.000 prove generando un punto casuale nel quadrato unitario e contando quanti di essi ricadono nel settore circolare. Questo valore, ovvero il numero di hit, viene memorizzato da ogni rank nella sua variabile locale *my_hits*.

```

int main(int argc, char **argv)
{
    upcxx::init();
    // each rank gets its own copy of local variables
    int64_t my_hits = 0;
    // the number of trials to run on each rank
    int my_trials = 100000;
    // initialize a random number generator for each rank
    srand(upcxx::rank_me());
    // local computation
    for (int i = 0; i < my_trials; i++) {
        my_hits += hit();
    }
    // rank 0 collects and sum the hits of all ranks
    int64_t hits = reduce_to_rank0(my_hits);
    // rank 0 prints the final estimate
    if (upcxx::rank_me() == 0) {
        int64_t trials = upcxx::rank_n() * my_trials;
        cout << "pi estimate: " << 4.0 * hits / trials <<
            endl;
    }
    upcxx::finalize();
    return 0;
}

```

Listing 4.33: Monte Carlo estimation of π in UPC++.

Una volta che ogni rank ha generato le proprie prove, tutti i risultati vengono raccolti dal rank 0, che determina il numero totale di successi attraverso il metodo `reduce_to_rank0` e li memorizza nella variabile *hits*. Infine, dato il numero totale di gradi, ottenuto tramite `upcxx::rank_n()`, il numero complessivo di punti generati (prove) può essere calcolato e la stima di π può essere derivata come $4 * \text{successi/prove}$

UPC++ fornisce diversi modi per eseguire l'operazione di riduzione, tra cui il più semplice è usare il metodo `allreduce`:

```
upcxx::allreduce(my_hits, plus<int>()).wait()
```

In questo caso, la riduzione collettiva viene calcolata applicando la funzione `plus` (ovvero, una somma) su tutti i valori locali dei miei hit. Questa operazione è asincrona e restituisce un *future* di tipo `int`, quindi il risultato finale deve essere atteso tramite il metodo `wait`. Sebbene il metodo `allreduce` rappresenti una

soluzione praticabile, è possibile implementare un'alternativa efficace che determina il numero complessivo di hit utilizzando i puntatori globali UPC++, basati sul modello di memoria condivisa APGAS.

Come prima operazione, il rank 0 assegna un puntatore globale all_hits_ptr a un array di numeri interi tramite la funzione **upcxx::new_array**. La dimensione dell'array è impostata uguale al numero di ranks, ottenuto come upcxx::rank_n(), poiché memorizzerà tutti i valori di hit dai ranghi remoti.

Il puntatore globale viene quindi trasmesso a tutti i rank, che aggiorneranno una posizione specifica dell'array condiviso in base al loro rankID ottenuto come upcxx::rank_me().

L'inserimento nell'array condiviso globale viene eseguito da ciascun rank utilizzando la funzione put remota upcxx::rput, che avvia una comunicazione unidirezionale per trasferire al processo remoto in modo asincrono.

```
int64_t reduce_to_rank0(int64_t my_hits)
{
    // Rank 0 creates an array to store all of the incoming
    // values
    upcxx::global_ptr<int64_t> all_hits_ptr = nullptr;
    if (upcxx::rank_me() == 0) {
        all_hits_ptr = upcxx::new_array<int64_t>(upcxx::
            rank_n());
    }
    // Rank 0 broadcasts the array global pointer to all
    // ranks
    all_hits_ptr = upcxx::broadcast(all_hits_ptr, 0).wait();
    // All ranks offset the pointer of the array by their
    // rank id
    upcxx::global_ptr<int64_t> my_hits_ptr = all_hits_ptr +
        upcxx::rank_me();
    // All rank put their local hits value into the shared
    // array
    upcxx::rput(my_hits, my_hits_ptr).wait();
}
```

Viene impiegato un **upcxx::barrier**, che funge da punto di sincronizzazione per garantire che tutti i trasferimenti remoti siano stati completati. Una volta completati tutti gli inserimenti, rank 0 utilizza la funzione **upcxx::global_ptr<T>::local()** per ottenere una versione locale del puntatore globale e sommare tutti i valori inseriti dai ranghi remoti. Infine, dealloca l'array condiviso utilizzando la funzione **upcxx::delete_array** e restituisce il risultato ridotto memorizzato nella variabile hits

```

    // wait for all insertions to be completed
    upcxx::barrier();
    // rank 0 performs the reduce-by-sum operation
    int64_t hits = 0;
    if (upcxx::rank_me() == 0) {
        // get a local pointer to the shared array
        int64_t *local_hits_ptrs = all_hits_ptr.local();
        // sum all the values stored in the array
        for (int i = 0; i < upcxx::rank_n(); i++) {
            hits += local_hits_ptrs[i];
        }
        upcxx::delete_array(all_hits_ptr);
    }
    return hits;
}

```

Listing 4.34: Determining the total number of hits by using global pointers.

CONFRONTO SU APPLICAZIONI BATCH

SPARK vs HADOOP

Discussiamo un'applicazione per scoprire automaticamente i modelli di mobilità degli utenti da

post Flickr geotaggati generati nella città di Roma. L'applicazione mira a scoprire le traiettorie più frequenti degli utenti in alcune posizioni o aree specifiche che sono di interesse per la nostra analisi, comunemente denominate **punti di interesse** (Pol).

Un Pol è una posizione considerata utile o interessante, come un'attrazione turistica o una sede aziendale. Poiché le informazioni su un Pol sono generalmente limitate a un indirizzo o alle coordinate GPS, è difficile abbinare le traiettorie ai Pol. Pertanto, è spesso utile definire le cosiddette **regioni di interesse** (Rol) che rappresentano i confini delle aree dei Pol. Una traiettoria può essere definita come una sequenza di Rol, che rappresentano un modello di movimento nel tempo. Una **traiettoria frequente è una sequenza di Rol che sono visitate frequentemente dagli utenti**.

Dopo aver raccolto un set di post geotaggati da Flickr, applichiamo una pre-elaborazione per filtrare i

post geotaggati e mappare ogni post geotaggato a un Rol. Quindi, applichiamo il trajectory mining per estrarre frequenti modelli di mobilità nelle traiettorie degli utenti attraverso i Rol, con l'obiettivo di comprendere meglio come le persone si muovono nella città di Roma.

I post sui social media spesso contengono informazioni sulla posizione del post o altri metadati che possono essere utilizzati per dedurre la posizione dell'utente al momento della creazione del post:

- una descrizione testuale
- un set di parole chiave associate al post
- una coppia latitudine/longitudine della posizione in cui è stato creato il post
- un ID che identifica l'utente che ha creato il post
- un timestamp che indica la data di creazione del post

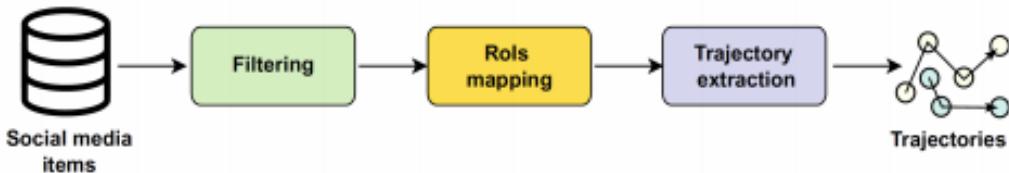


Fig. 5.1. Workflow of the trajectory mining application.

IMPLEMENTAZIONE CON SPARK

Iniziamo con un'implementazione dell'applicazione usando Apache Spark e il linguaggio Scala. Definiamo due classi, vale a dire **SingleTrajectory** e **UserTrajectory**, per rappresentare un singolo punto in una traiettoria (vale a dire, una coppia `<Poi, timestamp>`) e un utente in una data data (vale a dire, una coppia `<user_id, date>`), rispettivamente. Queste classi definiscono i metodi `equals` e `hashCode` che saranno usati dalle procedure successive quando si confrontano traiettorie e utenti

```

class SingleTrajectory(var poi: String = "", var daytime: String = "") {
  override def hashCode(): Int = 31 * poi.hashCode() + daytime.hashCode()
  override def equals(o: Any): Boolean = {
    if (o == null) return false
    if (getClass ne o.getClass) return false
    val other = o.asInstanceOf[SingleTrajectory]
    if (poi.equals(other.poi) && daytime.equals(other.daytime))
      return true
    false
  }
}

class UserTrajectory (var username: String = "", var daytime: String = "") {
  override def hashCode(): Int = 31 * username.hashCode() + daytime.hashCode()
  override def equals(o: Any): Boolean = {
    if (o == null) return false
    if (getClass ne o.getClass) return false
    val other = o.asInstanceOf[UserTrajectory]
    if (username.equals(other.username) && daytime.equals(other.daytime))
      return true
    false
  }
}
  
```

Listing 5.1: `SingleTrajectory` and `UserTrajectory` classes for representing a trajectory and a user, respectively.

Il metodo principale legge il dataset di input JSON in uno **Spark DataFrame** e applica una serie di funzioni per filtrare i punti dati non rilevanti per l'analisi, come quelli con coordinate GPS non valide o quelli non situati a Roma.

Il codice utilizza la classe **KMLUtils** per leggere un file Keyhole Markup Language (KML) contenente le coordinate di alcuni Pol a Roma e convertirlo in una mappa Scala in cui le chiavi sono i nomi dei Pol (*ad esempio, Colosseo, Musei Vaticani e Mausoleo di Adriano*) e i valori associati sono le coordinate che definiscono i confini di quell'area come un poligono.

Il programma calcola quindi le traiettorie dell'utente utilizzando l'algoritmo **FPGrowth** e restituisce gli itemset frequenti e le regole di associazione che rappresentano le traiettorie ottenute

```
def main(args: Array[String]): Unit = {
    val dataset = "FlickrRomeSample.json"
    val kmlPath = "rome.kml"
    val spark = SparkSession
        .builder
        .appName("TrajectoryMining")
        .master("local[*]")
        .getOrCreate()
    val stringShapeMap = KMLUtils.lookupFromKml(kmlPath).
        asScala
    var df = spark.read.json(dataset)
    df = filterFlickrDataframe(df)
    df = df.filter(r => filterIsGPSValid(r) &&
        filterIsInRome(r))
    val trajectories = computeTrajectoryUsingFPGrowth(df,
        stringShapeMap)
    trajectories._1.foreach(println)
    trajectories._2.foreach(println)
    spark.close()
}
```

Listing 5.2: Main method.

Loaded KML shapes:

```
Map(colosseum -> POLYGON((12.490838 41.891314,12.490249 41.890018,12.490204 41.889794,12.490321
41.889566,12.490494 41.889354,12.490967 41.889194,12.491686 41.88922,12.492165 41.889213,12.492505
41.88919,12.492 683 41.889199,12.49333 41.88923,12.493665 41.88934,12.493783 41.889477,12.494041
41.88977,12.494276 41.89039,12.49422 41.89073,12.493732 41.890963,12.49302 41.891225,12.492193
41.89145,12.491616 41.891608,12.490971 41.891729,12.490908 41.89152,12.490838 41.891314)),

vaticanmuseums -> POLYGON((12.455288 41.906637,12.454548 41.906892,12.451833 41.906366,12.451571
41.905994,12.454205 41.902972,12.455181 41.903076,12.455288 41.906637)), piazzacolonna ->
POLYGON((12.479552 41.900674,12.47962 41.900517,12.48023 41.900682,12.480522 41.900758,12.480407
41.90106,12.480311 41.901285,12.479823 41.901185,12.479384 41.90109,12.47955241.900674)),

piazzadelpopolo -> POLYGON((12.47534 41.911012,12.475206 41.910665,12.475384 41.910282,12.475788
41.91008,12.476215 41.909986,12.476639 41.910022,12.477036 41.910162,12.477325 41.910597,12.47739
41.91098,12.477296 41.911138,12.477138 41.911248,12.476891 41.911424,12.47655 41.911503,12.476038
41.911527,12.475716 41.911397,12.47534 41.911012))

...

```

```
val stringShapeMap = KMLUtils.lookupFromKml(kmlPath).asScala
```

Read JSON dataset into Spark DataFrame:

```
var df = spark.read.json(dataset)
```

[comments]	datePosted	dateTaken	description	geoData	owner	tags	title views
0 Dec 17, 2016 2:00... Dec 16, 2016 7:59... On the Via del Le... {16, 41.904057, 1... [97629199@N00} [via, del, leonci... Romans of Our Time 701							
0 Dec 16, 2016 8:46... Dec 16, 2016 5:37...			{16, 41.907604, 1... [7790945@N07}	[]	Xstnas Time	11	
0 Dec 17, 2016 12:0... Dec 16, 2016 5:13... Roma, ponte Milvi... {16, 41.936386, 1... [87405113@N03}				[] Riflessi magici ...	16		
0 Dec 16, 2016 5:18... Dec 16, 2016 5:09... Foto Alvaro ed El... {15, 41.89587, 12... [35155107@N08} [piazzamaceldecor... 1752 2009 Piazza ... 136							
0 Dec 16, 2016 5:18... Dec 16, 2016 5:09... Incisione Giusepp... {15, 41.89587, 12... [35155107@N08} [piazzamaceldecor... 1752 2009 Piazza ... 191							
0 Dec 16, 2016 5:18... Dec 16, 2016 5:09... Mappa del G.B. No... {15, 41.89587, 12... [35155107@N08} [piazzamaceldecor... 1748 Rione Monti ... 140							
0 Dec 16, 2016 5:18... Dec 16, 2016 5:09... Mappa del G.B. No... {15, 41.89587, 12... [35155107@N08} [piazzamaceldecor... 1748 1889 Pianta ... 233							
0 Dec 16, 2016 5:18... Dec 16, 2016 5:08... Foto d'anonimo di... {15, 41.89587, 12... [35155107@N08} [piazzamaceldecor... 1748 1889 La Pia... 367							
0 Dec 16, 2016 5:18... Dec 16, 2016 5:09... Mappa del G.B. No... {15, 41.89587, 12... [35155107@N08} [piazzamaceldecor... 1748 Pianta del R... 166							
0 Jan 5, 2017 3:10... Dec 16, 2016 2:33... Opened the door a... {15, 41.824867, 1... [60655827@N00} [eur, roma, rome,... Winter reflections 174							
0 Dec 23, 2016 1:19... Dec 16, 2016 8:25... edf {16, 41.851837, 1... [49597234@N06}				[] IMG_20161216_082525 19			
0 Dec 17, 2016 9:26... Dec 16, 2016 12:4... Chiedo venia. Il ... {16, 41.905903, 1... [5537130@N00}				[] Errata corrig... 77			
0 Dec 19, 2016 2:17... Dec 16, 2016 12:0...			{16, 41.90183, 12... [32162360@N07} [shakespeare, liv... ShakespeareLives ... 104				
0 Dec 19, 2016 2:18... Dec 16, 2016 12:0...			{16, 41.90183, 12... [32162360@N07} [shakespeare, liv... ShakespeareLives ... 98				
0 Dec 19, 2016 2:18... Dec 16, 2016 12:0...			{16, 41.90183, 12... [32162360@N07} [shakespeare, liv... ShakespeareLives ... 106				
0 Dec 19, 2016 2:18... Dec 16, 2016 12:0...			{16, 41.90183, 12... [32162360@N07} [shakespeare, liv... ShakespeareLives ... 104				
0 Dec 19, 2016 2:17... Dec 16, 2016 12:0...			{16, 41.90183, 12... [32162360@N07} [shakespeare, liv... ShakespeareLives ... 101				
0 Dec 19, 2016 2:17... Dec 16, 2016 12:0...			{16, 41.90183, 12... [32162360@N07} [shakespeare, liv... ShakespeareLives ... 100				
0 Dec 19, 2016 2:17... Dec 16, 2016 12:0...			{16, 41.90183, 12... [32162360@N07} [shakespeare, liv... ShakespeareLives ... 100				
0 Dec 19, 2016 2:18... Dec 16, 2016 12:0...			{16, 41.90183, 12... [32162360@N07} [shakespeare, liv... ShakespeareLives ... 172				

Il codice per la fase di filtraggio include tre filtri:

1. un filtro per selezionare solo le colonne interessanti dal dataset Flickr per l'analisi
2. un filtro che mantiene solo i dati con coordinate GPS valide (vale a dire, le cui longitudini e latitudini sono correttamente definite)
3. un filtro per mantenere solo i post che sono stati pubblicati a Roma

La funzione `filterIsInRome` sfrutta una classe di utilità, vale a dire GeoUtils, che fornisce un set di metodi di utilità per interagire con i dati geospaziali, come la conversione di una

coppia di dati di longitudine e latitudine in un punto o la verifica se un Pol è contenuto in un altro Pol (basato sulla libreria Java Spatial4j)

```

def filterFlickrDataframe(dataframe: DataFrame): DataFrame
    =
    dataframe.select("geoData.latitude", "geoData.
        longitude", "geoData.accuracy", "owner.id",
        "dateTaken")
}

def filterIsGPSValid(r: Row): Boolean = {
    r.getAs[Double]("longitude") > 0 && r.getAs[Double]
        ("latitude") > 0
}

def filter isInRome(r: Row): Boolean = {
    val p = GeoUtils.getPoint(r.getAs[Double]("longitude"),
        r.getAs[Double]("latitude"))
    GeoUtils.isContained(p, romeShape)
}

```

Listing 5.3: Filtering methods.

Filter DataFrame by latitude, longitude, owner.id and dateTaken:

latitude	longitude	accuracy	id	dateTaken
41.904057	12.478018	16	97629199@N00 Dec 16, 2016 7:59...	
41.907664	12.516125	16	77990945@N07 Dec 16, 2016 5:37...	
41.936386	12.471388	16	87405113@N03 Dec 16, 2016 5:13...	
41.89587	12.483166	15	35155107@N08 Dec 16, 2016 5:09...	
41.89587	12.483166	15	35155107@N08 Dec 16, 2016 5:09...	
41.89587	12.483166	15	35155107@N08 Dec 16, 2016 5:09...	
41.89587	12.483166	15	35155107@N08 Dec 16, 2016 5:09...	
41.89587	12.483166	15	35155107@N08 Dec 16, 2016 5:09...	
41.89587	12.483166	15	35155107@N08 Dec 16, 2016 5:09...	
41.89587	12.483166	15	35155107@N08 Dec 16, 2016 5:09...	
41.89587	12.483166	15	35155107@N08 Dec 16, 2016 5:09...	
41.824867	12.475115	15	60655827@N00 Dec 16, 2016 2:33...	
41.851837	12.556422	16	49597234@N06 Dec 16, 2016 8:25...	
41.905903	12.464705	16	55371306@N00 Dec 16, 2016 12:4...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	
41.90183	12.478119	16	32162360@N07 Dec 16, 2016 12:0...	

```

def filterFlickrDataframe(dataframe: DataFrame): DataFrame
    =
    dataframe.select("geoData.latitude", "geoData.
        longitude", "geoData.accuracy", "owner.id",
        "dateTaken")
}

```

Dopo aver filtrato il set di dati di input, possiamo estrarre le regole di associazione usando un algoritmo di mining di pattern frequenti. Qui utilizziamo **FPGrowth**, un algoritmo popolare per il mining di itemset frequenti in database transazionali.

Per adattare l'algoritmo al mining di traiettorie, definiamo un metodo **mapCreateTrajectory** che, data una riga del set di dati di input e il set di Pol estratti dal file KML, restituisce una tupla composta dall'ID utente e dal Pol che ha

visitato nella forma di <UserTrajectory, SingleTrajectory> precedentemente definito.

```
def mapCreateTrajectory(row: Row, shapeMap: mutable.Map[String, String]): (UserTrajectory, Set[SingleTrajectory]) = {
    val lat = row.getDouble(0)
    val lon = row.getDouble(1)
    val username = row.getString(3)
    val d = row.getAs[String](4).substring(0, 12)
    val point = GeoUtils.getPoint(lon, lat)
    var arr: Array[SingleTrajectory] =
        Array[SingleTrajectory]()
    for ((place, polygon) <- shapeMap) {
        val pol = GeoUtils.getPolygonFromString(polygon)
        if (GeoUtils.isContained(point, pol)) {
            val trajectory = new SingleTrajectory(place, d)
            arr = arr :+ trajectory
        }
    }
    val ut = new UserTrajectory(username, d)
    (ut, arr.toSet)
}
```

Listing 5.4: Method for extracting a user and the Pol visited by them.

Il primo passaggio del metodo **computeTrajectoryUsingFPGrowth** consiste nel preparare i dati della transazione sotto forma di `RDD[Array[String]]` creando traiettorie tramite il metodo `mapCreateTrajectory` definito sopra

Le traiettorie risultanti vengono filtrate per rimuovere quelle vuote e quindi ridotte per chiave per raggruppare insieme le traiettorie relative allo stesso utente.

Successivamente, vengono trasformate in un set di itemset univoci utilizzando la funzione `distinct`. Questo viene quindi passato come input all'algoritmo FPGrowth importato dalla libreria MLlib,

che crea un modello basato sugli itemset frequenti e genera regole di associazione utilizzando i parametri `minSupport` e `minConfidence`.

Infine, il metodo restituisce gli itemset frequenti e le regole di associazione generate

```

def computeTrajectoryUsingFPGrowth(df: DataFrame,
    stringShapeMap: mutable.Map[String, String],
    minSupport: Double = 0.01, minConfidence:Double = 0.2)
    = {
  val prepareTransaction = df
    .rdd
    .map(x=>mapCreateTrajectory(x, stringShapeMap))
    .filter(x=>x._2.nonEmpty)
    .reduceByKey((x,y)=> x ++ y)

  val transactions = prepareTransaction
    .map{x=>var arr: Array[String] = Array[String]()
      x._2.foreach(x=>{
        arr = arr :+ x.poi})
      arr}
    .map(x => x.distinct)

  val fpg = new FPGrowth()
    .setMinSupport(minSupport)

  val model = fpg.run(transactions)

  (model.freqItemsets.collect(),model.
    generateAssociationRules(minConfidence).collect())
}

```

Listing 5.5: Computing trajectory using FP-Growth.

prepareTransaction:

```

(user 35716709@N04, day Nov 20, 2016,Set(poi villaborghese, timestamp Nov 20, 2016))
(user 61240032@N05, day Nov 29, 2016,Set(poi stpeterbasilica, timestamp Nov 29, 2016, poi
colosseum, timestamp Nov 29, 2016, poi stpeterbasilica, timestamp Nov 29, 2016))
(user 99366715@N00, day Nov 30, 2016,Set(poi piazzanavona, timestamp Nov 30, 2016))
(user 52714236@N02, day Nov 30, 2016,Set(poi trastevere, timestamp Nov 30, 2016, poi
piazzanavona, timestamp Nov 30, 2016, poi romanforum, timestamp Nov 30, 2016))
(user 92919639@N03, day Dec 10, 2016,Set(poi romanforum, timestamp Dec 10, 2016))

...

```

```

val prepareTransaction = df
  .rdd
  .map(x=>mapCreateTrajectory(x, stringShapeMap))
  .filter(x=>x._2.nonEmpty)
  .reduceByKey((x,y)=> x ++ y)

```

x._2 is the third field of a SingleTrajectory object

```

transactions:
Array(villaborghese)
Array(stpeterbasilica, colosseum)
Array(piazzanavona)
Array(trastevere, piazzanavona, romanforum)
Array(romanforum)
...
val transactions = prepareTransaction
.map{x=>var arr: Array[String] = Array[String]()
      x._2.foreach(x=>{
        arr = arr ++ x.poi})
      arr}
    .map(x => x.distinct)

```

Frequent itemset (model.freqItemsets):

```

{campodefiori}: 388
{piazzadispagna}: 211
{piazzadispagna,colosseum}: 34
{piazzadispagna,pantheon}: 51
{piazzadispagna,trevifountain}: 32
...

```

```

val fpg = new FP_Growth()
.setMinSupport(minSupport)

val model = fpg.run(transactions)

```

Di seguito viene presentato un estratto delle regole di associazione prodotte come output, in cui i punteggi di confidenza e lift sono riportati per ogni regola estratta dall'algoritmo FP-Growth.

Un valore lift maggiore di 1 indica un'associazione positiva tra i due elementi, mentre un valore lift inferiore a 1 indica un'associazione negativa tra i due elementi. Il parametro minSupport per riprodurre questo output è 0,01.

Nell'implementazione MLlib dell'algoritmo, il livello di supporto minimo del pattern frequente è impostato in modo tale che qualsiasi pattern che appare più di (minSupport * dimensione del set di dati) volte venga emesso

```

{romanforum,stpeterbasilica} => {colosseum}: (confidence: 0.75; lift: 3.84)
{colosseum, stpeterbasilica} => {romanforum}: (confidence: 0.375; lift: 3.48)
{colosseum,stpeterbasilica} => {pantheon}: (confidence: 0.375; lift: 3.24)
{piazzadelpopolo} => {piazzadispagna}: (confidence: 0.25; lift: 2.98)

```

IMPLEMENTAZIONE CON HADOOP

La stessa applicazione descritta usando Spark è ora implementata tramite il paradigma MapReduce in Apache Hadoop. In particolare, definiamo:

- un mapper, chiamato DataMapperFilter, che esegue i passaggi di filtraggio
- un reducer, chiamato DataReducerByDay, che estrae i Rol per il mining delle traiettorie;
- un main, che combina il mapper e il reducer e quindi applica l'algoritmo FP-Growth dalla libreria Apache Mahout per estrarre le traiettorie dai Rol estratti dal reducer.

Qui viene utilizzata la stessa implementazione Java delle classi GeoUtils e KMLUtils descritte per l'applicazione Spark. Per scopi di utilità, abbiamo definito una classe per rappresentare e gestire i dati Flickr, chiamata Flickr, che fornisce metodi getter per ID utente, longitudine, latitudine e altri campi. Inoltre, questa classe consente di importare dati Flickr da una stringa o da un oggetto JSON per creare un elemento Flickr ed esportare i dati come stringa

La classe **DataMapperFilter** estende la classe Mapper della libreria Hadoop MapReduce e include alcune variabili di istanza, come un oggetto Shape che rappresenta un poligono geospaziale (ad esempio, i confini geografici della città di Roma), un oggetto Map che mappa i nomi delle posizioni sui poligoni e oggetti Text che rappresentano coppie chiave-valore di output.

Il metodo setup viene chiamato prima che venga eseguito il metodo map e inizializza la variabile romeShape con un poligono basato sulle coordinate di Roma e sulla variabile shapeMap chiamando i metodi di utilità nelle classi GeoUtils e KMLUtils

```
public class DataMapperFilter extends Mapper<LongWritable,
    Text, Text, Text> {
    private Shape romeShape;
    private Map<String, String> shapeMap;
    private final String kmlPath = "rome.kml";
    private Text outputKey = new Text();
    private Text outputValue = new Text();
    private final double LAT = 12.492373;
    private final double LNG = 41.890251;
    private final int RADIUS = 10000;

    @Override
    protected void setup(Mapper<LongWritable, Text, Text,
        Text>.Context context)
        throws IOException, InterruptedException {
        romeShape = GeoUtils.getCircle(GeoUtils.getPoint
            (LAT, LNG), RADIUS);
        shapeMap = KMLUtils.lookupFromKml(kmlPath);
    }
}
```

Il metodo map è la funzione di elaborazione principale del mapper. Utilizza la classe Flickr per analizzare i dati JSON di input ed estrarre informazioni sulle coordinate GPS del post, l'ID utente e altri metadati.

Il metodo applica due funzioni filtro, filterIsGPSValid e filterIsInRome, per verificare i criteri di filtraggio, ovvero avere coordinate GPS valide e trovarsi nelle vicinanze di Roma.

Se il post supera i filtri, il metodo utilizza la classe GeoUtils per creare un oggetto Shape che rappresenta la posizione del post e verifica se è contenuto in uno dei poligoni definiti nell'oggetto shapeMap estratto dal file KML dei Pol a Roma. Se viene trovata una corrispondenza, il Pol viene impostato per l'elemento Flickr.

Infine, l'ID utente e l'oggetto Flickr serializzato arricchito con informazioni sul Rol estratto vengono scritti nel contesto di output

```

public void map(LongWritable key, Text value, Context context)
    throws InterruptedException, IOException {
    Flickr f = new Flickr();
    f.importFromFlickrJ800(value.toString());

    if (!(filterIsGPValid(f) && filterIsInRome(f)))
        return;

    Shape point = GeoUtils.getPoint(f.getLongitude(),
                                    f.getLatitude());
    boolean found = false;
    for (Entry<String, String> entry : shapeMap.entrySet())
    {
        String place = entry.getKey();
        String polygon = entry.getValue();
        try {
            Shape pol = GeoUtils.getPolygonFromString(polygon);
            if (GeoUtils.isContained(point, pol)) {
                f.setRoi(place);
                found = true;
            }
        } catch (IOException | ParseException e) {
            e.printStackTrace();
        }
    }

    if (found) {
        outputKey.set(f.getUserId());
        outputValue.set(f.export());
        context.write(outputKey, outputValue);
    }
}

private boolean filterIsGPValid(Flickr f) {
    return f.getLongitude() > 0 && f.getLatitude() > 0;
}

private boolean filterIsInRome(Flickr f) {
    Point p = (Point) GeoUtils.getPoint(f.getLongitude(), f.
                                         getLatitude());
    return GeoUtils.isContained(p, romeShape);
}

```

Listing 5.7: DataMapperFilter class.

Ogni reducer riceve per ogni utente il set dei suoi post Flickr, che sono stati precedentemente arricchiti con informazioni sul Rol. Lo scopo del reducer è calcolare le traiettorie come una sequenza di Rol. Il nucleo della fase di riduzione è il metodo **concatenateLocationsByDay**, che crea la sequenza ordinata di Rol visitati ogni giorno.

Il metodo utilizza un comparatore per ordinare gli elementi Flickr per data, quindi, per ogni giorno, viene creata una stringa con i Rol visitati e aggiunta a un elenco, che verrà restituito al metodo reduce. Il metodo reduce restituisce quindi, nel contesto distribuito, l'elenco di Rol per ogni giorno separati da uno spazio vuoto.

```

public class DataReducerByDay extends Reducer<Text, Text,
    NullWritable, Text> {
    private Text outputValue = new Text();

    @Override
    public void reduce(Text key, Iterable<Text> values, Context
        context) throws java.io.IOException,
        InterruptedException {

        List<String> res = concatenateLocationsByDay(values);
        for (String s: res) {
            outputValue.set(s);
            context.write(NullWritable.get(), outputValue);
        }
    }
}

```

```

st peter basilica
piazza dispagna
mausoleum of hadrian
st peter basilica
capitoline hill
trevi fountain
pantheon piazza del popolo piazza dispagna trevi fountain piazza navona
colosseum mausoleum of hadrian

```

Il metodo **concatenateLocationsByDay** crea la sequenza ordinata di Roi visitati ogni giorno. Utilizza un comparatore per ordinare gli elementi Flickr in base alla data, quindi, per ogni giorno, viene creata una stringa con i Roi visitati e aggiunta a un elenco, che verrà restituito al metodo reduce

be returned to the reduce method.

```

private static List<String> concatenateLocationsByDay
    (Iterable<Text> listItems) {
    LocalDateTime oldTimestamp = new LocalDateTime(0);
    LocalDateTime currTimestamp = null;
    List<String> ret = new LinkedList<String>();
    Set<String> s = null;
    String oldLocation = null;
    String currentLocation = null;

    List<Flickr> lf = new LinkedList<Flickr>();
    for (Text value: listItems) {
        Flickr item = new Flickr();
        item.importFromString(value.toString());
        lf.add(item);
    }

    lf.sort(new Comparator<Flickr>() {
        @Override
        public int compare(Flickr f1, Flickr f2) {
            return (f1.getDateWithoutTime().compareTo(f2.
                getDateWithoutTime()));
        }
    });
    for (Flickr f : lf) {
        currTimestamp = f.getDateWithoutTime();
        if (Days.daysBetween(oldTimestamp, currTimestamp).
            getDays() > 0) {
            if (s != null)
                ret.add(s.toString().replaceAll("\\[", "").
                    replaceAll("\\]", "").replaceAll(" ", " ").
                    trim());
            s = new HashSet<String>();
            oldlocation = null;
            oldTimestamp = currTimestamp;
        }
        currentLocation = f.getRoi();
        if (!currentLocation.equals(oldLocation)) {
            s.add(currentLocation);
            oldLocation = currentLocation;
        }
    }
    if (s != null)
        ret.add(s.toString().replaceAll("\\[", "").
            replaceAll("\\]", "").replaceAll(" ", " ").
            trim());
    return ret;
}

```

Listing 5.8: DataReducerByDay class.

```

public class Main extends Configured implements Tool {

    private static String inputPath = "FlickrRome.json";
    private static String trajOutputPath = "outputMR/";
    private static String fpOutputPath = "fpOutput/";

    @Override
    public int run(String[] args) throws Exception {
        int minimumSupport = Integer.parseInt(args[0]);
        int maxPatterns = Integer.parseInt(args[1]);
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "file:///");
        /** Trajectory Job configuration */
        Job trajJob = Job.getInstance(conf, "TrajectoryJob");

        /* FileSystem setting */
        FileInputFormat.addInputPaths(trajJob, inputPath);
        FileOutputFormat.setOutputPath(trajJob, new Path
            (trajOutputPath));
        FileSystem fs = FileSystem.get(conf);
        if (fs.exists(new Path(trajOutputPath))) {
            fs.delete(new Path(trajOutputPath), true);
        }
        trajJob.setMapperClass(DataMapperFilter.class);
        // Set the output class of key and value for the mapper
        trajJob.setMapOutputKeyClass(Text.class);
        trajJob.setMapOutputValueClass(Text.class);

        trajJob.setReducerClass(DataReducerByDay.class);
        // Set the output class of key and value for the reducer
        trajJob.setOutputKeyClass(Text.class);
        trajJob.setOutputValueClass(Text.class);

        boolean completed;
        completed = trajJob.waitForCompletion(true);
        if (!completed)
            return 1;

        /** Run FP-Growth algorithm */
        if (!fs.exists(new Path(fpOutputPath)))
            fs.delete(new Path(fpOutputPath), true);
        FpGrowth.runFpGrowth(trajOutputPath, fpOutputPath,
            minimumSupport, maxPatterns);
        return 1;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
            new Main(), args);
        System.exit(res);
    }
}

```

Listing 5.9: Main class.

SCEGLIERE IL FRAMEWORK GIUSTO

FATTORI PRINCIPALI

I principali fattori da considerare nella selezione del framework appropriato per implementare un'applicazione Big Data includono:

- Dati in ingresso: si riferisce principalmente al volume dei dati (sia in termini di dimensione che di dimensionalità del dataset di input), alla velocità e alla varietà.
- Classe dell'applicazione: riguarda il tipo di applicazione che deve essere implementata (ad esempio, applicazioni batch, stream, basate su grafi o su query).
- Infrastruttura: si riferisce all'infrastruttura di archiviazione e calcolo che verrà utilizzata per eseguire l'applicazione Big Data (ad esempio, infrastrutture on-premise, basate su cloud o ibride).

DATI IN INGRESSO

VOLUME

- Il volume dei dati influisce sui requisiti di archiviazione delle applicazioni, poiché la memorizzazione di grandi quantità di dati richiede soluzioni di

archiviazione distribuita che garantiscano la replica dei dati, la tolleranza ai guasti e la scalabilità, come il **Hadoop Distributed File System (HDFS)**.

- Il volume dei dati incide anche sui requisiti di elaborazione, poiché l'elaborazione di grandi volumi di dati richiede sistemi di calcolo distribuito che possano scalare orizzontalmente, come **Hadoop**, utilizzato per l'elaborazione parallela, e **Spark**, che offre elaborazione in memoria adatta per algoritmi iterativi e analisi interattive.
- I dati ad alta dimensionalità potrebbero richiedere l'uso di tecniche di riduzione della dimensionalità, come **l'analisi delle componenti principali (PCA)** o la **decomposizione ai valori singolari (SVD)**. Un framework che supporta queste tecniche è **Spark**, tramite la sua libreria **Mlib**, che facilita l'analisi e l'estrazione di insight dai dati ad alta dimensionalità.

VELOCITA'

- La **velocità** con cui i dati vengono generati è una caratteristica fondamentale dei Big Data e richiede modelli di programmazione e sistemi in grado di acquisire, elaborare e analizzare i dati in tempo reale per consentire decisioni tempestive.
- La velocità dei dati influenza la capacità di elaborazione e analisi dell'applicazione, poiché sono necessarie tecniche come **windowing** e **aggregazione basata sul tempo** per catturare informazioni rilevanti da un flusso di dati generato ad alta velocità.
- L'elaborazione in tempo reale necessita di capacità di **elaborazione a bassa latenza**, come il **processing in-memory**, per migliorare la reattività di un'applicazione Big Data.
- Sebbene i sistemi di **streaming a micro-batch** come **Spark Streaming** possano essere utilizzati in alcuni scenari, i framework di **stream processing** come **Storm** vengono tipicamente adottati per elaborare flussi di dati in tempo reale, permettendo alle applicazioni di rispondere agli eventi nel momento in cui si verificano.

VARIETA'

- La **varietà**, ovvero l'eterogeneità di tipi di dati, formati e fonti, richiede la gestione dell'integrazione, trasformazione e analisi dei dati in modo flessibile e adattabile.
- I dati provenienti da più fonti e in diversi formati possono essere **pre-elaborati** prima dell'analisi utilizzando sistemi che supportano operazioni di **estrazione, trasformazione e caricamento (ETL)**, come **Hive** e **Pig**.
- I dati strutturati possono essere analizzati utilizzando tecniche tradizionali di database relazionali, come quelle fornite da **Hive** e **Pig**.
- I dati non strutturati (ad esempio, il testo) richiedono tecniche specifiche, come **l'elaborazione del linguaggio naturale (NLP)** per i dati testuali, supportata da alcuni sistemi, tra cui **Spark**.
- **Spark** è il framework più versatile per l'elaborazione di dati eterogenei, poiché fornisce API per **batch processing, stream processing, machine learning, analisi su grafi** e **DataFrames** per lavorare con diversi formati di dati, come **CSV, JSON e tabelle di database**.

CLASSE DELL'APPLICAZIONE

BATCH

- Le applicazioni batch elaborano grandi quantità di dati raccolti e analizzati insieme, generalmente durante le ore di minor utilizzo dei sistemi.
- Sono particolarmente utili per **l'analisi dei dati storici, la generazione di report e l'esecuzione di analisi complesse** che richiedono notevoli risorse di calcolo.
- **Spark** e **Hadoop** sono ampiamente utilizzati per il batch processing grazie alle loro capacità di archiviazione ed elaborazione distribuite:
 - **Hadoop** include archiviazione fault-tolerant e un framework per l'elaborazione distribuita.
 - **Spark** offre un motore di elaborazione dati veloce e flessibile con API di alto livello e supporto per machine learning.

- **Airflow** può essere utilizzato per sviluppare e monitorare applicazioni batch basate su workflow, **automatizzando i processi di elaborazione**.

STREAM

- Le applicazioni di **stream processing** elaborano e analizzano i dati mentre vengono ricevuti, senza la necessità di memorizzarli in repository centralizzati.
- Sono particolarmente utili in settori come **finanza, telecomunicazioni e trasporti**, dove è necessaria un'analisi in tempo reale.
- Esempi di applicazioni di **Big Data in streaming** includono:
 - Analisi dei dati in tempo reale
 - Rilevamento delle frodi
 - Monitoraggio in tempo reale di sensori e dispositivi IoT
- **Storm** e **Spark** vengono utilizzati per il processamento dei dati in streaming:
 - **Storm** offre elaborazione in tempo reale **a bassa latenza, scalabile e fault-tolerant**.
 - **Spark** supporta **lo streaming a micro-batch** attraverso API dedicate.

ON-PREMISE

- L'infrastruttura **on-premise** si riferisce alla distribuzione di hardware e software all'interno dell'azienda, evitando il trasferimento di grandi quantità di dati su server remoti.
- Questa configurazione consente **maggior sicurezza e conformità alle normative sulla privacy**.
- Le infrastrutture on-premise spesso si basano su hardware generico con **Hadoop** per processare grandi dataset a costi ridotti.
- **Spark** è una soluzione più veloce per l'elaborazione in-memory, ma richiede **una grande quantità di RAM**, aumentando i costi.

CLOUD BASED

- L'infrastruttura **cloud-based** utilizza risorse cloud per archiviare ed elaborare i dati.
- I servizi cloud offrono **scalabilità e flessibilità**, consentendo di aggiungere o rimuovere risorse in base alle esigenze.
- Servizi specifici per Big Data includono **Amazon EMR, Azure HDInsight e Google Cloud Dataproc**.
- Tuttavia, il cloud introduce sfide di **sicurezza, conformità e gestione dei dati**.
- Le infrastrutture cloud devono essere conformi a normative come **GDPR e CCPA**.

ALTRI FATTORI

- **Competenze di programmazione** di sviluppatori e progettisti.
- **Caratteristiche dell'ecosistema** del framework scelto.
- **Dimensione e attività della community di sviluppatori**.
- **Requisiti di privacy e accesso ai dati**.
- **Costi dell'infrastruttura hardware/software**.
- **Disponibilità di librerie di analisi dati e machine learning**.
- **Livello di astrazione** offerto dal framework.