ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

ANALISI E VALUTAZIONE DELLA PIATTAFORMA SPARK

Relazione finale in
LABORATORIO DI BASI DI DATI

Relatore
MATTEO GOLFARELLI

Presentata da
LORENZO GATTO

Co-relatori
LORENZO BALDACCI

Prima Sessione di Laurea
Anno Accademico 2014 – 2015

Alla mia famiglia, che mi ha sostenuto
Ai miei amici, che hanno creduto in me

Indice

Introduzione

Negli ultimi anni i dati, la loro gestione e gli strumenti per la loro analisi hanno subito una trasformazione. Si è visto un notevole aumento dei dati raccolti dagli utenti, che si aggira tra il 40 e il 60 percento annuo, grazie ad applicazioni web, sensori, ecc.. Ciò ha fatto nascere il termine Big Data, con il quale ci si riferisce a dataset talmente grandi che non sono gestibili da sistemi tradizionali, come DBMS relazionali in esecuzione su una singola macchina. Infatti, quando la dimensione di un dataset supera pochi terabyte, si è obbligati ad utilizzare un sistema distribuito, in cui i dati sono partizionati su più macchine.

Per gestire i Big Data sono state create tecnologie che riescono ad usare la potenza computazionale e la capacità di memorizzazione di un cluster, con un incremento prestazionale proporzionale al numero di macchine presenti sullo stesso. Il più utilizzato di questi sistemi è Hadoop, che offre un sistema per la memorizzazione e l'analisi distribuita dei dati. Grazie alla ridondanza dei dati ed a sofisticati algoritmi, Hadoop riesce a funzionare anche in caso di fallimento di uno o più macchine del cluster, in modo trasparente all'utente.

Su Hadoop si possono eseguire diverse applicazioni, tra cui MapReduce, Hive e Apache Spark. É su quest'ultima applicazione, nata per il data processing, che è maggiormente incentrato il progetto di tesi. Un modulo di Spark, chiamato Spark SQL, verrà posto in confronto ad Hive nella velocità e nella flessibilità nell'eseguire interrogazioni su database memorizzati sul filesystem distribuito di Hadoop.

Nel primo capitolo viene fatta una panoramica sui Big Data, descrivendone le loro caratteristiche e il loro ciclo di vita. Vengono poi descritte le tecnologie attualmente utilizzate per la loro memorizzazione, soffermandosi sulla più utilizzata, Hadoop, che offre un filesystem distribuito e un gestore di risorse attraverso i quali più applicazioni possono eseguire contemporaneamente computazioni sui dati. Viene descritta l'architettura di Hadoop e il modo in cui si effettuano letture e scritture in HDFS (Hadoop Distributed Filesystem), il suo filesystem distribuito.

Nel secondo capitolo viene introdotto Spark, un framework open-source

per l'analisi di dati che può essere eseguito su Hadoop. Esso è una creazione relativamente recente, nata come sostituta a MapReduce per eseguire query interattive e algoritmi iterativi. Viene descritta la sua architettura ed il suo modo di funzionamento in confronto a MapReduce, così come viene spiegata la sua interazione con il gestore di risorse di Hadoop. Spark offre un insieme di moduli fortemente interconnessi (Spark Stream, Spark SQL, MLlib e GraphX) che verranno brevemente introdotti.

Il terzo capitolo spiega brevemente le potenzialità dell'API Java di Spark, quella che permette di caricare ed elaborare dati per ottenere informazioni utili. Una caratteristica principale di tale API è che si può scegliere di tenere temporaneamente dei dataset in memoria per un efficiente riutilizzo futuro, operazione non permessa in MapReduce. É tale caratteristica che rende Spark efficiente sugli algoritmi iterativi e sull'analisi interattiva dei dati.

Nel quarto capitolo viene illustrato il modulo Spark SQL, il sottosistema che permette ad applicazioni di eseguire interrogazioni sui dati tramite una variante dell'SQL, chiamata HiveQL. Si possono interrogare dati strutturati e semistrutturati, tra cui i database Hive e i file JSON.

Noi utilizzeremo Spark SQL per eseguire interrogazioni su dei database Hive, sulla base di un famoso benchmark per i sistemi di Business Intelligence chiamato TCP-H. I tempi raccolti su sei tipi di database diversificati da dimensione e formato di memorizzazione saranno posti in confronto con i tempi ottenuti da Hive, che per eseguire le query utilizza dei job MapReduce. Vedremo come Spark, usando più efficientemente la memoria, riesce ad ottenere tempi di esecuzione nettamente migliori.

Capitolo 1

I Big Data e la piattaforma Hadoop

1.1 Cosa sono i Big Data

Il termine Big Data è un termine ampio ed in evoluzione che indica qualsiasi collezione di dati così ampia da rendere difficile o impossibile memorizzarla in un sistema software tradizionale, come lo è un DBMS (Database Management System) relazionale. Anche se il termine non si riferisce ad alcuna quantità in particolare, solitamente si inizia a parlare di Big Data a partire da Terabyte di dati, cioè quando i dati non possono essere più memorizzati o processati da una singola macchina.

1.2 Caratteristiche dei Big Data

I Big Data hanno numerose caratteristiche che li differenziano dalle tradizionali collezioni di dati. La più importante è il **volume**, cioè la quantità di dati che bisogna memorizzare. Si pensi ad esempio a Facebook, che memorizzava 300 PB di dati nel 2014, con 600 TB di nuovi dati da memorizzare ogni giorno.

Quando la quantità di dati eccede la capacità di un singolo disco o di un RAID di dischi, bisogna per forza utilizzare un cluster, in modo tale da distribuire i dati tra i vari nodi del cluster attraverso un filesystem distribuito.

Un'altra caratteristica dei Big Data è la **varietà**: i dati possono provenire da fonti diverse ed in diverse forme, ad esempio possono essere strutturati,

semi-strutturati o non-strutturati. Si pensi al testo di un tweet, alle immagini o ai dati provenienti dai sensori: ovviamente i dati sono di tipo diverso e integrarli tra loro richiede appositi sforzi.

I dati non strutturati non possono essere memorizzati in un RDMS, quindi li si memorizzano in appositi database NoSQL che riescono ad adattarsi con facilità alla variabilità dei dati. Gli RDMS richiedono invece che la struttura di un database sia fissata prima del suo utilizzo e rimanga invariata.

Nei giorni nostri i dati vengono generati da una quantità sempre maggiore di device e persone. Una percentuale sempre maggiore della popolazione ha accesso a Internet e ad uno Smartphone e si è in vista ad un'esplosione di sensori a causa dell'emergente Internet Of Things. Ciò fa sì che una grandissima quantità di dati deve essere memorizzata *rapidamente*. La **velocità** è infatti una terza caratteristica dei Big Data, che indica la rapidità con cui si rendono disponibili nuovi dati. Le tecnologie per la gestione di questo aspetto dei Big Data sono chiamate *streaming data* e *complex event processing*, che fanno in modo che i dati siano analizzati man mano che arrivano e non con un lunghissimo processo batch. Interrogativi a cui danno risposta tali sitemi sono del tipo *Quante volte è stata cercata la parola mela nell'ultimo giorno?*

Quelle finora descritte sono le 3 classiche caratteristiche dei Big Data, ma ultimamente ve ne sono aggiunte altre. La **variabilità** è una di queste, e si riferisce all'inconsistenza presente nei dati, che ostacola il processo di manipolazione e gestione efficace dei dati. La **complessità** indica invece il fatto che i dati provengono da fonti diverse e devono essere collegati tra loro per ricavare informazioni utili.

1.3 Acquisizione dei dati

Diversamente dai tradizionali sistemi, la tipologia e la quantità di fonti di dati sono molteplici. Non si ha più solo a che fare con dati strutturati, ma anche dati non strutturati provenienti da social network, da sensori, dal web, da smartphone, ecc..

L'acquisizione dei Big Data può avvenire in modi diversi, a seconda della fonte dei dati. I mezzi per l'acquisizione di dati possono essere suddivisi in quattro categorie:

- Application Programming Interface: le API sono protolli usati come interfaccia di comunicazione tra componenti software. Alcuni esempi di API sono la Twitter API, la Graph Facebook API e le API offrite da alcuni motori di ricerca come Google, Bing e Yahoo!. Esse permettono ad esempio di ottenere i tweet relativi a determinati argomenti (Twitter

API) o di esaminare i contenuti pubblicitari che rispondono a determinati criteri di ricerca nel caso della Graph API di Facebook.

- Web Scraping: si possono prendere dati semplicemente analizzando il Web, cioè la rete di pagine collegate tra loro tramite hyperlinks. Software come **Apache Tika** automatizzano il processo di lettura di dati e metadati contenuti in file HTML, XML, PDF, EPUB, file office, audio, immagini, JAR, ecc.. Tika riesce anche a determinare la lingua in cui è scritto un documento.

- Strumenti ETL: gli strumenti ETL (Extract, Transform, Load) sono quegli strumenti utilizzati in Data Warehousing per convertire i database da un formato o tipo ad un altro.

  L'applicazione ETL di punta per Hadoop è **Apache Sqoop**, che permette di caricare i dati strutturati presenti in un RDBMS in HDFS, Hive o HBase, così come permette di fare l'operazione inversa. Sono supportati nativamente i database HSQLSB, MySQL, Oracle, PosgreSQL, Netezza e Teradata. Sqoop offre una semplice interfaccia a linea di comando per la movimentazione dei dati.

- Stream di dati: sono disponibili tecnologie per la cattura e il traferimento di dati in tempo reale. Le tecnologie più diffuse sono **Apache Flume** e **Microsoft StreamInsight**

1.4 Gestione e memorizzazione

Il bisogno di un'elevata scalabilità e il fatto che c'è bisogno di memorizzare dati che potrebbero non essere strutturati fa sì che i tradizionali DBMS relazionali non siano adatti alla memorizzazione dei Big Data. Per questo motivo sono stati creati nuovi sistemi che permettono di memorizzare tipi di dati non relazionale offrendo scalabilità orizzontale, cioè le prestazioni aumentano in maniera lineare rispetto al numero di nodi presenti. Ciò si contrappone all'aumento di prestazioni di una singola macchina, operazione complessa e costosa.

Tra le tecnologie presenti per la memorizzazione dei Big Data, la più diffusa è Hadoop, un software open-source affidabile e scalabile per il calcolo distribuito. I software di calcolo distribuito sono stati progettati per sfruttare la potenza di calcolo e la memoria di un insieme di macchine, suddividendo il lavoro da svolgere tra le stesse.

Hadoop, per la memorizzazione dei dati, utilizza HDFS (Hadoop Distributed Filesystem), un filesystem distribuito che divide i file in blocchi, i quali

vengono distribuiti sui nodi di un cluster. Maggiori dettagli sul suo funzionamento saranno dati in seguito. Ovviamente il filesystem è stato pensato per funzionare in caso di rottura di un nodo di un cluster, accadimento molto comune in un grande cluster.

1.4.1 Database NoSQL

Un database NoSQL offre meccanismi per memorizzare i dati che potrebbero non essere relazionali, cioè essere memorizzabili su tabelle relazionali. Le strutture dei dati memorizzati nei sistemi NoSQL (ad esempio documenti, grafi o coppie chiave/valore) rendono alcune operazioni più veloci rispetto a generici database relazionali. Si utilizzano infatti diversi tipi di database NoSQL in base al problema da risolvere. Le tipologie di database NoSQL più utilizzate sono:

- **Key/Value Store:** i database di questo tipo permettono di memorizzare array associativi (o dizionari). In un array associativo si hanno un insieme di record che possono essere identificati sulla base di una chiave univoca. Viene spesso utilizzata una struttura chiamata *HashMap* per l'implementazione di questo tipo di DBMS, la quale garantisce tempi costanti per le comuni operazioni di inserimento, cancellazione, modifica e ricerca. Famosi DBMS di questo tipo sono **Dynamo**, **CouchDB** e **Redis**, che lavora in memoria per le massime prestazioni.

- **Document-oriented database:** questo tipo di database permette di memorizzare in maniera efficiente dati semistrutturati. Essi possono essere considerati una generalizzazione dei database key/value, nei quali vengono permesse strutture innestate. Sono spesso usati i formati JSON o XML per la memorizzazione dei dati. **MongoDB** è il più diffuso database orientato ai documenti ed è uno dei DBMS NoSQL più utilizzati.

- **Column-oriented database:** i DBMS colonnari, a differenza dei tradizionali RDBMS, che memorizzano i dati riga per riga, li memorizzano colonna per colonna. Per ogni colonna si memorizzano coppie chiave/valore, dove la chiave è l'identificativo di riga ed il valore è il valore associato a quella colonna per la specifica riga. Questa rappresentazione permette di risparmiare una notevole quantità di spazio in caso di sparsità dei dati.

  I DBMS di tipo colonnare più diffusi sono **HBase** e **Cassandra**. HBase è un DBMS scritto in Java ispirato a Google BigTable. Fa parte del progetto Hadoop e funziona memorizzando i dati su HDFS. La sua integrazione con il resto dell'ecosistema Hadoop è molto elevata, infatti può

essere fonte di dati per le applicazioni MapReduce o Spark. Cassandra invece può funzionare senza un'installazione di Hadoop e, a contrario di HBase, non ha un singolo punto di rottura.

- **Graph database:** un database a grafo utilizza un insieme di nodi con un insieme di archi che li connettono per memorizzare le informazioni. Essi sono molto utili per rappresentare le relazioni tra oggetti e tra i possibili utilizzi c'è la rappresentazione della rete delle amicizie in un social network o la rete dei collegamenti tra pagine Web. Alcuni graph database permettono di assegnare attributi o valori agli archi, ad esempio per rappresentare la lunghezza di una strada per poi calcolare il cammino minimo tra due nodi del grafo.

  Altri esempi di applicazioni di questo tipo di DBMS sono il calcolo delle componenti connesse/fortemente connesse, il numero di triangoli e il calcolo del PageRank, che permette di capire quali nodi sono più importanti in un grafo. Quest'ultimo algoritmo è alla base del funzionamento di Google.

I database NoSQL devono rispettare il **teorema CAP** (Consistency, Availability, Partition Tollerance), anche conosciuto come teorema di Brewer, che afferma che un sistema distribuito non può garantire simultaneamente consistenza, disponibilità e tolleranza di partizione. Queste tre proprietà vengono ora descritte in dettaglio:

- **Consistenza:** si intende che tutti i nodi vedono la stessa versione dei dati in uno stesso momento.
- **Disponibilità:** le letture e le scritture hanno sempre successo, a meno che non si sia contattato un nodo fuori servizio
- **Tolleranza di partizione:** la garanzia che il sistema funziona anche quando si aggiungono o rimuovono nodi dal sistema, ad esempio a causa di problemi di rete.

Per fare un esempio HBase garantisce consistenza e tolleranza di partizione, mentre Cassandra garantisce disponibilità a discapito della garanzia di consistenza.

1.5 Analisi e processamento dei dati

L'avvento dei Big Data ha portato un enorme mole di dati da analizzare per estrapolare informazioni utili alle organizzazione. I dati possono essere ad

esempio utilizzati per capire le preferenze degli utenti e studiarne il comportamento, in modo da fare proposte su eventuali acquisti e aumentare il profitto di un online shop. Alternativamente si può analizzare il grafo delle amicizie di un social network cercando di capire quali sono le probabili conoscenze di una persona tra quelle non indicate nel grafo. I tool più diffusi per l'analisi dei dati nell'ambito Big Data sono:

- Pig
- Hive
- Spark
- Impala
- Mahout
- Presto
- R
- Drill

Alcuni di questi strumenti, tra cui Spark, Mahout e R offrono algoritmi avanzati di machine learning, text-mining, analisi di grafi o statistica avanzata. Dato che tali algoritmi dovranno essere eseguiti su un cluster, sono offerti solo algoritmi altamente parallelizzabili. Ci sono infatti numerosi algoritmi utili che non sono scalabili in un cluster.

1.6 Introduzione ad Hadoop

Apache Hadoop è un framework open-source per la memorizzazione e l'analisi distribuita di grandi quantità di dati. Fu creato nel 2005 da Doug Cutting e Mike Cafarella, originariamente per il progetto *Nutch*, un motore di ricerca basato su Lucene e Java implementato utilizzando MapReduce e un filesystem distribuito, i quali furono successivamente divisi dal progetto.

MapReduce e HDFS (Hadoop Distributed File System) furono ispirati da dei paper pubblicati da Google sul funzionamento dei loro MapReduce e Google File System. Hadoop, che prende il nome da un elefante peluche del figlio di Doug, è attualmente un progetto top-level di Apache usato e sviluppato da una grande community di contributori e utenti. Pubblicato con licenza open-source Apache 2.0, esso è utilizzato dalle più grandi aziende informatiche tra cui Yahoo!, Facebook, Adobe, EBay, IBM, LinkedIn e Twitter. Attualmente

Hadoop è alla versione 2 la quale, rispetto alla prima versione, consente ad applicazioni diverse da MapReduce di sfruttare le potenzialità del framework.

Prima di Hadoop le elaborazioni dei dati erano svolte da sistemi di **High Performance Computing** e Grid Computing. Hadoop però offre un insieme di librerie facili da utilizzare e sfrutta la relicazione dei dati sui singoli nodi per migliorare i tempi di accesso ai dati, evitando se possibile di trasferirli in rete. Le interrogazioni su Hadoop richiedono spesso la lettura di una grande quantità di dati, a differenza delle interrogazioni su sistemi tradizionali dove spesso è richiesta la lettura di un singolo o di pochi record di una tabella.

1.7 Caratteristiche di Hadoop

Il framework Hadoop include i seguenti moduli:

- **Hadoop Common:** fornisce funzionalità utilizzate da tutti gli altri moduli del progetto.
- **Hadoop Distributed File System (HDFS):** un filesystem distribuito che memorizza i dati su *commodity hardware*, fornendo una banda aggregata molto alta. É necessario utilizzare un filesystem distribuito in quando la quantità di dati da memorizzare può eccedere quella memorizzabile in una singola macchina. É quindi necessario partizionare i dati su più macchine, fornendo meccanismi di recupero nel caso in cui una macchina si rompa. Rispetto ai normali filesystem, quelli distribuiti richiedono comunicazione di rete e sono più complessi da sviluppare.
- **Hadoop YARN:** il gestore di risorse introdotto con la versione 2.0 di Hadoop, responsabile di gestire le risorse computazionali del cluster e assegnarle alle applicazioni.
- **Hadoop MapReduce:** un modello di programmazione per l'elaborazione di grandi quantità di dati, realizzato per far sì che i calcoli siano altamente parallelizzabili. Esso si basa sul principio "divide et impera", nel quale un problema corposo viene suddiviso in sottoproblemi indipendenti che possono essere risolti parallelamente. Il risultato finale dipende da qualche forma di aggregazione dei risultati parziali.

Solitamente con il termine Hadoop ci si riferisce all'intero ecosistema Hadoop che, oltre ai moduli sopra citati, comprende sistemi software che possono essere installati e funzionare su Hadoop come Apache HBase, Apache Hive, Apache Spark, ecc..

Il sistema Hadoop è un sistema altamente affidabile, in quanto può funzionare su cluster di commodity hardware ed è stato progettato per continuare a funzionare anche in caso di problemi su uno o più nodi del cluster. Il sistema è altamente scalabile in quanto si possono aggiungere o rimuovere nodi al cluster su necessità. Per esempio il cluster di Yahoo ha più di 4500 nodi di computer con 16 GB di RAM, 4 TB di Hard Disk e 8 core [1].

In un cluster Hadoop non tutti i nodi sono uguali, in quanto viene usata un'**architettura master/slave**. Tra i nodi master vi sono quelli che eseguono il ResourceManager, il NameNode e l'ApplicationMaster, mentre tra gli slave vi sono i NodeManager e i Datanode. Tali concetti verranno ripresi in seguito nel capitolo.

1.8 Hadoop Distributed Filesystem

L'**Hadoop Distributed Filesystem** è un filesystem distribuito scritto in Java progettato per essere eseguito su commodity hardware, in cui i dati memorizzati vengono partizionati e replicati sui nodi di un cluster. HDFS è fault-tolerant e sviluppato per essere distribuito su macchine a basso costo. Un'importante caratteristica è il poter funzionare su grandi cluster. Malgrado usi un'architettura master/slave (nella quale il master può essere un collo di bottiglia), è stato provato con buoni risultati su cluster con migliaia di nodi.

Il filesystem fornisce un'alta banda di accesso ai dati ed è per questo adatto ad applicazioni che lavorano su grandi dataset. Il filesystem è progettato per quelle applicazioni che accedono a interi dataset o a gran parte di essi, dato che il tempo di lettura di un dataset può essere trascurabile rispetto alla latenza necessaria per accedere ad un singolo record. L'HDFS non rispetta lo standard POSIX per favorire le performance ed a causa del diverso utilizzo che si ha rispetto ai normali filesystem.

HDFS fu inizialmente sviluppato per il progetto Nutch, per poi essere diviso e diventare un sottoprogetto di Apache Hadoop.

1.8.1 Architettura

HDFS ha un'architettura master/slave [2]. In un cluster HDFS vi è un singolo NameNode (il master), un server che gestisce il namespace del filesystem e regola l'accesso ai file da parte dei client. I file sono memorizzati separatamente nei DataNode, di cui solitamente ce n'è uno per nodo del cluster. Internamente un file è spezzato in uno o più **blocchi** (solitamente grandi 128 MB ciascuno) e questi blocchi sono memorizzati in un insieme di DataNode (Figura 1.1).

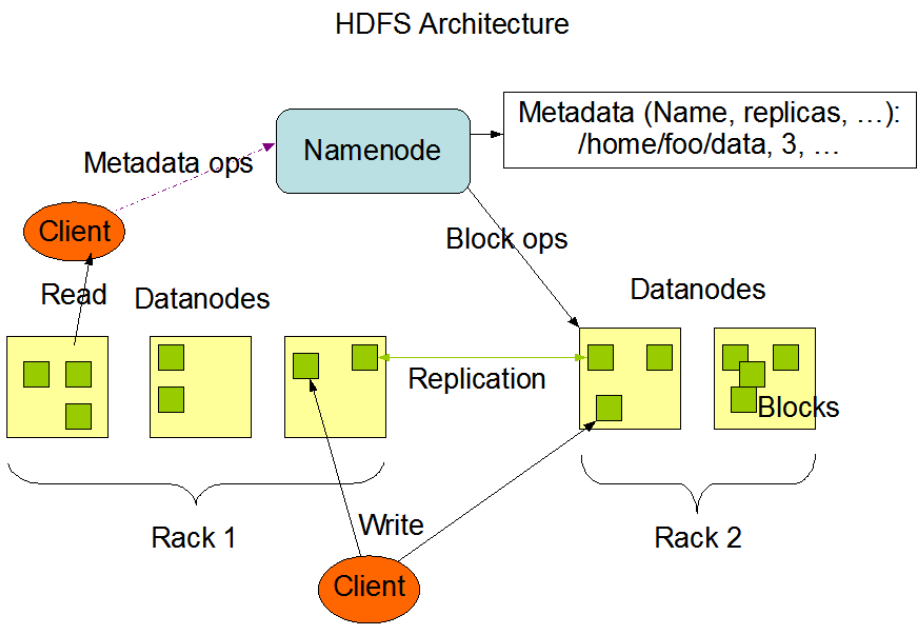

Figura 1.1: Architettura di HDFS

A differenza di molti filesystem, se un file è più piccolo della dimensione di un blocco non viene allocato lo spazio per un blocco intero, risparmiando spazio utilizzabile da altri file. Ogni blocco di file viene **replicato** tra più nodi del cluster per fornire tolleranza ai guasti e disponibilità, e per questo non è richiesto l'utilizzo di architetture RAID. Solitamente si tengono 3 repliche per ogni blocco di file, con possibilmente una in un rack diverso da quello degli altri blocchi.

Il NameNode, che gestisce lo spazio dei nomi (namespace) del filesystem, cioè la struttura gerarchica di file e cartelle, memorizza per ogni blocco di file l'insieme dei DataNode che lo contiene. Tale mappatura tra blocco e DataNode viene ricostruita periodicamente e ad ogni riavvio del sistema.

Il NameNode esegue operazioni come apertura, chiusura e rinominazione di file e cartelle, così come decide su quali DataNode memorizzare i blocchi dei file. Se il nodo che lo contiene cade, non si ha più accesso al filesystem in quanto non sarebbe possibile sapere in quali nodi del cluster sono presenti i vari blocchi di un file. Hadoop ha per questo ideato due modi per migliorare la situazione, dato che come gli altri nodi anche un nodo master prima o poi si guasterà. Il primo modo è fare il **backup** dei file che compongono lo stato persistente dei metadati del filesystem. Il secondo prevede l'utilizzo di un *Secondary Namenode* che periodicamente si aggiorna sulle modifiche apportate

al filesystem leggendo i file di log delle modifiche. In caso di fallimento del NameNode principale, il secondario può prendere il suo posto, ma potrebbe esserci della perdita di dati in quanto il suo stato non è sempre perfettamente allineato. L'esistenza di un singolo NameNode semplifica notevolmente l'architettura del sistema, però fa sì che il suo fallimento renda inaccessibile l'intero filesystem fino al suo ripristino. Il NameNode contiene solo i metadati e il sistema è progettato in modo tale che i dati non passino attraverso lui.

I DataNode sono invece responsabili di servire le richieste di lettura e scrittura da parte dei client del filesystem. I DataNode fanno le operazioni di creazione, cancellazione e replicazione dei dati su richiesta del NameNode. Solitamente c'è un DataNode per ogni nodo del cluster, i quali sono raggruppati in *rack*. HDFS sfrutta la gerarchia del cluster per una replicazione intelligente dei dati.

HDFS supporta una tradizionale organizzazione gerarchica dei file. A differenza dei filesystem di Linux, HDFS non supporta né i soft link che gli hard link. L'implementazione del filesystem non preclude però una futura implementazione di tali funzionalità.

Replicazione dei dati

L'HDFS è progettato per memorizzare grandi file in macchine facenti parte di un cluster. La memorizzazione di un file, che da solo potrebbe non entrare in un singolo nodo (HDFS non impone un limite esplicito alla dimensione dei file), viene spezzato in blocchi che sono replicati una certa quantità di volte configurabile (di default 3). Tutti i blocchi di un file, tranne l'ultimo, sono della stezza grandezza, di default 128 MB. Il NameNode decide dove piazzare le repliche di ogni blocco di file. Periodicamente riceve un *Heartbeat* e un *Blockreport* da parte dei DataNode, così saprà che il DataNode funziona correttamente e saprà quali blocchi esso contiene (Figura 1.2). La politica di replicazione è critica per l'affidabilità e per la performance di HDFS. Si utilizzano quindi informazioni sulla posizione dei nodi per decidere dove posizionare le repliche, in modo da garantire un buon livello di affidabilità e di utilizzo di banda nel cluster.

Siccome la comunicazione tra rack distinti è più lenta della comunicazione nello stesso rack, Hadoop non piazza le repliche in rack unici e distinti. Ciò permetterebbe di accedere ai dati anche in caso di fallimento di interi rack, ma non ottimizzerebbe la banda di rete utilizzata durante la scrittura di un blocco sul filesystem.

Per il caso comune, cioè con il fattore di replicazione impostato a 3, HDFS piazza una replica in un rack, la seconda replica in un diverso nodo dello stesso rack e l'ultima replica in un nodo facente parte di un rack diverso. Siccome

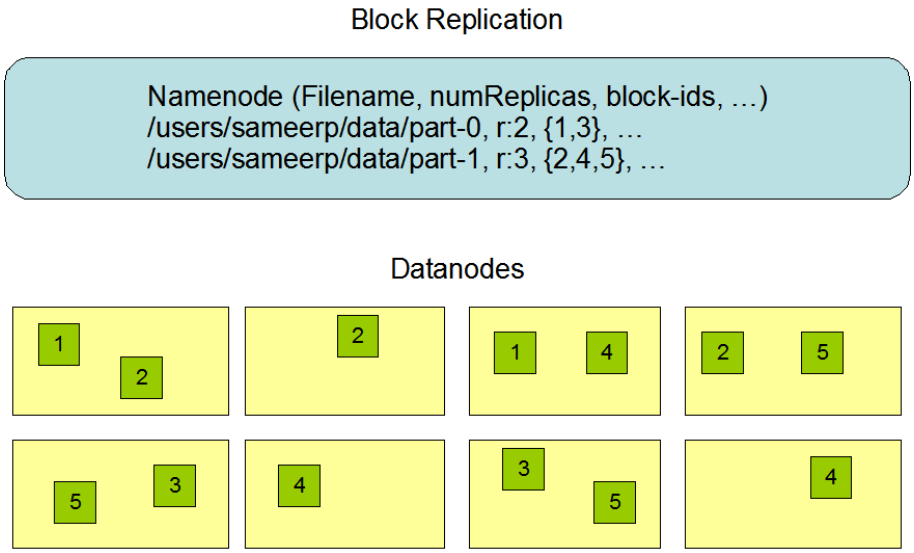

Figura 1.2: Replicazione dei blocchi

la probabilità di fallimento di un intero rack è molto più bassa di quella di fallimento di un nodo, tale comportamento non compromette l'affidabilità del sistema pur ottenendo buone prestazioni in lettura e scrittura.

Nel caso in cui si utilizzano più di 3 repliche, le ulteriori repliche vengono distribuite uniformemente tra i rack.

Quando si vogliono leggere dati dal filesystem, HDFS prova a far soddisfare la richiesta di lettura dal nodo che contiene la replica più vicina al client, per minimizzare la latenza e la banda di rete totale utilizzata.

1.8.2 Lettura e scrittura

Quando un client apre un file, prende la lista dei blocchi e le posizioni di essi nel cluster dal NameNode. Le posizioni vengono restituite ordinate per distanza dal client. Il client prova così a connettersi al DataNode più vicino, ed in caso di fallimento ne prova uno più lontano e così via. Il fallimento potrebbe essere dovuto a varie cause, tra cui il fatto che il DataNode potrebbe non essere raggiungibile, il DataNode non contenga la replica cercata o la replica sia corrotta (il checksum non corrisponde). Più client possono leggere lo stesso file, ma solo uno può scriverlo. Si può leggere un file aperto in scrittura, ed in tal caso il client chiede la lunghezza dell'ultima replica prima di leggere il suo contenuto.

Oltre a leggerli, si possono ovviamente aggiungere file nel filesystem HDFS e scriverci dentro. Però, una volta che il file è scritto, esso non può essere alterato, se non per il fatto che è possibile aggiungere dati in fondo (operazione *append*). Solo un client alla volta può scrivere in un file. Quando si apre un file in scrittura si ottiene una **licenza** per il file: nessun altro client può scriverci finchè non la si rilascia. Il client deve periodicamente rinnovare tale licenza mandando un heartbeat al NameNode: la licenza ha un limite temporale da non far scadere altrimenti un altro client può impadronirsene. Quando un file viene chiuso, la licenza è automaticamente revocata. La licenza riguarda solo la scrittura del file, infatti non vieta ad altri client di leggere il file concorrentemente.

Un file HDFS è formato da blocchi. Quando si vuole scrivere un nuovo blocco, il NameNode gli assegna un ID univoco e determina una lista di DataNode per ospitarlo, ordinati in modo da minimizzare la distanza totale tra di essi. Tali nodi formano una conduttura (pipeline) nella quale i byte sono trasferiti da un nodo della lista al successivo tramite pacchetti di grandezza tipica di 64 KB. Nuovi pacchetti possono essere trasmessi prima di ricevere l'acknowledgment per i pacchetti precedenti. Il numero di pacchetti trasmessi senza acknowledgment è limitata dal client.

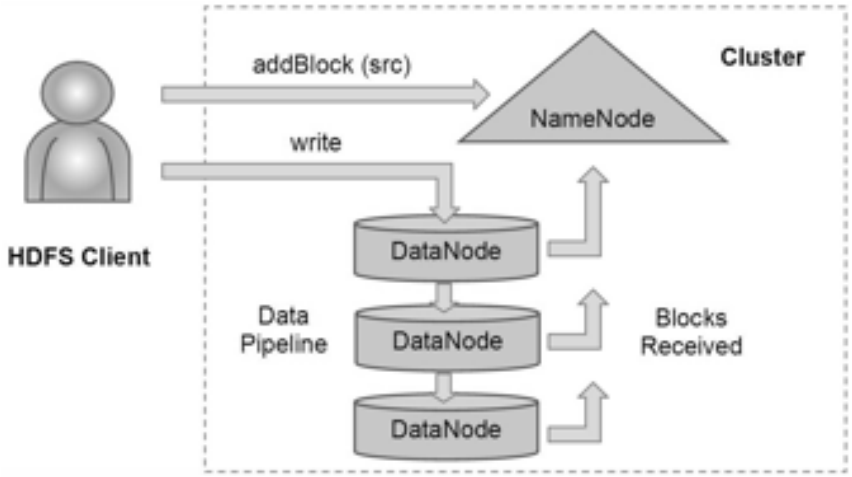

Figura 1.3: Creazione file

Dopo che i dati sono stati scritti, viene garantita la loro visibilità solo alla chiusura del file o alla chiamata di un apposito metodo di *flush*.

HDFS memorizza un **checksum** per ogni blocco di file, in modo tale che un client che legge dal filesystem distribuito possa rilevare la corruzione dei dati, causata dal DataNode, dalla rete o dal client stesso. Quando un client

crea un file HDFS, esso calcola un checksum per ogni blocco e lo manda ai DataNode assieme ai dati. I DataNode memorizzano tali checksum in un apposito repository e quando un client richiede un blocco, il rispettivo checksum viene trasmesso con esso, così che il client lo possa verificare. Se il checksum non corrisponde ai dati, il client notifica il NameNode della replica corrotta e prende la replica da un altro nodo del cluster.

1.9 YARN

YARN (Yet Another Resource Negotiator) è una caratteristica principale della seconda versione di Hadoop. Prima di YARN, uno stesso nodo del cluster, su cui stava in esecuzione il JobTracker, si occupava sia della gestione delle risorse del cluster sia dello scheduling dei task delle applicazioni MapReduce (che erano le uniche possibili). Con l'avvento di YARN i due compiti sono stati separati e sono svolti rispettivamente dal ResourceManager e dall'ApplicationMaster. Inoltre i TaskTracker presenti nei nodi del cluster per svolgere le operazioni di MapReduce sono stati sostituiti da NodeManager, che si occupano di lanciare e monitorare i container, cioè quei componenti che svolgono lavori specifici e a cui sono allocati una certa quantità di risorse del nodo (RAM, CPU, ecc.).

YARN permette di eseguire applicazioni diverse da MapReduce, tra cui Spark, Tez o Impala, così è possibile fare stream processing ed eseguire query interattive mentre si fa batch processing. YARN permette a più utenti di connettersi al cluster e lanciare applicazioni diverse concorrentemente.

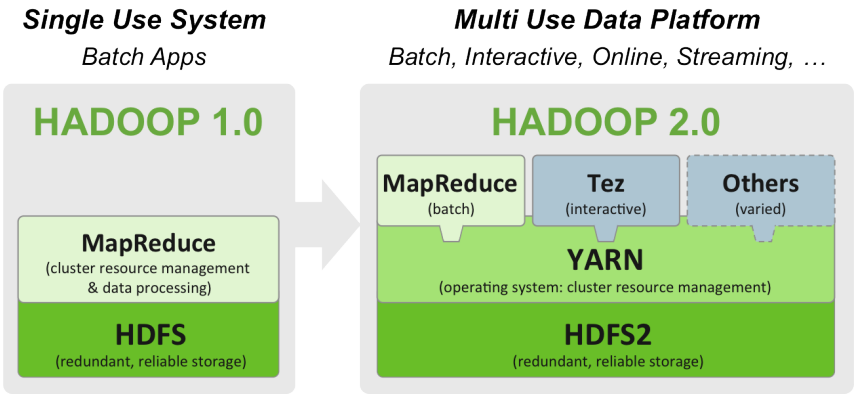

Figura 1.4: Hadoop 1.x vs Hadoop 2.x

Il ResourceManager di YARN è l'entità che governa il cluster decidendo l'allocazione delle risorse alle applicazioni concorrenti in esecuzione. Le risorse vengono richieste dall'ApplicationMaster, il primo container allocato per un'applicazione. Esso comunica con i NodeManager per inizializzare i container e monitorare la loro esecuzione. Il NodeManager, l'entità che sta in ogni nodo del cluster, si occupa di creare e distruggere container, così come monitora le risorse utilizzate in un nodo e controllarne la sua *salute*.

Il ciclo di vita di un'applicazione è il seguente:

- arriva una richiesta da un client per l'esecuzione di un'applicazione
- il ResourceManager crea un container per l'ApplicationMaster
- l'ApplicationMaster negozia con il ResourceManager le risorse per i container
- viene eseguita l'applicazione, mentre l'ApplicationMaster ne monitora la corretta esecuzione
- alla conclusione dell'applicazione, l'ApplicationMaster libera le risorse comunicando con il ResourceManager

É da notare che è l'ApplicationMaster a garantire la fault-tollerance di un'applicazione, monitorando lo stato dei container e richiedere nuovi container al ResourceManager se necessario. Ciò fa sì che il nodo contenente il ResourceManager non sia sovraccaricato e permette di avere una scalabilità superiore.

1.10 MapReduce

MapReduce è un modello di programmazione per processare grandi dataset su sistemi di calcolo paralleli. Un Job MapReduce è definito da

- i dati di input
- una procedura **Map**, che per ogni elemento di input genera un certo numero di coppie chiave/valore
- una fase di *shuffle* in rete
- una procedura **Reduce**, che riceve in ingresso elementi con la stessa chiave e genera un informazione riassuntiva da tali elementi
- i dati di output

MapReduce garantisce che tutti gli elementi con la stessa chiave saranno processati dallo stesso reducer, dato che i mapper usano tutti la stessa funzione hash per decidere a quale reducer mandare le coppie chiave/valore. Questo paradigma di programmazione è molto complicato da usare direttamente, dato il numero di job necessari per svolgere operazioni complesse sui dati. Sono stati per questo creati tool come Pig e Hive che offrono un linguaggio di alto livello (Pig Latin e HiveQL) e trasformano le interrogazioni in una serie di job MapReduce che vengono eseguiti in successione.

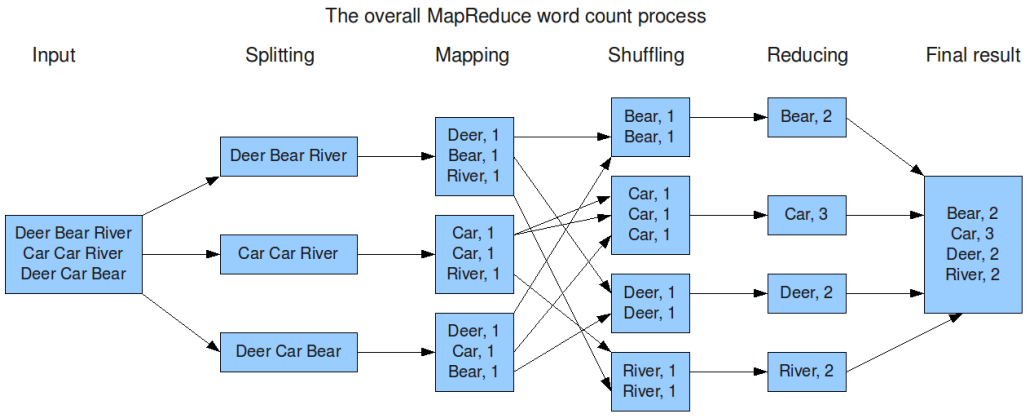

Figura 1.5: Funzionamento di MapReduce

1.11 Spark

Apache Spark è un progetto che diversamente ad Hadoop MapReduce non obbliga l'utilizzo del disco fisso, ma può effettuare operazioni direttamente in memoria centrale riuscendo ad offrire prestazioni anche 100 volte superiori su applicazioni specifiche. Spark offre un insieme di primitive più ampio rispetto a MapReduce, semplificando di molto la programmazione. Questo framework verrà approfondito più in dettagli nei prossimi capitoli.

1.12 L'ecosistema Hadoop

Al nucleo centrale di Hadoop si aggiungono un insieme di progetti correlati, anch'essi sviluppati da parte dell'Apache Foundation, che vanno a completare quello che viene definito l'**ecosistema Hadoop** (Figura 1.6). Tali progetti,

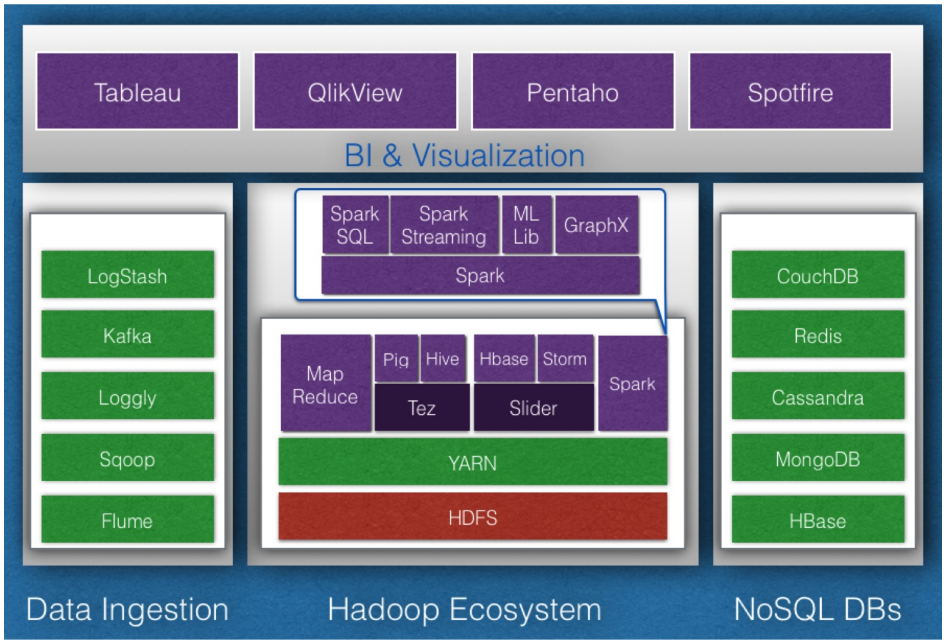

Figura 1.6: Ecosistema Hadoop

non essenziali per il funzionamento di Hadoop, sono:

- **Ambari:** uno strumento intuitivo e facile da usare che, attraverso un'interfaccia web, permette di amministrare un cluster Hadoop. Ad esempio permette di lanciare un sistema Hadoop da zero e di monitorarne lo stato di salute tramite una dashboard.
- **Avro:** un sistema per serializzare i dati
- **Cassandra:** un database NoSQL distribuito, creato da Facebook. MapReduce e Spark possono leggere da questo database, che memorizza i dati colonna per colonna. Tale database non ha un singolo punto di rottura ma non garantisce la consistenza dei dati.
- **HBase:** un DBMS non relazionale ispirato a Google BigTable, scritto in Java. A contrario di Cassandra, questo DBMS offre consistenza ma ha

un singolo punto di rottura. HBase è molto usato nel mondo di Hadoop, offrendo buone prestazioni. Esso è usato da Facebook per implementare la chat, rimpiazzando Cassandra.

- **Hive:** un sistema di data warehouse che facilita l'esecuzione di query e la gestione di grandi dataset distribuiti. Hive fornisce un meccanismo per definire la struttura dei dati e permette di interrogarli tramite un linguaggio SQL-like chiamato HiveQL. Tali query verranno trasformate in una serie di job MapReduce per essere eseguite.

- **Mahout:** una libreria per il machine learning. Essa è specializzata nel collaborative filtering, nel clustering e nella classificazione. Ovviamente vengono forniti solo quegli algoritmi che possono essere altamente parallelizzabili.

- **Pig:** una piattaforma per analizzare grandi quantità di dati che consiste in un linguaggio di alto livello, chiamato Pig Latin, per scrivere programmi di elaborazione e analisi. L'infrastruttura di Pig consiste in un compilatore che produce una sequenza di job MapReduce. Pig è quindi simile ad Hive per il suo utilizzo, ma offre un linguaggio differente.

- **Zookeeper:** uno strumento per sincronizzare gli oggetti comuni nel cluster, come ad esempio i file di configurazione. Zookeeper offre quei servizi che vengono spesso erroneamente non implementati per la loro complessità di implementazione.

Capitolo 2

Il sistema Spark

2.1 Introduzione

Spark è un framework open-source per l'analisi di grandi quantità di dati su cluster, nato per essere veloce e flessibile. Caratterizzato dalla capacità di memorizzare risultati (solitamente parziali) in memoria centrale, si offre come valida alternativa a MapReduce, il quale memorizza obbligatoriamente i risultati delle computazioni su disco.

L'utilizzo ottimale della memoria permette a Spark di essere ordini di grandezza più veloce, rispetto a MapReduce, nell'esecuzione di algoritmi iterativi, cioè quegli algoritmi che svolgono iterativamente le stesse istruzioni su dei dati fino a che non si verifichi una certa condizione. Ciò non toglie che Spark rimane una valida alternativa, probabilmente più flessibile e facile da utilizzare, anche quando si vuole o si deve utilizzare il disco fisso, ad esempio perchè i dati non entrano tutti in memoria. Esso offre infatti un'API molto più semplice da utilizzare rispetto al semplice paradigma MapReduce. Tale API verrà brevemente illustrata nel capitolo successivo.

Entrambi i framework possono funzionare su YARN, il gestore di risorse della piattaforma Hadoop. Spark può però funzionare su una varietà di gestori di risorse, tra cui uno offerto da Spark stesso (Standalone cluster manager) e Apache Mesos.

Si possono leggere dati da una moltitudine di fonti, tra cui HDFS (il file system distribuito di Hadoop), Amazon S3, Cassandra, HBase, ecc., così come sono supportati numerosi formati di file strutturati, semi-strutturati o non strutturati.

Spark può essere usato nei linguaggi di programmazione Java, Scala e Python, ed è prevista un'integrazione con il linguaggio R nel prossimo futuro.

Quest'ultimo linguaggio è infatti molto comodo per svolgere analisi di tipo statistico. L'utilizzo di linguaggi di scripting (Scala e Python) permette un'esplorazione interattiva dei dati: si possono interrogare i dati più volte senza rileggerli da disco ad ogni interrogazione.

2.1.1 Chi usa Spark

Spark è attualmente usato da più di 500 organizzazioni mondiali, tra cui eBay Inc, IBM, NASA, Samsung e Yahoo!. Un sondaggio indica che il 13% dei data scientists usavano Spark nel 2014, con un ulteriore 20% intenzionati ad usarlo nel 2015 e il 31% che lo stavano prendendo in considerazione [3]. Spark, date le sue potenzialità e la sua semplicità di utilizzo, sta quindi crescendo esponenzialmente. Con i suoi 465 contributori, esso fu il progetto di Big Data open source più attivo del 2014.

2.1.2 Storia di Spark

Spark fu creato nel 2009 nell'UC Berkeley AMPLab come progetto di ricerca, da ricercatori che lavorando su MapReduce si resero conto dei suoi limiti con gli algoritmi iterativi e con le interrogazioni interattive. Spark è scritto principalmente in Scala ed è stato progettato fin dall'inizio per questi tipi di utilizzo. Già dai primi benchmark si notava un aumento di prestazioni superiore a 10 volte su alcune applicazioni. Nel 2010 il progetto fu reso open source con licenza BSD e nel 2013 fu donato all'Apache Software Foundation, dove fu cambiata la licenza in Apache 2.0. Nel 2014 diventò uno dei progetti principali dell'Apache Software Foundation e dei Big Data, con Intel, Yahoo! e Databricks tra i suoi più grandi contributori. Spark segna la storia per aver ordinato 100 TB di dati in un terzo del tempo rispetto a MapReduce, usando un decimo delle macchine [4].

2.2 Caratteristiche di Apache Spark

Nel suo core, Spark offre un motore di computazione che viene sfruttato da tutti gli altri suoi componenti. Ciò significa che un'ottimizzazione al core di Spark si manifesta su tutte le applicazioni che sfruttano questo framework. Il core di Spark si occupa delle funzionalità base, come lo scheduling, la distribuzione e il controllo dell'esecuzione dell'applicazione utente. A sfruttare il core ci sono vari tools (Figura 2.1):

- Spark SQL

- Spark Streaming
- MLlib
- GraphX

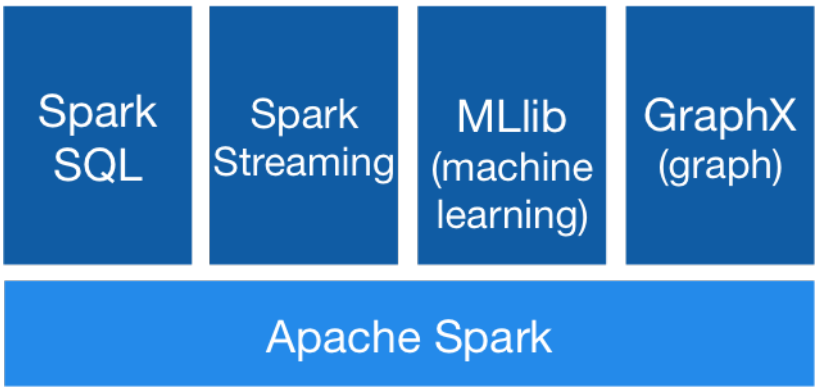

Figura 2.1: Lo stack di Spark

Essi si integrano perfettamente tra di loro, facendo sì che una stessa applicazione possa usare tutti i componenti di Spark contemporaneamente. Il fatto che Spark offra così tante funzionalità fa sì che il tempo di installazione, di testing e il tempo per imparare un nuovo tool e integrarlo con il resto dello stack sia ridotto rispetto all'uso di tanti software differenti e indipendenti. In più, se verranno aggiunte funzionalità, esse saranno disponibili semplicemente aggiornando Spark. Tutti i moduli di Spark sono di seguito descritti:

- **Spark Core:** il core di Spark contiene le funzionalità base di Spark, tra cui la gestione della memoria, della rete, dello scheduling, il recupero dal fallimento di un nodo del cluster (vengono ricalcolati solo i dati persi), ecc..

  Il core di Spark fornisce un'API che permette di gestire i cosiddetti Resilient Distributed Datasets (RDDs), che rappresentano dati distribuiti sul cluster e su cui vengono svolte operazioni di trasformazione per poi recuperare i dati modificati.

- **Spark SQL:** Spark SQL permette di fare interrogazioni su dati strutturati e semistrutturati usando il linguaggio HiveQL, una variante del famoso linguaggio SQL. Si possono interrogare file JSON, file di testo e qualunque formato di file supportato da Hive, tra cui Parquet, ORC,

Avro, ecc.. Spark SQL si integra perfettamente con il resto del sistema Spark, infatti i risultati delle query possono essere trasformati e analizzati usando l'API di Spark che funziona sugli RDD, così come è possibile analizzare dati con HiveQL dopo avergli associato uno schema tabellare.

- **Spark Streaming:** Spark Streaming permette di analizzare flussi di dati in tempo reale, ad esempio log di errori o un flusso di tweet. Anche questo componente si integra perfettamente con il resto. É ad esempio possibile fare il join tra il flusso di dati e un database storico in tempo reale. I flussi di stream possono arrivare da fonti come Apache Flume, Kafka, o HDFS, e vengono ingeriti in piccoli *lotti* per essere analizzati. Si possono svolgere tutte le operazioni fattibili sui dati statici, ma in più sono presenti funzionalità specifiche relative al tempo come le *sliding windows*.

- **MLlib:** Spark MLlib è una libreria di machine learning altamente ottimizzata, che sfrutta il fatto che i dati di Spark possono essere memorizzati in memoria. Molti degli algoritmi di machine learning sono infatti iterativi. Ovviamente MLlib offre la possibilità di usare solo gli algoritmi che sono per loro natura parallelizzabili, tra cui la regressione lineare, il K-means, le foreste casuali, ecc.. É possibile usare gli stream di Spark Streaming per il training di algoritmi di machine learning.

- **GraphX:** GraphX è una libreria per l'analisi di grafi talmente grandi che non potrebbero essere analizzati su una singola macchina (ad esempio i grafi dei social network). Un grafo è una collezione di nodi (o vertici) collegati da archi. Ad esempio i nodi possono essere le persone e gli archi le amicizie. La libreria offre algoritmi come PageRank (per misurare quanto è "importante" ogni nodo di un grafo), il calcolo delle componenti connesse, il calcolo dei triangoli, ecc..

  GraphX unifica ETL, analisi esplorativa e calcolo iterativo su grafi in un unico sistema. I grafi sono gestiti come gli altri dataset, con cui si possono addirittura fare dei join. Tramite l'API Pregel si possono scrivere algoritmi iterativi custom per l'analisi di grafi.

2.3 Architettura

Spark può lavorare sia in un singolo nodo che in cluster.

Nel caso generale vi sono una serie di processi in esecuzione per ogni applicazione Spark: un driver e molteplici executor. Il driver è colui che gestisce

l'esecuzione di un programma Spark, decidendo i compiti da far svolgere ai processi executor, i quali sono in esecuzione nel cluster. Il driver, invece, potrebbe essere in esecuzione nella macchina client.

Nel programma principale di un'applicazione Spark (il programma *driver*) è presente un oggetto di tipo SparkContext, la cui istanza comunica con il gestore di risorse del cluster per richiedere un insieme di risorse (RAM, core, ecc.) per gli *executor* (Figura 2.2). Sono supportati diversi cluster manager tra cui YARN, Mesos, EC2 e lo Standalone cluster manager di Spark.

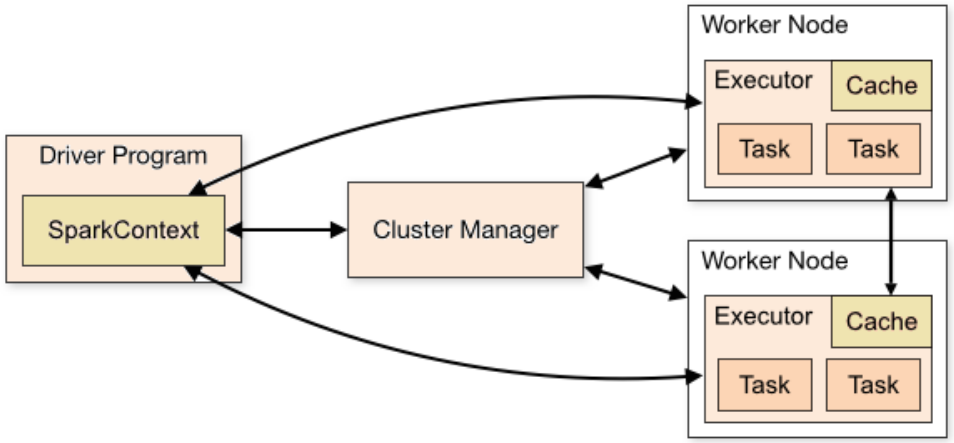

Figura 2.2: La comunicazione tra il driver e il cluster

Viene usata un'architettura di tipo master/slave, dove è presente un processo coordinatore e tanti processi worker. Per ogni applicazione Spark ci sarà in esecuzione un driver (il coordinatore) e tandi processi executor (i worker). Dato che ogni esecutore è in un processo separato, applicazioni diverse non possono condividere dati se non scrivendoli prima su disco.

Se si lavora in un singolo nodo si ha invece un solo processo che contiene sia il driver che un executor, ma questo è un caso particolare. Lavorare in un singolo nodo permette di fare il testing delle applicazioni, dato che si usa la stessa API che si utilizzerebbe se si lavorasse in un cluster.

Un'applicazione Spark si compone di **jobs**, uno per ogni azione. Si ha cioè un job ogni qualvolta si vuole ottenere dal sistema il risultato di una computazione. Ogni job è composto da un insieme di **stage** che dipendono l'un l'altro, svolti in sequenza, ognuno dei quali viene eseguito da una multitudine di **task**, svolti parallelamente dagli executor.

2.3.1 Il driver

Il driver è il processo principale, quello in cui è presente il metodo main contenente il codice utente. Il codice, che contiene operazioni di traformazione e azioni sugli RDD (i dataset distribuiti), dovrà essere eseguito in parallelo dai processi executor distribuiti nel cluster. Il driver può essere eseguito sia all'interno del cluster che nella macchina client che manda in esecuzione l'applicazione Spark.

Esso svolge le seguenti due funzioni:

- convertire il programma utente in un insieme di task, cioè la più piccola unità di lavoro in Spark. Ogni programma Spark è strutturato in questo modo: si leggono dati da disco in uno o più RDD, li si trasformano e si recupera il risultato della computazione. Le operazioni di trasformazione vengono effettuate solo nel momento del bisogno, cioè quando si richiede un risultato. In memoria infatti Spark memorizza un grafo aciclico diretto (DAG) delle operazioni da fare per ottenere il contenuto di un RDD. Le operazioni di trasformazione o di salvataggio/recupero di dati vengono trasformate in una serie di stage eseguiti in sequenza, ognuno dei quali è composto da un insieme di task che vengono eseguiti dagli executor.

- fare lo scheduling dei task sui nodi executor. Lo scheduling dei task viene fatto in base a dove sono memorizzati i file, per evitare il più possibile di trasferirli in rete. Se un nodo fallisce, lo scheduling in un altro nodo viene fatto automaticamente dalla piattaforma, e si ricalcolano solo i dati persi.

2.3.2 Gli executor

Gli executor (esecutori) sono i processi che svolgono i compiti (tasks) dettati dal driver. Ogni applicazione ha i propri executor (cioè i propri processi), ognuno dei quali può avere più thread in esecuzione. Gli executor hanno una certa quantità di memoria assegnata (configurabile), che gli permette di memorizzare i dataset in memoria se richiesto dall'applicazione utente (tramite l'istruzione *cache* su un RDD). Gli esecutori di diverse applicazioni Spark non possono comunicare tra di loro, facendo sì che diverse applicazioni non possono condividere i dati tra esse se non scrivendoli prima su disco. Gli esecutori vivono per tutta la durata di un applicazione; se un esecutore fallisce Spark riesce comunque a continuare l'esecuzione del programma ricalcolando solamente i dati persi. É bene che il driver e i nodi esecutori siano nella stessa rete o comunque vicini, dato che il driver comunica continuamente con essi.

2.3.3 I cluster manager

I cluster manager si occupano di gestire le risorse all'interno di un cluster. Ad esempio quando più applicazioni richiedono risorse del cluster, il cluster manager effettua lo scheduling nei nodi in base alla memoria e ai core della CPU liberi. Alcuni cluster manager permettono anche di dare priorità diverse a diverse applicazioni. Spark supporta i seguenti cluster manager:

- YARN: il nuovo gestore di risorse di Hadoop
- Mesos
- Standalone cluster manager

In più Spark fornisce uno script per essere eseguito in un cluster EC2 di Amazon.

2.4 Interazione con Hadoop YARN

YARN (Yet Another Resourse manager) è il nuovo gestore di risorse dei cluster Hadoop, introdotto nella versione 2 di Hadoop con l'obbiettivo di migliorare le performance e la flessibilità. Spark è la seconda applicazione più usata dopo MapReduce su YARN. Ci sono delle differenze nell'esecuzione tra MapReduce e Spark: in MapReduce ogni applicazione svolge due compiti, map e reduce, per poi liberare le risorse nel cluster. In Spark sono supportate le operazioni di map e di reduce, insieme a molte altre (per migliorare la flessibilità e la semplicità di programmazione), combinabili nell'ordine più pertinente per l'applicazione. Esse verranno illustrate nel capitolo successivo. La differenza sostanziale tra i due sistemi sta però nel fatto che in Spark le risorse vengono liberate solo alla fine di un applicazione, facendo sì che tra un job e l'altro non vi sia bisogno di scrivere e rileggere i dati da disco fisso. Gli executor rimangono in vita per tutta la durate di un'applicazione Spark, facendo sì che le risorse del cluster debbano essere condivise in maniera intelligente, dato che il numero di executor è fisso così come le risorse che essi utilizzano. É possibile configurare il numero di nodi executor, il numero di core e memoria che ognuno di essi utilizza, così come molte altre variabili (si legga la documentazione in materia [6]).

2.4.1 Perché eseguire Spark su YARN

Ci sono dei vantaggi offerti da YARN rispetto agli altri cluster manager:

- YARN permette di condividere le risorse con altre applicazioni che usano lo stesso cluster manager, tra cui job MapReduce o altri job Spark.
- Si possono dare priorità diverse a diverse applicazioni.
- Si può scegliere il numero di executor, mentre lo Standalone cluster manager obbliga ad usare un executor per nodo del cluster.
- YARN è solitamente già installato in un cluster insieme ad HDFS.

2.4.2 Esecuzione di Spark su YARN

Quando si esegue Spark su YARN, ogni executor viene eseguito in un container di YARN, che eseguirà i comandi dettati dal driver. Il driver invece può essere eseguito sia nel nodo che lancia il job Spark che in un nodo del cluster scelto dal Resource Manager, ad esempio YARN. La figura 2.3 illustra questo caso.

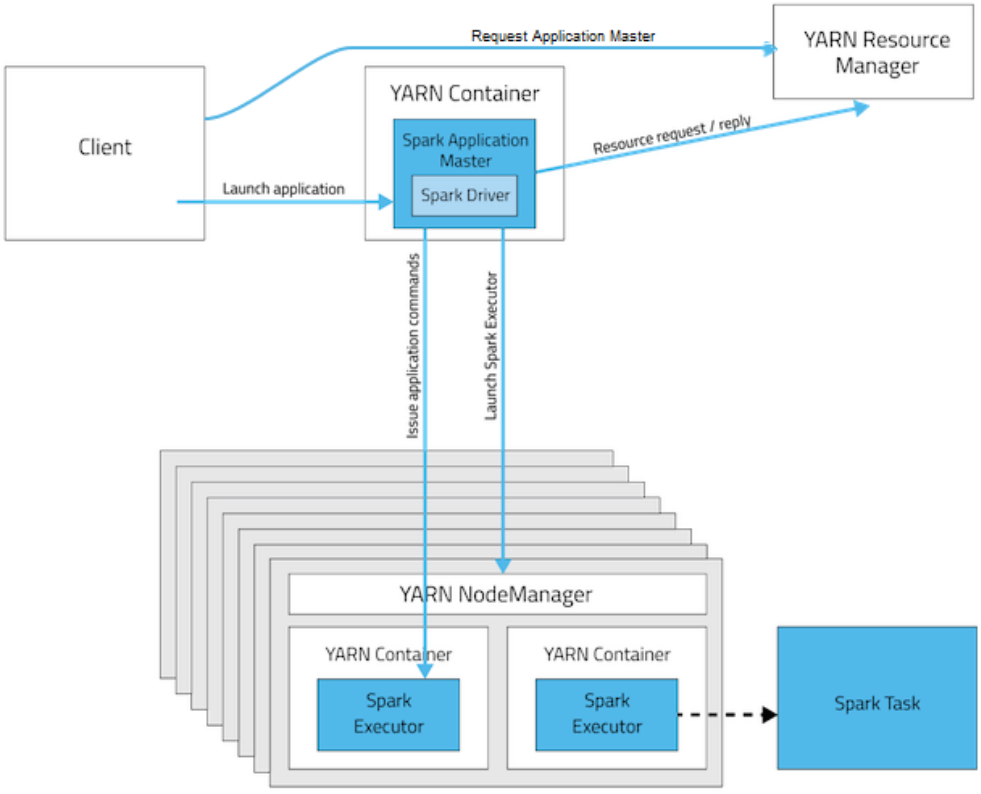

Figura 2.3: Spark in cluster mode

Per capire la differenza tra i due modi di esecuzione bisogna introdurre il concetto di ApplicationMaster. In YARN, ogni applicazione ha un ApplicationMaster, che è il primo container ad essere inizializzato per un'applicazione all'interno del cluster. L'ApplicationMaster è il processo che richiede risorse al cluster manager e chiede ai NodeManager (di cui ce n'è uno per nodo) di inizializzare i container per suo conto.

In modalità *yarn-cluster* il driver sarà eseguito all'interno dell'ApplicationMaster. Ciò fa sì che non c'è bisogno che chi manda in esecuzione un'applicazione Spark rimanga connesso al cluster, dato che tale nodo gestirà le risorse e dirà ad ogni executor quali task eseguire.

Questa modalità non è però adatta per eseguire analisi interattive di dati. I programmi interattivi devono essere eseguiti in modalità *yarn-client* (Figura 2.4), in cui l'ApplicationMaster ha il compito di richiedere le risorse al cluster mentre sarà il driver in esecuzione nel PC client che deciderà i compiti da far svolgere ai nodi del cluster.

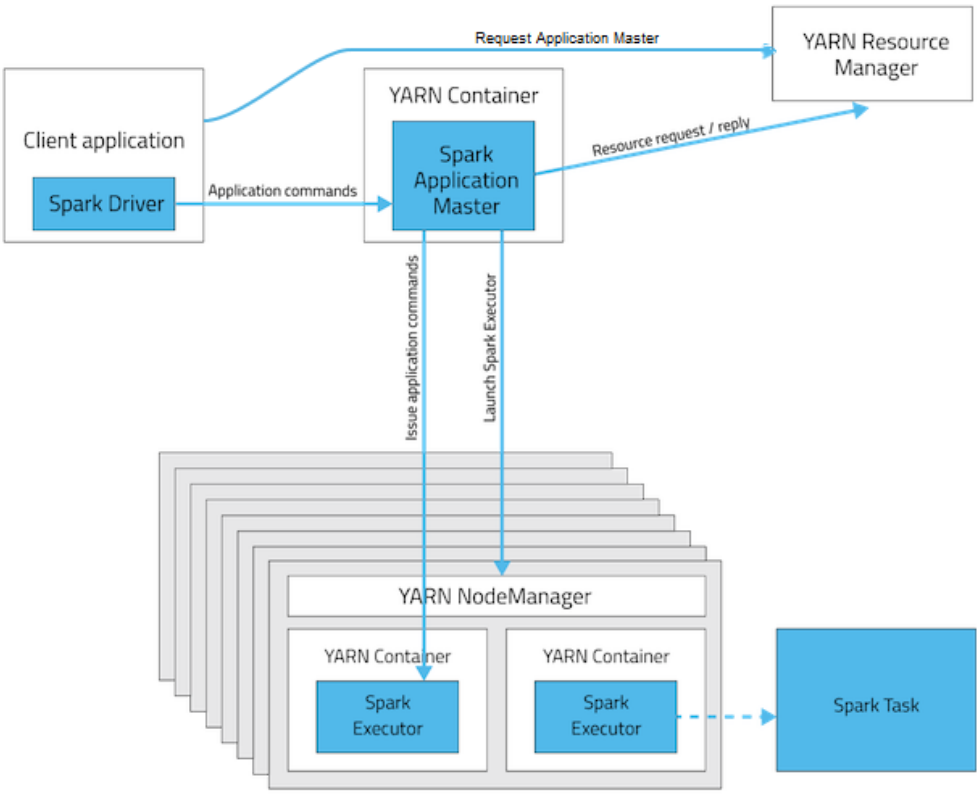

Figura 2.4: Spark in client mode

Viene ora illustrato come un applicazione Spark interagisce con YARN, passaggio per passaggio:

- Il client Spark, con il comando spark-submit manda in esecuzione un'applicazione sul cluster. La prima cosa che viene fatta è la connessione al ResourceManager, che crea il container per l'ApplicationMaster
- L'ApplicationMaster negozia con il ResourceManager le risorse per i container sulla base della configurazione di Spark (ad esempio quantità RAM/core per container, numero massimo container)
- Viene eseguita l'applicazione. L'ApplicationMaster decide quali task devono essere eseguiti dai singoli nodi del cluster, mentre monitora il loro stato di salute per rischedulare i compiti se necessario
- Alla conclusione dell'applicazione, l'ApplicationMaster libera le risorse comunicando con il ResourceManager

Il comando *spark-submit* ha in generale la seguente forma:

```
spark-submit
--class <main-class>
--master <master-url>
--deploy-mode <deploy-mode>
--conf <key>=<value>
other options
<application-jar>
[application-arguments]
```

In base al cluster manager scelto vi sono parametri opzionali. Per utilizzare YARN bisogna impostare --master a yarn , e --deploy-mode a client o cluster); alternativamente è possibile impostare --master a yarn-client o yarn-cluster ottenendo lo stesso risultato. É possibile specificare una lunga serie di parametri di configurazione, tra cui i più importanti sono:

- **--num-executors:** specifica il numero di esecutori da richiedere per l'applicazione Spark
- **--executor-memory:** specifica la quantità di memoria da richiedere per ogni executor
- **--executor-cores:** specifica la quantità di core utilizzati da ogni executor
- **--driver-memory:** specifica la quantità di memoria utilizzata dal driver

Questi parametri possono essere modificati dal codice dell'applicazione attraverso un oggetto di tipo *SparkConf*, che viene dato come parametro durante la creazione dell'oggetto *SparkContext*, il quale rappresenta la connessione con Spark [5, 6].

Capitolo 3

Il sottosistema di programmazione

3.1 Introduzione

Questo capitolo introduce la principale astrazione di Spark per lavorare sui dati, chiamata RDD (Resilient Distributed Dataset), che è semplicemente una collezione distribuita di elementi. Tutti i lavori svolti con Spark possono essere considerati come un caricamento di dati in un RDD, una trasformazione di RDD o azioni fatte su RDD per prendere dei risultati.

Una semplice API permette di operare sui dataset, in modo tale che i dati siano automaticamente distribuiti sul cluster e le operazioni parallelizzate. L'API è disponibile in Java, Scala e Python. Ci focalizzeremo sulla versione Java, anche se tra le varie versioni non ci sono differenze sostanziali.

L'API di programmazione fa parte del core di Spark, il componente su cui si appoggiano tutti gli altri moduli (Figura 2.1).

3.2 Un'idea generale sugli RDD

Un RDD è una collezione fault-tolerant e non modificabile di elementi sulla quale si può operare in parallelo. Essa è memorizzata in maniera distribuita, spezzata in parti chiamate **partizioni**, ognuna delle quali sta in un esecutore di Spark. Per processare i dati, il sistema creerà un task per ogni partizione. Gli RDD possono contenere elementi facenti parte di una qualsiasi classe Java, anche definita da un utente. Per capire il meccanismo di funzionamento di un RDD, si consideri il seguente esempio:

```
JavaRDD<String> linee = sc.textFile("data.txt");
JavaRDD<Integer> lunghezzaLinee = linee.map(s -> s.length());
```

```
int lunghezzaTotale = lunghezzaLinee.reduce((a, b) -> a + b);
```

La prima riga definisce un RDD contenente stringhe a partire da un file esterno. Tramite il metodo *textFile*, ogni riga del file di partenza diventa un elemento di un RDD contenente stringhe. Il file non viene toccato a questo punto, in quanto non sono stati richiesti risultati. La variabile *linee* è quindi una sorta di puntatore al file. La seconda linea trasforma l'RDD di partenza creandone un altro: per ogni riga del file di partenza, si ottiene un intero contenente la lunghezza della linea. L'ultima riga calcola attraverso il metodo *reduce* la lunghezza totale del file. Solo a questo punto vengono svolti calcoli paralleli, in quanto il metodo *reduce* è un'**azione**.

Per svolgere la computazione, il file data.txt viene caricato parallelamente dai nodi del cluster, e ogni macchina esegue una parte del metodo *map* (che è una **trasformazione**). A questo punto ogni macchina fa una reduce locale e manda il risultato al programma driver di Spark, che si preoccuperà di raccogliere i risultati parziali e di sommarli per ottenere la lunghezza totale del file.

Il motivo per il quale i dati vengono letti solo alla richiesta di un'azione è che non sempre l'intero file è necessario per il risultato. Supponendo di voler ottenere in output le righe con più di 100 caratteri, solo queste saranno tenute in memoria durante la lettura del file.

Dopo il calcolo, il dataset non rimane in memoria nei nodi del cluster. Se ci fosse servito per una futura operazione, avremmo dovuto chiamare il metodo *persist* sull'oggetto *lunghezzaLinee* prima di chiamare il metodo reduce. In alternativa il dataset verrebbe ricalcolato ad ogni azione il cui risultato dipende da esso. Se il dataset è molto grande e di difficile ricomputazione è anche possibile memorizzarlo su disco. Maggiori informazioni saranno date in seguito.

Gli oggetti di tipo RDD potrebbero quindi non contenere dati. Internamente, un RDD contiene le seguenti proprietà:

- una lista di **partizioni**, che potrebbe non contenere elementi o contenere solo parte del contenuto logico dell'RDD. In tal caso Spark cercherà di calcolare solo il necessario per ottenere il contenuto dell'RDD alla chiamata di un'azione.
- una lista di RDD dal quale l'RDD corrente deriva
- una funzione che determina come è calcolato l'RDD
- opzionalmente, la funzione di ripartizione applicata all'RDD
- opzionamente, una lista di nodi preferiti nel calcolo di ciascuna parte dell'RDD (ad esempio dove sono allocati i blocchi HDFS)

Per maggiori dettagli sulla struttura interna di un RDD si consiglia di leggere [7].

Nell'API Java di Spark gli RDD estendono tutti l'interfaccia *JavaRDDLike*. La classe JavaRDD<T>, che tramite i generics permette di ottenere dataset contenenti dati diversificati, è la più basilare e consente di lavorare con generici insiemi di valori. Spark fornisce delle varianti della classe JavaRDD, le cui più comuni sono:

- **JavaDoubleRDD:** può contenere solo elementi di tipo Double, offrendo la possibilità di effettuare con semplicità operazioni statistiche come media, mediana e varianza
- **JavaPairRDD<K,V>:** permette di lavorare con coppie chiave/valore, ad esempio per effettuare aggregazioni sulla base della chiave o operazioni di join tra dataset differenti.
- **JavaSchemaRDD:** un RDD simile a tabelle di un database, che contiene dati strutturati in colonne, ognuna delle quali ha un nome. Questo tipo di RDD viene sfruttato dal modulo Spark SQL, approfondito nel capitolo successivo.

3.3 Operazioni sugli RDD

Come accennato in precedenza, le operazioni sugli RDD possono essere suddivise in due categorie:

- **Trasformazioni:** a partire da un RDD se ne crea uno nuovo che dipende in qualche modo da quello di partenza. Ad esempio ogni elemento dell'RDD originario può risultare in un elemento trasformato nell'RDD di destinazione (metodo *map*) o possono essere selezionati degli elementi in base a una condizione booleana (metodo *filter*).
- **Azioni:** le azioni sono quelle operazioni che devono restituire un risultato al programma driver o che scrivono dell'output su disco fisso. Esempi sono il metodo *count* che restituisce il numero di elementi presenti in un RDD o il metodo *collect* che li restituisce.

L'unione di trasformazioni e azioni semplifica di molto la programmazione rispetto al modello MapReduce. Verrà di seguito fornita una breve introduzione all'API Java di Spark. L'intento di questa introduzione è quello di capire le potenzialità di questo sistema, mentre se si vuole implementare un programma Spark è necessario visitare la documentazione ufficiale[8] per controllare il

tipo di ogni parametro preso dai metodi offerti dall'API, che per semplicità ometteremo.

3.3.1 Trasformazioni sugli RDD

Le trasformazioni sono operazioni che, dato un RDD, ne restituiscono uno nuovo dipendente dall'RDD originario. Un oggetto RDD può essere visto come un insieme di informazioni che verranno utilizzate per il suo calcolo quando si effettuerà un'azione. Spark infatti non effettua calcoli finchè non è strettamente necessario. Vediamo la lista di traformazioni più comunemente utilizzate presenti in tutti gli RDD Spark:

- **map:** ogni elemento dell'RDD di partenza viene mappato in un elemento dell'RDD di destinazione
- **flatMap:** simile a map, ma ogni elemento viene mappato in 0 o più elementi sull'RDD di destinazione
- **filter:** vengono mantenuti solo quegli elementi per cui è valida una certa condizione
- **distinct:** vengono rimossi gli elementi duplicati dell'RDD
- **sample:** è possibile farsi restituire un sottoinsieme di una determinata grandezza dell'RDD di partenza, dove gli elementi vengono scelti casualmente con o senza ripetizione.
- **union:** vengono uniti insieme due RDD
- **intersection:** restituisce gli elementi presenti in entrambi gli RDD
- **subtract:** toglie gli elementi presenti in un RDD da un altro
- **cartesian:** effettua il prodotto cartesiano tra due RDD

Se si utilizzano RDD di tipo chiave/valore (JavaPairRDD) sono anche presenti queste utili trasformazioni:

- **reduceByKey:** gli elementi con la stessa chiave vengono ridotti in un solo elemento dove il valore dipende dai valori presenti nell'RDD iniziale
- **groupByKey:** raggruppa i valori con la stessa chiave in una lista
- **mapValues:** cambia i valori lasciando inalterata la chiave

- **flatMapValues:** per ogni coppia chiave/valore crea 0 o più coppie chiave/valore, dove la chiave è uguale a quella di partenza
- **keys:** restituisce un RDD contenente solo le chiavi dell'RDD di partenza
- **values:** restituisce un RDD contenente solo i valori dell'RDD di partenza
- **subtractByKey:** rimuove gli elementi la cui chiave è presente in un altro RDD
- **join:** effettua il join tra due RDD. I valori verranno uniti in un oggetto di tipo *Tuple2*. É possibile fare anche join esterni

3.3.2 Azioni sugli RDD

Mentre le traformazioni dichiarano gli RDD, le azioni producono valori che tornano al programma driver di Spark o che vengono memorizzati su file. A causa della *lazy evaluation*, i calcoli vengono effettuati nel cluster solo quando si effettua un'azione. Se si vuole quindi anticipare il calcolo, ad esempio per caricare un dataset in memoria, un trucco è quello di chiamare il metodo *count* sull'RDD per il quale si è prima chiamato il metodo *persist*.

Le azioni più comuni che si utilizzano sono probabilmente *reduce*, *collect* e *take*. Il metodo *reduce* permette di aggregare i dati, ad esempio per prenderne il massimo elemento o la sommatoria. Se prima si effettua un conteggio degli elementi, si riesce ad ottenere anche la media dei valori. Il metodo *collect* restituisce al programma driver tutti gli elementi di un RDD. Bisogna fare attenzione al fatto che tutti gli elementi devono entrare nella memoria centrale di un singolo nodo (il driver), pena fallimento del programma. Per ovviare a ciò sono presenti i metodi *take(n)* che restituisce n elementi dell'RDD e *takeSample(rimpiazzo, n, seed)* che restituisce n elementi presi a caso con o senza rimpiazzo. Una lista delle azioni presenti in tutti gli RDD è:

- **reduce:** aggrega gli elementi di un dataset
- **collect:** restituisce tutti gli elementi al programma driver sotto forma di un array
- **count:** conta il numero di elementi in un RDD
- **first:** restituisce il primo elemento di un RDD
- **take:** si restituiscono n elementi al driver
- **takeSample:** si restituiscono n elementi al driver, presi a caso tra quelli presenti

- **saveAsTextFile:** si salva l'RDD su disco usando la rappresentazione in stringa degli elementi

Quando si lavora con RDD di tipo chiave/valore sono presenti anche le azioni:

- **countByKey**: conta il numero di elementi per ogni chiave e restituisce una *Map* al programma driver
- **collectAsMap:** restituisce il dataset come *Map* invece che come array di tuple
- **lookup:** restituisce in una lista tutti i valori associati ad una data chiave

3.4 Caricamento degli RDD

Ci sono due modi per creare RDD: parallelizzare una collezione di dati presente nel programma driver o caricare un dataset da uno storage esterno, ad esempio da un file contenuto in HDFS, da HBase o da qualsiasi sorgente di dati supportata dalla classe *InputFormat* di Hadoop, tra cui Amazon S3 e il DBMS Cassandra.

3.4.1 Parallelizzare collezioni

É possibile parallelizzare collezioni chiamando il metodo *parallelize* della classe *JavaSparkContext*. Gli elementi della collezione saranno copiati in un dataset distribuito nel quale sarà possibile svolgere del calcolo parallelo. Per esempio, per parallelizzare un vettore contenente i numero da 1 a 5:

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

Una volta creato si possono svolgere operazioni parallelizzate sugli elementi, come il calcolo della somma vista in precedenza. Il metodo *parallelize* prende un secondo parametro opzionale, il numero di partizioni. Ogni RDD è spezzato nel cluster in un insieme di partizioni, ognuna delle quali contiene parte dei dati. Solitamente si vogliono da 2 a 4 partizioni per ogni core del cluster, in modo da massimizzare il parallelismo senza far pesare troppo l'overhead dovuto alla gestione di troppe partizioni. Spark tenta comunque di scegliere il numero giusto di partizioni in base alla grandezza del cluster.

3.4.2 Formati di file

Spark supporta una numerosa quantità di formato di file, tra cui file di testo, file JSON, file CSV, SequenceFiles, Protocol buffers e Object files.

File di testo

I file di testo sono un formato semplice e popolare quando si lavora con Hadoop. Quando si carica un file di testo, tramite il metodo *textFile*, ogni riga del file diventa un elemento dell'RDD.

```
JavaRDD<String> input =
  sc.textFile("file:///home/user/directory/README.txt")
```

Se si usa un path del filesystem locale, il file deve essere presente in ogni nodo del cluster nella directory specificata. Anche il metodo *textFile*, così come *parallelize* e molti altri metodi dell'API, prende un parametro opzionale che indica quante partizioni creare per il file. Di default, viene creata una partizione per ogni blocco del file (128 MB di default su HDFS) e non si possono però avere meno partizioni che blocchi.

Per leggere file spezzati in parti, come quelli generati dai reducer di un job MapReduce, si può utilizzare il metodo *textFile* passandogli come parametro il percorso della directory contenente le parti del file. L'RDD restituito è del tutto analogo a quello che si otterrebbe leggendo un file unico. Un'alternativa da usare se si vuole sapere da quale file deriva ogni riga di testo è usare il metodo *wholeTextFiles* che, per ogni riga di testo, restituisce una coppia chiave/valore dove la chiave è il nome del file di input.

Per salvare file di testo gli RDD offrono il metodo *saveAsTextFile*. Il percorso specificato viene trattato come una cartella, contenente più file in modo che più nodi possano scrivere il risultato della computazione contemporaneamente (si ricorda che in HDFS solo un client alla volta può scrivere su file).

File JSON

JSON è un popolare formato per la memorizzazione di dati semistrutturati. Il modo più semplice per leggere un file JSON è l'utilizzo del metodo *jsonFile* del sottosistema Spark SQL. Maggiori informazioni verranno date nel capitolo successivo. Un'alternativa è quella di caricare il file JSON come se fosse un file di testo e usare un parser per trasformarlo in un RDD di oggetti di un certo tipo, dipendente dal contenuto del file.

File CSV

I file CSV sono file di testo in cui ogni riga del file contiene un numero fisso di elementi separati da una virgola. Varianti del formato prevedono una separazione con tabulazione o con punto e virgola. Se i campi del file non hanno caratteri di andata a capo la situazione è molto semplice: si legge il file CSV come se fosse un file di testo e si parsa ogni riga del file. É di seguito presentato un esempio:

```
import au.com.bytecode.opencsv.CSVReader;

public static class ParseLine implements Function<String
    , String[]> {
    public String[] call(String linea) throws Exception {
        CSVReader reader = new CSVReader(new StringReader(
            linea));
        return reader.readNext();
    }
}
JavaRDD<String> csvFile = sc.textFile(inputFile);
JavaPairRDD<String[]> csvData = csvFile.map(new
    ParseLine());
```

Se ci sono caratteri di andata a capo nei campi del file, bisogna leggere il file per intero prima di parsarlo.

```
public static class ParseLine implements FlatMapFunction
    <Tuple2<String, String>, String[]> {
    public Iterable<String[]> call(Tuple2<String, String>
        file) throws Exception {
        CSVReader reader = new CSVReader(new StringReader(
            file._2()));
        return reader.readAll();
    }
}
JavaPairRDD<String, String> csvData = sc.wholeTextFiles(
    inputFile);
JavaRDD<String[]> keyedRDD = csvData.flatMap(new
    ParseLine());
```

Sequence File

I Sequence File sono file binari che contengono coppie chiave/valore. Questo formato è molto usato in MapReduce, quindi è molto probabile che si hanno dei file da leggere in questo formato. I Sequence File contengono elementi che estendono l'interfaccia *Writable* di Hadoop. Sono presenti le classi

- IntWritable per memorizzare numeri interi di tipo Integer. Si può usare la classe VIntWritable per memorizzare numeri di grandezza variabile, utile se si memorizza una grande quantità di numeri piccoli
- LongWritable per numeri di tipo Long. É anche presente la classe VLongWritable
- FloatWritable per Float
- DoubleWritable per Double
- BooleanWritable per Boolean
- BytesWritable per Byte[]
- Text per String
- ArrayWritable<TW> per TW[], dove TW è di tipo Writable
- MapWritable<AW, BW> per Map<AW, BW>

Per leggere i Sequence File si può usare il metodo *sequenceFile(path, keyClass, valueClass, minPartitions)* nella quale keyClass e valueClass devono essere di tipo Writable. Un esempio che legge un file con chiavi di tipo Text e valori di tipo IntWritable è

```
public static class ConvertToNativeTypes
      implements PairFunction<Tuple2<Text, IntWritable
         >, String, Integer> {
   public Tuple2<String, Integer> call(Tuple2<Text,
      IntWritable> record) {
   return new Tuple2(record._1.toString(), record._2.get
      ());
   }
}
JavaPairRDD<Text, IntWritable> input = sc.sequenceFile(
   fileName, Text.class, IntWritable.class);
JavaPairRDD<String, Integer> result = input.mapToPair(
   new ConvertToNativeTypes());
```

3.4.3 Filesystem supportati

Filesystem locale

Quando si legge un file da un filesystem locale, il file deve essere disponibile in tutti i nodi del cluster nello stesso percorso. É possibile usare filesystem remoti come NFS e AFS montandoli su sutti i nodi nello stesso percorso, e Spark non avrà problemi a funzionare. Se i file non sono già su tutti i nodi del cluster li si possono parallelizzare tramite il metodo *parallelize*, ma questa è un operazione sconsigliata in quento richiede un notevole uso della rete.

Amazon S3

Amazon S3 è un servizio di storage offerto dal servizio cloud di Amazon. Solitamente si utilizza un cluster Amazon EC2 per accedere a dati memorizzati su S3, in modo da utilizzare la veloce rete interna di Amazon e non trasferire dati attraverso Internet, perdendo ordini di grandezza in termini di prestazioni o costo.

Per accedere da Spark a dati memorizzati su cluster S3 bisogna impostare correttamente le variabili di sistema AWS_SECRET_ACCESS_KEY e AWS_ACCESS_KEY_ID sulla base delle credenziali create dalla console di Amazon Web Services. Dopo di chè si possono leggere file usando il prefisso *s3n://*, ad esempio *sc.textFile("s3n://bucket/LOGFILE.log")*. Per leggere file da Amazon S3 bisogna avere sia i permessi di *read* che di *list*, in quanto Spark effettua il listing sulle directory per capire quali file debba leggere.

HDFS

Il filesystem distribuito di Hadoop è una scelta popolare quando si utilizza Spark. Solitamente i nodi contenenti dati sono gli stessi su cui è installato Spark, ed il sistema sfrutterà questo fatto per far elaborare i dati dagli stessi nodi su cui sono memorizzati evitando di utilizzare la rete. I percorsi HDFS vanno specificati in questo modo: *hdfs://master:port/path*, dove master è il NameNode del filesystem.

3.5 Ottimizzazione delle prestazioni

Verranno illustrati i parametri ed il modo di utilizzo di Spark che permettono di migliorare le prestazioni senza compromettere le funzionalità del sistema. Capire il modo in cui Spark gestisce i dati è una buona cosa se si vuol ottenere il massimo dal sistema.

3.5.1 Come funziona il sistema

Quando si crea un RDD da uno precedente usando una trasformazione, il nuovo RDD mantiene un puntatore all'RDD o agli RDD padri, insieme a metadati che indicano il tipo di relazione che si ha con loro. Il puntatore ai padri permette ricorsivamente di arrivare ad un RDD di partenza. Il caso in cui si ha più di un genitore è dato dalle operazioni *union*, *join*, ecc..

Lo scheduler di Spark crea un piano fisico di esecuzione ogni qualvolta si richiede il risultato di un'azione. Iniziando dall'RDD finale da calcolare, si visita il grafo delle dipendenze all'indietro per capire cosa bisogna calcolare. Si parte da file di ingresso, dati parallelizzati o RDD persistenti.

Le trasformazioni da fare per il calcolo dell'RDD finale sono raggruppate in stage. Ogni stage contiene trasformazioni che possono essere svolte senza muovere dati in rete. Il numero di shuffle in rete è una buona indicazione di quanto sarà lento il programma, ovviamente prendendo in considerazione la quantità di dati da movimentare.

Per vedere il piano di esecuzione per il calcolo di un RDD si può chiamare il metodo *toDebugString* sullo stesso. Il numero di indentanzioni indicherà il numero di stage. Un possibile risultato è il seguente (Figura 3.1):

```
(2) ShuffledRDD[140] at reduceByKey at <console>:17
 +-(2) MappedRDD[139] at map at <console>:17
 |  MappedRDD[138] at map at <console>:15
 |  data.txt MappedRDD[137] at textFile at <console>:13
 |  data.txt HadoopRDD[136] at textFile at <console>:13
```

Da questo piano di esecuzione si evince che viene letto il file data.txt e viene trasformata ogni riga tramire due map. Dopo di chè li si trasferiscono in rete per effettuare una reduceByKey, che per ogni chiave di un dataset di tipo key/value calcola un dato aggregato. Di default viene utilizzata una funzione hash per decidere in quale nodo debba andare ogni coppia chiave/valore, ma è possibile specificare una funzione di ripartizione *custom* che sfrutta le proprietà dei dati in modo da rendere uniforme la distribuzione.

Dato che i dataset sono suddivisi in **partizioni**, bisogna verificare che si usi il numero giusto di esse. Le scelte di default di Spark, cioè quella di avere una partizione per ogni blocco di file, e di avere il parallelismo dei dataset spostati in rete (shuffled) basata su quella del dataset genitore (quello da cui deriva), sono convenienti nella maggior parte dei casi. Se si avessero troppe poche partizioni, ad esempio meno del numero di core del cluster, ovviamente non si riuscirebbe a sfruttare tutta la potenza di calcolo. Se invece se ne avessero troppe, ci sarebbe un overhead dovuto alla gestione delle partizioni.

In entrambi i casi si può usare il metodo *repartition* che, prendendo come

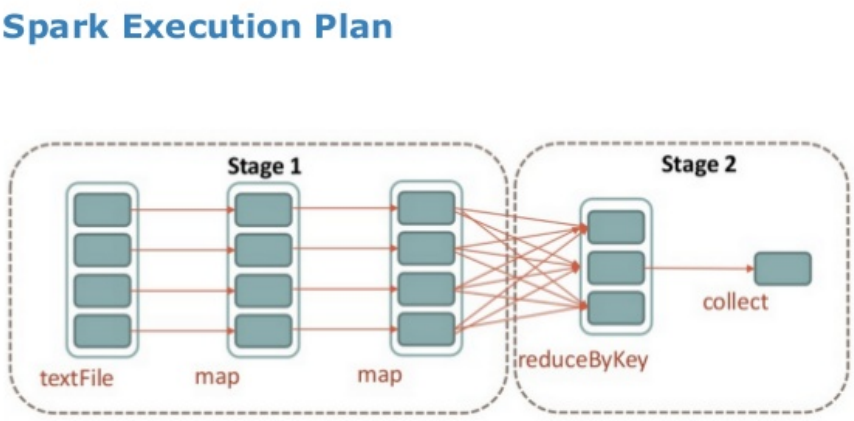

Figura 3.1: Spark execution plan

parametro il numero di partizioni che si vogliono, trasferisce i dati in rete in modo da distribuirli nel cluster. Se si riduce il numero di partizioni è consigliato usare il metodo *coalesce*, che permette di evitare l'utilizzo della rete: le partizioni verranno fuse localmente nei singoli nodi.

3.5.2 Persistenza dei dati

Di default un RDD viene ricalcolato ogni volta che serve per un'azione. Se lo si vuole utilizzare per più di un'azione, bisogna chiamare il metodo *persist* prima di invocare l'azione. Ciò farà sì che le partizioni dell'RDD risultante vengano memorizzate in un supporto. É possibile specificare dove memorizzare tali partizioni attraverso gli StorageLevel, di cui i più importanti sono:

- MEMORY_ONLY: vengono memorizzati i dati così come sono in memoria
- MEMORY_ONLY_SER: vengono memorizzati i dati in memoria, però serializzati per occupare meno spazio. Ciò significa però utilizzare più risorse della CPU
- MEMORY_AND_DISK: vengono memorizzati i dati in memoria e nel disco fisso. Viene utilizzato il disco fisso solo se la memoria non è sufficiente.

- MEMORY_AND_DISK_SER: vengono memorizzati i dati in memoria e nel disco fisso, serializzati.

- DISK_ONLY: i dati vengono memorizzati solamente nel disco fisso.

Il fatto che gli RDD non siano *cachati* di default potrebbe sembrare strano, ma non ha senso salvare dati che non si utilizzerebbero in futuro. Per questo bisogna specificare quali dati saranno utili.

Se si provano a memorizzare in memoria più dati di quanti lo spazio consente, Spark cancellerà dalla memoria automaticamente le partizioni usate meno di recente. Se lo storage level della partizione da cancellare è MEMORY_AND_DISK o MEMORY_AND_DISK_SER, i dati vengono spostati su disco, altrimenti verranno ricalcolati la prossima volta che serviranno. Per questo bisogna evitare di memorizzare dati inutilmente: si rischia di eliminare altri dati utili in futuro per concedergli spazio. Un semplice consiglio è quello di rendere persistente una partizione solo se ricalcolarla richiederebbe shuffle in rete, l'operazione più onerosa in un cluster. É possibile eliminare dati cachati attraverso il metodo *unpersist*, per dare spazio a nuovi dataset.

3.5.3 Lavorare con le partizioni

Lavorare partizione per partizione permette alle applicazioni di risparmiare il tempo dedicato a lavori altrimenti svolti elemento per elemento, come connessioni a database. Se ad esempio si vogliono filtrare coppie chiave/valore in base al fatto che la loro chiave è presente in un database, è preferibile aprire la connessione al database una sola volta per ogni partizione piuttosto che per ogni elemento del dataset. La classe *RDD* presenta i seguenti metodi per lavorare con le partizioni:

- **mapPartitions:** prende come parametro una funzione (più precisamente un oggetto che definisce un metodo) che prende in ingresso e restituisce un iteratore[9] di elementi. I valori in ingresso saranno i valori presenti nella partizione

- **mapPartitionsWithIndex:** come *mapPartitions*, ma viene dato in ingresso anche l'indice della partizione

- **foreachPartition:** prende come parametro una funzione, la quale prende in ingresso un iteratore agli elementi di una partizione. Non sono restituiti valori. Questo metodo è utile ad esempio per memorizzare dei dati in un database.

Un esempio di uso del metodo *mapPartitions* è quello di velocizzare le aggregazioni. Supponiamo di voler calcolare la media di valori. Un modo è quello di trasformare con una *map* ogni elemento in una coppia (valore, 1) in modo che sommando gli elementi tra loro tramite una *reduce* si abbia una coppia (somma, #elementi). Il rapporto tra i due valori sarà la media cercata. Un'alternativa a ciò è farsi restituire da *mapPartitions* una coppia (sommaParziale, #elementi) per ogni partizione. Si farà quindi una *reduce* come prima, ma si è evitato di trasformare ogni elemento in una coppia.

3.5.4 Configurazione di Spark

Di default, quando si esegue Spark su YARN, esso richiede solamente due container per gli executor. Ovviamente questa è una cosa insensata in un cluster di grandi dimensioni, dato che si sfrutterebbero solamente due nodi. Il parametro di configurazione *--num-executors*, che indica il numero massimo di executor, deve quindi essere cambiato per sfruttare tutte le risorse a disposizione. É anche possibile impostare il numero di core e di memoria utilizzati da ogni esecutore, tramite le configurazioni *--executor-cores* e *--executor-memory*. Essi non indicano la quantità massima utilizzabile, ma la precisa quantità da utilizzare. É per questo una buona idea impostare tali valori ad essere un sottomultiplo del massimo.

Una domanda frequente che ci si fa è se conviene mettere un solo esecutore per nodo del cluster, che sfrutta tutte le sue risorse, o se conviene metterne una quantità maggiore di più piccoli. É da notare che le risorse totali a disposizione utilizzabili da Spark non cambiano. Dato che non vi è comunicazione tra i container e di conseguenza ogni variabile broadcast deve essere trasferita ad ognuno di essi, è solitamente preferibile usare un piccolo numero di esecutori (e quindi container) grandi. Un'altro motivo è che i join fatti su RDD possono richiedere molta memoria, come spiegato nel capitolo successivo. Se i container sono troppo grandi si avranno però delle latenze consistenti dovute al garbage collector di Java. Per evitare questo tipo di problema è per questo consigliato non dare ad ogni executor oltre 64 GB di RAM.

Capitolo 4

Il sottosistema SQL

Verrà ora introdotto Spark SQL, il modulo di Spark per lavorare con dati strutturati e semistrutturati. I dati strutturati sono quei dati per i quali i record hanno un determinato insieme di campi, di tipo conosciuto. Nei dati semistrutturati, come quelli contenuti in un file JSON, i campi potrebbero variare tra un record e l'altro, in base alle necessità.

Spark SQL permette di leggere dati da fonti diverse, tra cui database Hive e file JSON, e di interrogarli usando il linguaggio di alto livello HiveQL, una variante dell'SQL. É possibile connettersi e sfruttare l'ottimizzato ambiente di Spark tramite qualunque strumento che supporta i connettori standard JDBC/ODBC, come Tableau, molto usato in Business Intelligence (Figura 4.1).

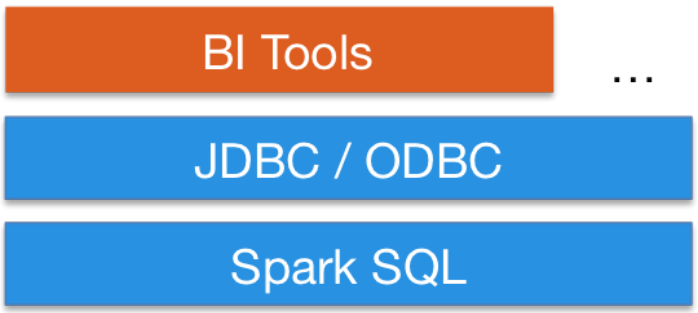

Figura 4.1: Connettori di Spark SQL

Spark SQL, come detto in precedenza, si integra perfettamente con il resto del sistema Spark, in quanto gli oggetti restituiti dalle interrogazioni sono degli RDD (degli SchemaRDD in particolare). É possibile convertire oggetti

di tipo *JavaRDD* in *SchemaRDD* e viceversa, rendendo possibile interrogare con HiveQL qualsiasi dataset si voglia. In pratica gli SchemaRDD sono degli RDD di oggetti Row, con associate informazioni sulle colonne.

Verrà di seguito illustrata l'interazione di Spark SQL con Hive (Figura 4.2), il DBMS di Hadoop simile ai tradizionali RDMS, che sfrutta MapReduce per l'esecuzione delle interrogazioni. Verranno nel capitolo successivo analizzati i risultati di benchmark che mettono i due sistemi a confronto.

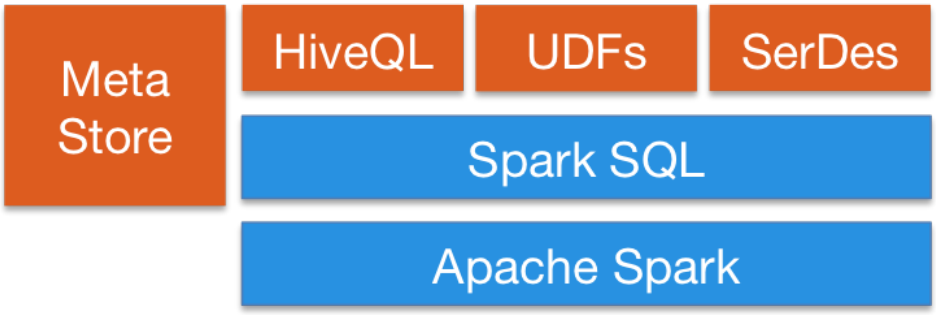

Figura 4.2: Architettura di Spark SQL

Per prima cosa, per utilizzare Spark SQL, bisogna creare un oggetto di tipo *JavaHiveContext* a partire da un *JavaSparkContext*:

```
JavaSparkContext sc = ...;
JavaHiveContext sqlContext = new JavaHiveContext(sc);
```

É anche possibile creare un oggetto di tipo *JavaSQLContext*, con il quale però non si possono accedere alle tabelle Hive (utile se non si ha Hive installato).

4.1 Metadata Repository

Spark permette di leggere le tabelle Hive, in qualsiasi formato esse siano memorizzate. Dato che Hive memorizza i metadati separatamente rispetto ai dati, Spark deve leggere il metastore (il metadata repository) per inferire quali sono i nomi e i tipi delle colonne.

Il metastore permette di avere astrazione sui dati. Senza astrazione, un utente dovrebbe fornire informazioni sui formati dei dati, su come leggerli e su come scriverli. In Hive, queste informazioni sono definite durante la creazione di una tabella e sono usate ogni qualvolta la tabella è referenziata. Ciò è molto simile a quello che accade in un sistema tradizionale.

I metadati memorizzati da Hive riguardano:

- **Database:** un database è un namespace per le tabelle.
- **Tabelle:** i metadati per una tabella contengono informazioni sulle colonne, sul proprietario, sulla memorizzazione e sulla seriallizzazione/deserializzazione. Le informazioni sulla memorizzazione includono dove è memorizzato il database e quale formato è stato utilizzato (es. Avro o Parquet). I metadati sulla serializzazione e deserializzazione includono l'implementazione di classi serializzanti e deserializzanti dei dati, con informazioni su come utilizzarle. Tutte queste informazioni sono fornite durante la creazione di una tabella.
- **Partizioni:** ogni partizione può avere metadati su colonne e su serializzazione/deserializzazione. Questo permette di facilitare cambi di schema senza modificare vecchie partizioni.

Il metastore è memorizzato in un database esterno tramite un ORM chiamato DataNucleus piuttosto che in HDFS. Il motivo di ciò è che l'HDFS non permette di modificare i file se non aggiungendo dati in fondo ad esso, cosa non accettabile per contenere metadati. Si hanno però problemi di sincronizzazione e di scalabilità per l'utilizzo di un database esterno, ma questo è attualmente l'unico modo per interrogare e aggiornare i metadati.

Il metastore può essere configurato in due modi: remoto o embedded. In modalità remota il metastore è un servizio Thrift, usabile da client non Java. In modalità embedded invece i client Hive si connettono direttamente al metastore usando JDBC.

4.2 Espressività SQL

HiveQL è un linguaggio molto simile ad SQL, che permette di creare tabelle, caricare dati e fare interrogazioni. Vengono supportate tutte le clausule SQL tra cui *SELECT*, *FROM*, *WHERE*, *GROUP_BY* e *ORDER_BY*. Vengono inoltre aggiunte le clausule *CLUSTER_BY* e *SORT_BY*, che sono specifiche di Hive. Per una corretta comprensione di queste due clausule si rimanda a [12].

Gli operatori supportati da HiveQL sono:

- Operatori relazionali (=, !=, ==, <>, <, >, >=, <=, ecc.)
- Operatori aritmetici (+, -, *, /, %, ecc.)
- Operatori logici (AND, OR, ecc.)
- Costruttori di tipi complessi (map, struct, ecc.)

- Funzioni matematiche (sign, ln, cos, ecc.)
- Funzioni su stringhe

Questi sono i tipi di dato supportati, che comprendono strutture complesse come array e dizionari:

- TINYINT
- SMALLINT
- INT
- BIGINT
- BOOLEAN
- FLOAT
- DOUBLE
- STRING
- BINARY
- TIMESTAMP
- DATE
- ARRAY
- MAP
- STRUCT

Le differenze rispetto all'SQL risultano essere poche. Le più sostanziali risultano essere le seguenti:

- non sono supportati tipi di dato autoincrementali, molto usati nei database relazionali per definire le chiavi primarie. É in via di sviluppo il loro supporto [13].
- non sono possibili sottoquery all'interno della clausula *WHERE* prima della versione 1.3.0 di HiveQL. Comunque sia, anche nella versione 1.3 sono presenti dei limiti [14], tra cui il fatto che le sottoquery sono possibili solo nel lato sinistro di un'espressione, *IN/NOT IN* sono usabili solo su singola colonna, le referenze alla query padre sono permesse solo nel *WHERE* e se si utilizza *EXISTS/NOT EXISTS* bisogna avere un predicato nella sottoquery che referenzia la query genitore.

Sono permessi i JOIN tra tabelle, compresi i join esterni (LEFT|RIGHT|FULL OUTER JOIN), il LEFT SEMI JOIN e il CROSS JOIN.

Il LEFT JOIN è un tipo di join nel quale sono restituite tutte le righe della tabella di sinistra corrisposte da righe della tabella di destra, più le righe della tabella di sinistra non corrisposte (gli elementi della tabella di destra saranno *null*). Il RIGHT JOIN è analogo, restituendo però tutte le tuple della tabella di destra.

Il FULL JOIN restituisce invece sia le righe della tabella di sinistra sia le righe della tabella di destra non corrisposte, unite al risultato di un normale inner-join.

Il LEFT SEMI JOIN è un operatore che restituisce una riga della colonna di sinistra solo se esiste una corrispondenza nella tabella di destra: a contrario di un normale join non saranno quindi presenti nell'output le colonne della tabella di destra.

Il CROSS JOIN invece è un semplice prodotto cartesiano, in cui gli elementi dell'output sono dati da tutti gli elementi della prima tabella accoppiati con tutti gli elementi della seconda tabella.

Ci sono alcune funzionalità di Hive non ancora supportate dal modulo Spark SQL. Alcune di esse sono:

- tabelle con buckets: sono tabelle partizionate in base ad una funzione hash, che permettono di effettuare join più velocemente
- il tipo di dati UNION
- collezionare statistiche sulle colonne

4.3 Le fonti dei dati

Sia la lettura che la scrittura dei dati richiede di utilizzare la classe *SchemaRDD*, i cui oggetti sono simili a tabelle di database relazionali. Ogni SchemaRDD contiene degli oggetti di tipo Row, array di lunghezza fissa che contengono i campi di un record. Dato un oggetto Row, il metodo get(position) permette di prendere l'elemento nella posizione voluta, che dovrà essere convertito nel tipo giusto, dato che viene restituito un generico Object. Sono presenti anche metodi più specifici, come getString, getInt, ecc. [16]. Uno SchemaRDD contiene pure informazioni sul nome e sul tipo di ogni colonna.

Spark SQL può leggere dati automaticamente da una moltetudine di fonti, tra cui le più usate sono file JSON e database Hive. Nel primo caso, lo schema viene capito automaticamente dal contenuto del file; mentre nel secondo viene letto dal metastore (hive-site.xml deve essere configurato correttamente).

Alternativamente è possibile convertire normali RDD in SchemaRDD, per poi interrogarli con HiveQL.

4.3.1 JSON

Si possono caricare file JSON e Spark ne capirà automaticamente la struttura permettendo di interrogarli tramite il linguaggio HiveQL. Dato che il progetto di tesi è maggiormente orientato sui dati strutturati, verrà fornito solo un esempio illustrativo sulle potenzialità del sistema.

Per interrogare un file JSON bisogna prima di tutto caricarlo tramite il metodo *jsonFile* di JavaHiveContext, per poi registrarlo come tabella temporanea. Supponendo di avere un file JSON di voti, questo esempio seleziona la matricola ed il voto per ogni record:

```
SchemaRDD voti = sqlCtx.jsonFile(jsonFile);
voti.registerTempTable("voti");
SchemaRDD results = hiveCtx
    .sql("SELECT studente.matricola, voto FROM voti");
```

É da notare come le strutture innestate siano anch'esse interrogabili.

4.3.2 Apache Hive

Possono esser letti tutti i formati di tabella supportati da Hive, tra cui i TextFiles, gli RCFiles, Parquet, Avro e ORC. I file Avro memorizzano i dati riga per riga, in formato binario, rendendoli più compatti rispetto ai file di testo. I file Parquet sono invece più elaborati: le tabelle sono memorizzate per colonna, ed ogni colonna è a sua volta memorizzata per pagine, che contengono parte dei valori presenti nella colonna (Figura 4.3). Le pagine sono memorizzate compresse, e tale compressione è molto più efficiente di quanto si otterrebbe memorizzando i dati riga per riga, in quanto gli elementi di una colonna tendono ad avere una maggiore similitudine. Solitamente si riesce a ridurre la grandezza di un database ad un terzo [19].

Per interrogare tabelle Hive si utilizza il metodo *sql* di JavaHiveContext, che restituisce uno SchemaRDD contenente il risultato della query.

4.4 Ottimizzazione e piani di esecuzione

Spark SQL può memorizzare tabelle in memoria in formato colonnare, usando il metodo *sqlContext.cacheTable(tableName)*. In questo modo Spark

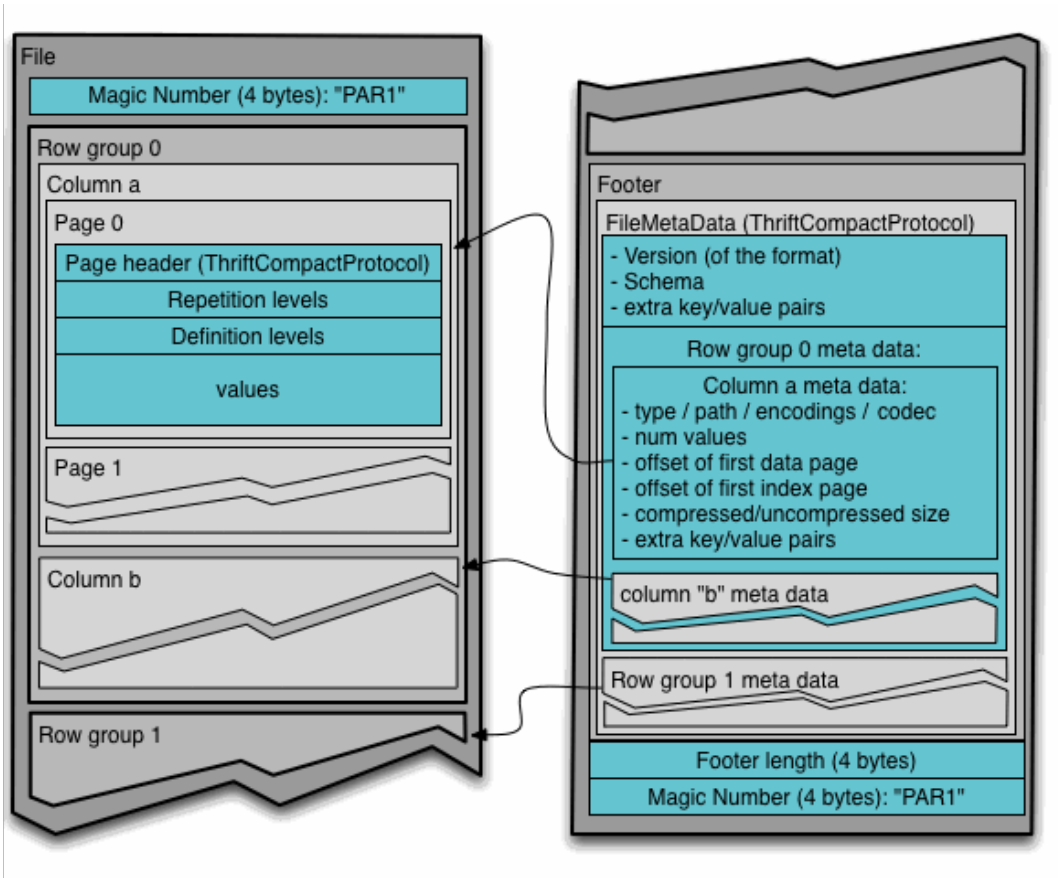

Figura 4.3: Memorizzazione di database Parquet

scansionerà successivamente solo le colonne necessarie. Se invece si utilizza il metodo *persist* dei generici RDD, le tabelle non saranno memorizzate in formato colonnare, non permettendo future ottimizzazioni, se non il fatto che la tabella non sarà ricalcolata alla successiva azione che dipende da essa. I dati in memoria vengono compressi automaticamente in base alla configurazione di Spark, ed è possibile rimuovere dati persistenti tramite una chiamata al metodo *sqlContext.uncacheTable(tableName)*.

Se si leggono dei dati e li si filtrano attraverso un predicato, il normale modo di eseguire l'interrogazione sarebbe di leggere l'intero dataset e di filtrare i record di interesse. Se però il modo in cui sono scritti i dati permette di recuperare i dati sulla base di un range, o di un'altra restrizione, Spark SQL riesce a caricare direttamente solo i dati di interesse, leggendo molti meno dati.

4.4.1 Modalità di esecuzione dei Join

In Spark SQL ci sono due modi principali di fare i join, chiamati BroadcastHashJoin e ShuffledHashJoin. Per capirne il meccanismo di funzionamento, supponiamo di avere due tabelle, chiamate A e B, e supponiamo che si voglia fare un equi-join sul campo *key* di A e sul campo *value* di B.

Se si utilizza uno ShuffledHashJoin, i record della tabella A vengono trasferiti in rete sulla base di una funzione hash applicata al campo *key*. La funzione viene cioè usata per decidere a quale executor inviare i dati. La stessa cosa viene fatta con la tabella B sul il campo *value*, usando la medesima funzione hash. A questo punto si ha la garanzia che per ogni chiave della prima tabella, le corrispettive tuple della tabella B saranno mantenute dallo stesso esecutore di Spark. Un join effettuato in locale da tutti gli executor, fatto con le tecniche usabili su tradizionali database (es. un hash join), ci darà il risultato del join. La figura 4.4 illustra questo tipo di join.

Per fare il join in locale, Spark crea in memoria, su ogni executor, un hash table per una delle due tabelle (la build table), mentre scorre l'altra tabella (la stream table) per fare il join. A questo punto ogni nodo del cluster contiene parte del risultato del join. Lo ShuffledHashJoin è l'unica soluzione se le tabelle A e B sono entrambe di grandi dimensione, tali da non entrare in memoria centrale.

Se una delle due tabelle fosse piccola, potrebbe essere conveniente usare un BroadcastHashJoin, nel quale la più piccola delle due tabelle viene replicata su ogni executor di Spark. Ogni executor farà quindi un hash join tra le partizioni della tabella grande contenute nel nodo e la tabella piccola (Figura 4.5), mantenendo un hash table della tabella piccola in memoria e scorrendo la tabella grande per fare il join.

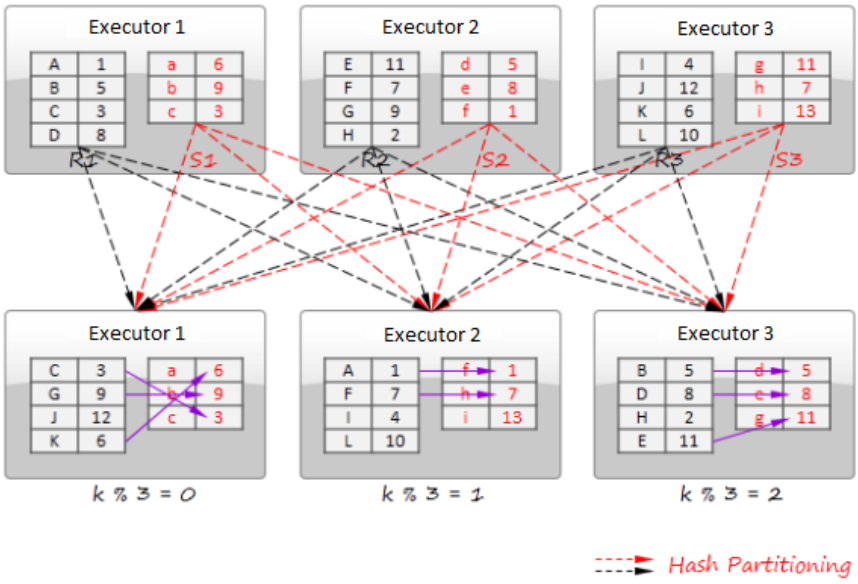

Figura 4.4: Lo ShuffledHashJoin di Spark SQL

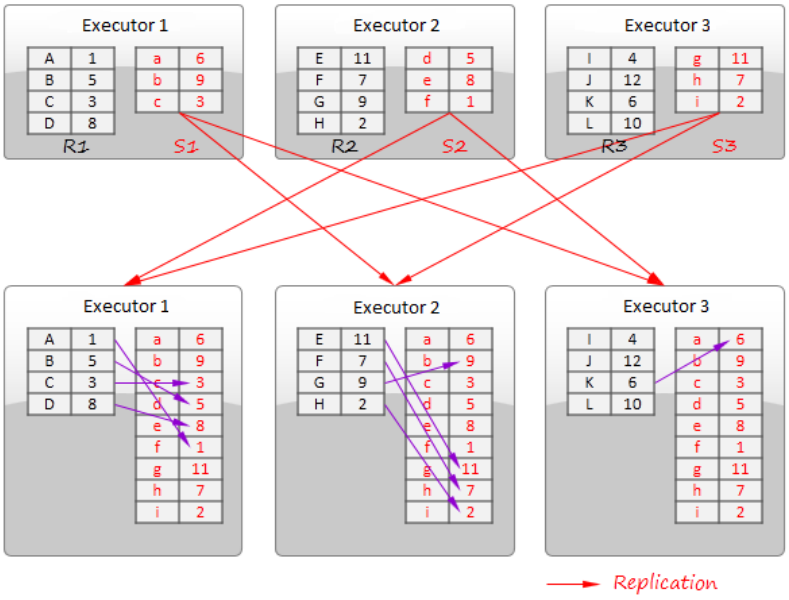

Figura 4.5: Il BroadcastHashJoin di Spark SQL

Il tipo di join da utilizzare viene deciso in automatico dal sistema sulla base di un impostazione (SQLConf.AUTO_BROADCASTJOIN_THRESHOLD): se una delle due tabelle su cui si vuole fare il join ha una grandezza stimata minore della soglia fissata, si effettua un BroadcastHashJoin dove la build table è la tabella trasmessa in rete; se entrambe le tabelle sono più grandi di questa soglia, si effettua uno ShuffledHashJoin e si decide quale delle due tabelle sarà la build table.

Un limite della tecnica attualmente utilizzata per fare i join è che se i dati sono molti, potrebbe non essere possibile creare un hash table in memoria. Attualmente non c'è una soluzione a ciò se non assegnare più memoria possibile ai container degli executor. Sono attualmente in corso di implementazione algoritmi specifici che risolvono tale problema [17, 18].

4.4.2 Modalità di esecuzione delle aggregazioni

Le aggregazioni vengono svolte in maniera intelligente. Supponiamo che si abbia una query con clausula GROUP BY, ad esempio "*SELECT SUM(importo) FROM pagamenti GROUP BY destinatario*". Spark, per fare l'aggregazione, ne farà prima una parziale sui singoli executor, in modo da avere su ogni esecutore il risultato dell'aggregazione dei dati da esso posseduto. Dopo di ché il sistema trasferisce tale risultato in rete usando una funzione hash applicata alla colonna di aggregazione, in modo tale che i valori potranno, per ogni valore di tale colonna, essere sommati ancora una volta per ottenere il risultato globale dell'aggregazione. Fare due aggregazioni differenti permette di utilizzare meno banda di rete. Se si trasferissero i dati in rete prima di un aggregazione parziale, si trasferirebbe l'intero dataset. Ogni executor, invece, trasferisce una tupla per ogni valore univoco della colonna di aggregazione (*destinatario* nel nostro esempio).

4.4.3 Modalità di esecuzione dell'ordinamento

L'operazione di ordinamento viene fatta in due fasi. Prima di effettuare un vero e proprio ordinamento, si trasmettono i dati in rete sulla base di una funzione di ripartizione monotona (RangePartitioning), per poi effettuare un ordinamento locale nei singoli esecutori. Grazie alla monotonia della funzione di ripartizione utilizzata, si ha la garanzia che i valori di un dataset sono, per specifici range di valori, presenti tutti nello stesso executor e che l'ordinamento sarà quindi globale. L'algoritmo di ordinamento utilizzato localmente da Spark è il *TimSort*, che ha sostituito QuickSort dalla versione 1.1 del software.

4.4.4 Gli operatori del piano di esecuzione

Quando si ottiene un piano di esecuzione, attraverso il metodo *toDebugString* chiamato in uno SchemaRDD (ottenuto, ad esempio, come risultato del metodo *sql* di un oggetto *JavaHiveContext*) si avranno nel piano degli operatori che descrivono come viene eseguita l'interrogazione. Alcuni di tali operatori, presenti nei piani di esecuzione delle interrogazioni presenti in Appendice A, sono di seguito descritti. I piani di esecuzione sono riportati, in forma testuale, in Appendice B, mentre sono rappresentati in forma grafica nelle figure 5.4, 5.5 e 5.6, con gli operatori riportati tra parentesi di seguito. Nella rappresentazione grafica, le linee e gli operatori blu indicano la presenza di un trasferimento in rete.

Gli **HiveTableScan** (*TABLESCAN*) sono le letture dei dati da tabelle di un database. Vengono letti meno dati possibili. Ad esempio quando si utilizza il formato Parquet si sfrutta la memorizzazione colonnare leggendo solo le colonne di interesse per la query.

Le operazioni **Filter** (σ) sono delle operazioni che permettono di selezionare tuple di un dataset sulla base di un predicato di selezione. Per svolgere questa operazione si scorre ogni partizione dell'RDD di partenza creando un nuovo RDD con lo stesso numero di partizioni, che solitamente contengono meno tuple.

Le operazioni **Project** (Π) permettono di selezionare le colonne di interesse di uno SchemaRDD. Per ogni tupla del dataset, di tipo Row, si estraggono i valori necessari per creare il nuovo SchemaRDD.

L'operatore **Exchange** (*linee o operatori blu*) trasferisce i dati in rete sulla base di una certa funzione che mappa le tuple a numeri interi, che indicano il numero di partizione su cui verranno memorizzati i dati. Solitamente si utilizza un HashPartitioning, che mappa i valori secondo una funzione hash, o un RangePartitioning, che preserva l'ordine delle tuple per eseguire un ordinamento.

L'operatore **Aggregate** (Σ) aggrega i dati, ad esempio per eseguire somme o prendere il massimo di una colonna. Di solito, per ogni aggregazione dell'SQL, ci sono due aggregazioni nel piano di esecuzione: la prima è un'aggregazione parziale nei singoli nodi (Aggregate true), mentre la seconda è l'aggregazione globale (Aggregate false) fatta dopo lo shuffle dei risultati parziali.

Lo **ShuffledHashJoin** ($\bowtie_s$) effettua il join tra due tabelle, prima trasferite attraverso un **Exchange**. La tabella indicata come build table sarà quella sulla quale verrà creta un tabella hash in memoria, mentre l'altra viene scorsa per effettuare il join. Un'alternativa a questo tipo di join è il **BroadcastHashJoin** ($\bowtie_b$), nel quale la build table viene inviata per intero a tutti gli esecutori del

cluster.

Per eseguire un ordinamento delle partizioni si utilizza l'operatore **Sort** (*S*). Se applicato dopo un RangePartitioning, si avrà un ordinamento globale.

L'operazione **TakeOrdered** (*S+L*) prende i primi n elementi in base all'ordinamento richiesto. Ciò è equivalente ad applicare l'operatore **Limit** dopo aver effettuato un ordinamento (**Sort**).

L'inserimento di dati in una tabella viene indicato dall'operatore **InsertIntoHiveTable**, che indica su quale database e su quale tabella si inseriscono dati. Il valore booleano indica se vengono sovrascritti i dati precedentemente memorizzati (*overwrite*).

Capitolo 5

Valutazione delle performance

Il progetto che si è sviluppato è stato svolto con l'intento di confrontare le performance e le caratteristiche dei sistemi Spark e Hive. Spark, tramite il modulo Spark SQL, permette di eseguire query scritte in HiveQL sfruttando il core di Spark per la loro esecuzione. Hive, invece, trasforma le query in dei job MapReduce eseguiti in sequenza. É in corso d'opera l'implementazione del progetto *Hive On Spark*, che permetterà ad Hive di scegliere Spark come "motore" per eseguire le query. Come vedremo, l'utilizzo efficiente della memoria porta ad un aumento delle prestazioni consistente, a volte anche di un ordine di grandezza.

I test sono stati effettuati utilizzando un benchmark standard, TPC-H, un benchmark che simula un sistema di supporto alle decisioni mandando in esecuzione specifiche interrogazioni. Le query e i dati sono stati scelti per essere rilevanti, cioè in modo tale che i risultati ottenuti dal benchmark siano un buon indice delle prestazioni di un DBMS in un utilizzo reale. Le query sono molto complesse ed eseguite su grandi volumi di dati, dando risposte a problemi critici di interesse industriale.

Sono fornite 22 query, che abbiamo eseguito sia su Hive che su Spark SQL in database grandi 1, 10 e 100 GB, memorizzati sia in formato Avro che Parquet. Tre query più semplici sono state eseguite anche in database da 1 TB.

Lo schema del database è illustrato, senza tipi di dato, nella figura 5.1. Per una descrizione completa dello schema del database si rimanda alle specifiche ufficiali [20].

Il cluster su cui sono stati eseguiti i benchmark è composto da 7 nodi di commodity hardware, in cui ogni nodo ha le seguenti caratteristiche tecniche:

- **CPU:** Intel i7-4790, 4 core, 8 threads, 3.6 Ghz
- **RAM:** 32 GB

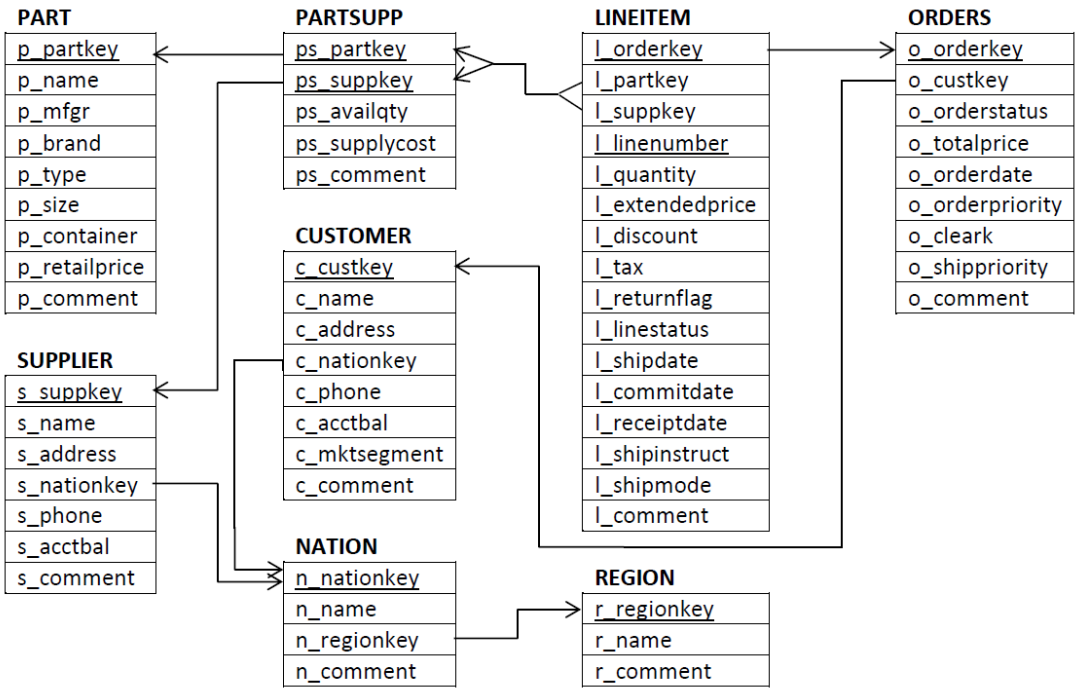

Figura 5.1: Lo schema del database usato per il benchmark

- **HARD-DRIVE:** 2 x 2 TB HDD
- **Ethernet:** Gigabit
- **Sistema operativo:** CentOS 6.6 (Linux)

5.1 Le interrogazioni

Le ventidue interrogazioni utilizzate sono una modifica delle query originali, scritte in SQL standard, per funzionare in HiveQL [21](si ricorda che, ad esempio, HiveQL non ha pieno supporto per le query innestate). Non verranno riportate tutte le 22 query, che sono molto lunghe e complesse, mentre tre query da noi selezionate, la numero 1, la numero 3 e la numero 6 sono riportate in Appendice A.

5.2 I tempi di esecuzione di Spark

Abbiamo provato due differenti configurazioni di Spark. La prima configurazione è quella con 6 executors che sfruttano 7 core ciascuno e 21 GB di RAM, mentre la seconda configurazione usa 12 executors con 4 core e 4 GB di RAM allocati (ovviamente ci sarà più di un executor per nodo). Il motivo per cui non sono stati assegnati 8 core nella prima configurazione è che il *driver* di Spark usa un core su una macchina, e su quella macchina non si sarebbe potuto mettere un container con allocati 8 core. Le risorse complessive utilizzate sono quindi maggiori rinunciando di utilizzare un core piuttosco che rinunciando a mettere un executor in una macchina, anche perchè i fattori limitanti le prestazioni di Spark sembrano essere la velocità dei dischi fissi e della rete.

Tabella 5.1: Tempi di esecuzione di Spark SQL (6 executors, 7 core, 21 GB di RAM), formato Avro

Query	1 GB	10 GB	100 GB	1 TB
1	46.27	63.75	190.54	1,692.92
2	72.03	78.38	117.34	
3	47.19	60.08	217.21	4,573.77
4	63.73	78.70	248.25	
5	53.62	69.26	303.08	
6	42.76	49.23	175.60	1,678.68
7	55.26	70.55	286.71	
8	53.29	71.12	388.64	
9	66.00	84.29	397.68	
10	48.53	57.87	207.54	
11	90.29	97.47	123.86	
12	49.13	54.46	205.04	
13	49.77	53.50	92.25	
14	44.58	49.44	161.08	
15	59.25	65.99	146.37	
16	74.41	83.51	/	
17	64.04	91.08	586.34	
18	64.17	84.18	604.95	
19	47.85	64.11	265.77	
20	105.28	117.15	255.09	
21	85.74	123.00	862.81	
22	67.80	76.81	144.62	
TOT	**1,351.07**	**1,644.01**	5,980.86	7,945.37
Inc		**1.21x**	3.83x	21.70x

Verranno ora illustrati i tempi di esecuzione ottenuti dalla prima configurazione (6 executors, 7 core, 21 GB di RAM). I tempi in secondi impiegati nell'esecuzione delle 22 query sul formato Avro sono riportati in Tabella 5.1, dove l'incremento (**Inc**) del tempo di esecuzione è calcolato rispetto al database 10 volte più piccolo, sui risultati comuni.

La query 16 fallisce sul database da 100 GB, dato che Spark in quella query non riesce a fare un hash join mettendo una tabella in memoria dopo aver trasferito le due tabelle di join in rete. Maggiori informazioni su come Spark fa i join sono date nel capitolo precedente. Nel formato Parquet si hanno invece i tempi di esecuzione riportati in Tabella 5.2.

Si può notare come le differenze prestazionali tra Avro e Parquet siano rilevanti solo sui database più grandi (100 GB e 1 TB). I motivi delle differenze

Tabella 5.2: Tempi di esecuzione di Spark SQL (6 executors, 7 core, 21 GB di RAM), formato Parquet

Query	1 GB	10 GB	100 GB	1 TB
1	45.90	48.88	87.82	453.80
2	74.00	78.89	93.17	
3	47.81	52.51	109.36	1,646.56
4	63.67	65.07	111.59	
5	55.32	65.72	225.91	
6	42.57	42.24	69.83	347.25
7	55.29	63.75	176.72	
8	53.42	62.85	204.12	
9	67.99	81.49	248.17	
10	49.32	53.38	93.06	
11	94.59	94.52	108.33	
12	48.16	52.16	97.60	
13	51.79	52.10	83.18	
14	43.96	45.10	62.98	
15	60.66	61.89	80.40	
16	77.34	80.46	/	
17	63.84	86.62	406.03	
18	64.81	79.58	300.19	
19	48.89	60.95	207.22	
20	102.89	110.05	166.81	
21	87.09	123.79	560.38	
22	69.10	71.63	85.91	
TOT	**1,368.49**	**1,533.72**	3,578.85	2,447.63
Inc		**1.12x**	2.46x	9.16x

Tabella 5.3: Tempi di esecuzione di Spark SQL (12 executors, 4 core, 4 GB di RAM), formato Avro

Query	1 GB	10 GB	100 GB	1 TB
1	46.27	48.73	186.67	1,741.37
2	72.03	68.84	109.83	
3	47.19	53.29	253.44	/
4	63.73	65.28	239.14	
5	53.62	60.5	/	
6	42.58	37.95	118.1	1,690.62
7	55.26	63.22	605.32	
8	53.29	62.45	/	
9	66	77.69	/	
10	48.53	48.66	199.11	
11	90.29	87.39	114.71	
12	49.13	44.75	188.75	
13	49.77	44.35	87.3	
14	44.58	39.17	140.86	
15	59.25	57.23	138.85	
16	74.41	71.16	/	
17	64.04	87.27	771.85	
18	64.17	79.48	/	
19	47.85	55.77	286.87	
20	105.28	110.14	/	
21	85.74	125.02	/	
22	67.8	67.67	123.88	
TOT	**1,350.81**	**1,456.01**	3,564.68	7,945.37
Inc		**1.07x**	4.10x	11.26x

prestazionali sono che Avro memorizza le tabelle riga per riga, mentre Parquet le memorizza colonna per colonna. Ciò fa sì che se servono solo alcune colonne, utilizzando Parquet si leggono meno dati da disco. Ad accentuare ciò c'è il fatto che Parquet comprime a blocchi i dati contenuti nelle colonne, mentre Avro memorizza le celle semplicemente in forma binaria. La figura 5.2 mostra le differenze sui tempi di esecuzione tra Avro e Parquet su database da 100 GB. Il guadagno utilizzando Parquet risulta essere di circa il 40%, mentre è tracurabile nei database più piccoli.

Con la seconda configurazione (12 executors, 4 core, 4 GB di RAM ciascuno) si hanno dei tempi di esecuzione molto simili (Tabella 5.3). Si ha un maggior numero di interrogazioni non riuscite, dovuto al fatto che Spark fallisce se non riesce a fare gli hash join mettendo una tabella in memoria. Avere

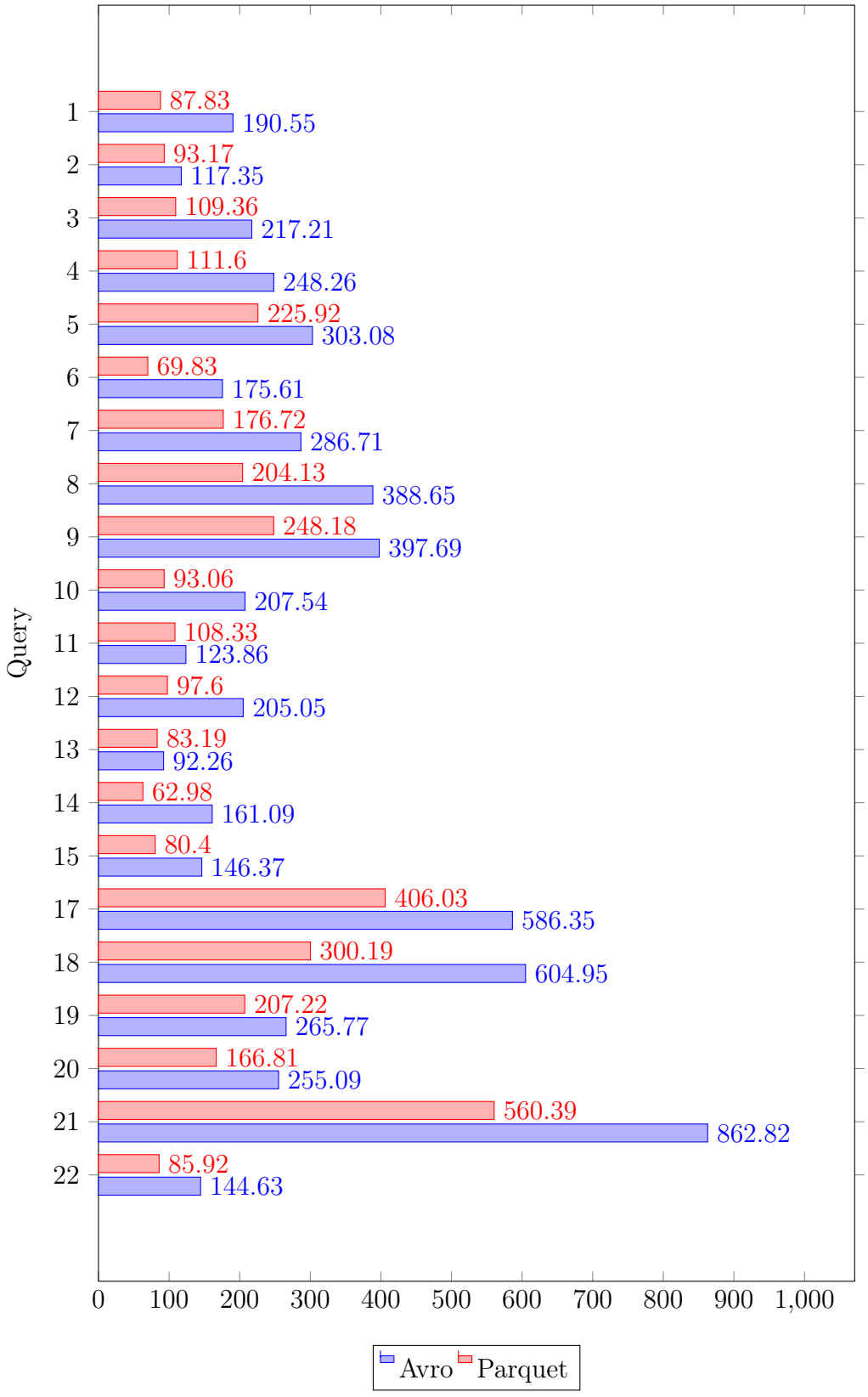

Figura 5.2: Differenza tra Avro e Parquet su database da 100 GB in Spark SQL

Tabella 5.4: Tempi di esecuzione di Spark SQL (12 executors, 4 core, 4 GB di RAM), formato Parquet

Query	1 GB	10 GB	100 GB	1 TB
1	45.90	53.65	84.15	466.81
2	74.00	66.78	88.64	
3	47.81	44.48	122.20	/
4	63.67	57.54	118.03	
5	55.32	57.49	22,971.00	
6	42.57	32.01	52.25	302.80
7	55.29	56.36	251.49	
8	53.42	55.32	/	
9	67.99	76.60	1,220.45	
10	49.32	45.45	101.26	
11	94.59	87.69	104.18	
12	48.16	42.01	96.11	
13	51.79	47.44	83.03	
14	43.96	36.04	54.37	
15	60.66	52.00	72.54	
16	77.34	72.42	/	
17	63.84	86.07	/	
18	64.81	75.24	/	
19	48.89	51.86	29,151.89	
20	102.89	101.21	204.98	
21	87.09	128.11	/	
22	69.10	64.34	84.52	
TOT	**1,368.49**	**1,390.21**	54,861.18	2,447.63
Inc		**1.01x**	56.38x	5.64x

meno memoria a disposizione vuol dire riuscire ad eseguire un minor numero di query complesse, mentre quando la RAM allocata agli executor è sufficiente, i tempi di esecuzione sono solitamente simili a quelli ottenuti con l'altra configurazione.

I tempi in secondi impiegati nell'esecuzione delle 22 query sul formato Parquet sono riportati in Tabella 5.4. Come con l'altra configurazione, le differenze tra Avro e Parquet sono marcate solo su database grandi.

5.3 I tempi di esecuzione di Hive

Hive trasforma le query, scritte in HiveQL, in dei job MapReduce. Dato che MapReduce scrive obbligatoriamente i dati su disco tra un job e l'altro,

Tabella 5.5: Tempi di esecuzione di Hive (25 container, 2 core, 2 GB di RAM), formato Avro

Query	1 GB	10 GB	100 GB	1 TB
1	96.39	372.55	3,429.42	38,721.29
2	141.12	227.36	1,042.51	
3	140.15	568.05	5,153.81	58,024.24
4	169.43	627.58	5,476.24	
5	195.29	572.41	4,318.51	
6	54.25	255.56	2,426.24	26,105.34
7	212.63	933.58	8,795.79	
8	209.85	405.87	2,049.66	
9	221.40	663.72	3,371.39	
10	167.60	470.27	3,800.61	
11	144.90	162.87	315.97	
12	120.57	433.77	3,994.27	
13	134.56	257.67	1,500.05	
14	80.73	313.59	2,931.94	
15	119.94	350.12	3,969.99	
16	146.22	266.86	1,353.73	
17	152.83	930.73	9,153.96	
18	235.37	1,105.89	10,268.33	
19	92.15	629.76	6,071.59	
20	188.82	503.74	4,147.04	
21	381.01	1,597.09	14,574.61	
22	155.31	185.63	1,180.37	
TOT	**3,560.63**	**11,834.76**	**99,326.15**	122,850.87
Inc		**3.32x**	**8.39x**	11.15x

questo processo risulta essere molto più lento rispetto a quello che fa Spark, che cerca il più possibile di lavorare in memoria centrale.

I test sono stati effettuati usando 25 container con 2 core e 2 GB di RAM ciascuno. Uno di tali container è l'ApplicationMaster, cioè colui che alloca le risorse e gestisce i compiti che devono svolgere gli altri container.

I tempi di esecuzione sui database da 1, 10 e 100 GB ottenuti eseguendo le 22 query su database memorizzati in formato Avro sono riportati in Tabella 5.5. Malgrado i piccoli container, Hive non fallisce nell'esecuzione delle query come farebbe Spark, in quanto non forza l'utilizzo della memoria per fare i join.

I tempi di esecuzione ottenuti su database Parquet sono riportati in Tabella 5.6, dalla quale si evince, confrontandola con la Tabella 5.5, che anche con Hive il guadagno prestazionale dovuto all'utilizzo del formato Parquet è rilevante solo a partire dal database da 100 GB.

Confrontando i tempi di esecuzione ottenuti eseguendo le stesse query in

Tabella 5.6: Tempi di esecuzione di Hive (25 container, 2 core, 2 GB di RAM), formato Parquet

Query	1 GB	10 GB	100 GB	1 TB
1	79.65	194.74	1,680.40	16,021.40
2	136.84	224.74	963.45	
3	136.62	475.37	3,693.33	36,077.81
4	155.45	398.92	3,221.76	
5	189.33	458.28	3,219.69	
6	40.09	94.89	753.571	7,209.99
7	206.44	668.18	6,928.22	
8	202.11	325.10	1,379.55	
9	340.59	576.94	2,433.33	
10	159.00	359.15	2,265.62	
11	145.32	157.90	275.30	
12	111.05	272.57	2,085.01	
13	138.75	265.05	1,455.31	
14	66.09	147.71	1,047.82	
15	106.33	180.94	2,081.93	
16	149.08	279.27	1,386.57	
17	144.82	820.50	7,366.18	
18	231.99	1,142.66	9,855.87	
19	88.61	596.06	5,253.87	
20	177.23	320.53	2,024.75	
21	377.09	1,263.73	10,230.00	
22	159.00	177.66	1,058.83	
TOT	**3,541.58**	**9,400.98**	**70,660.37**	23,231.39
Inc		**2.65x**	**7.52x**	9.68x

questi due sistemi diversi, si nota facilmente come Spark SQL sia nettamente più veloce di Hive. Le differenze prestazionali, mostrate in figura 5.3 per quando riguarda il database Parquet da 100 GB, sono nette: Spark riesce ad essere mediamente 19 volte più veloce rispetto ad Hive. Sugli altri database si hanno differenze prestazionali simili.

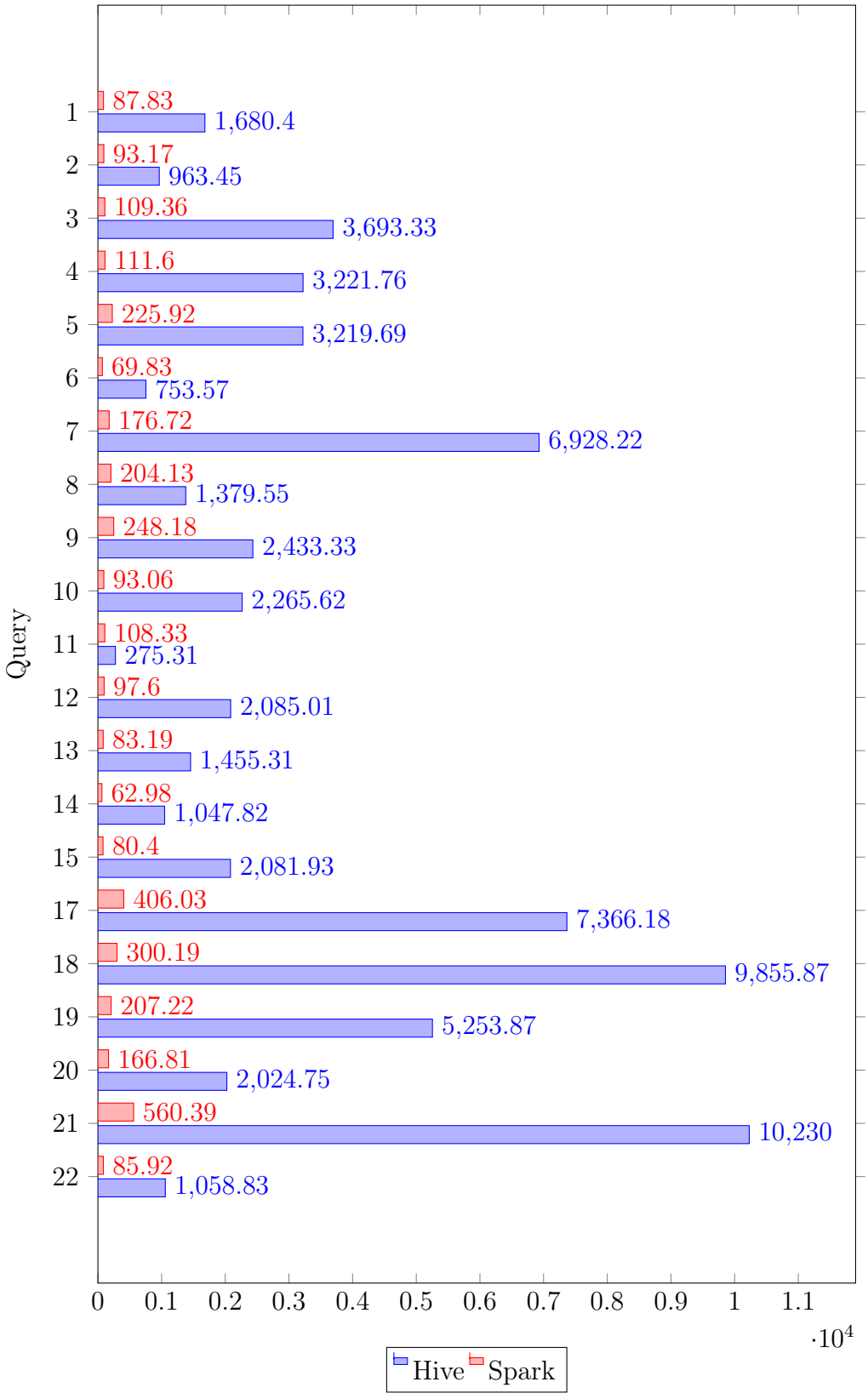

Figura 5.3: Hive vs Spark SQL, 100 GB Parquet

5.4 I piani di esecuzione

Sono ora presentati i piani di esecuzione per le tre query selezionate in forma grafica, utilizzando gli operatori dell'algebra relazionale spiegati nella sottosezione 4.4.4. I piani di esecuzione in forma testuale sono riportati in Appendice B.

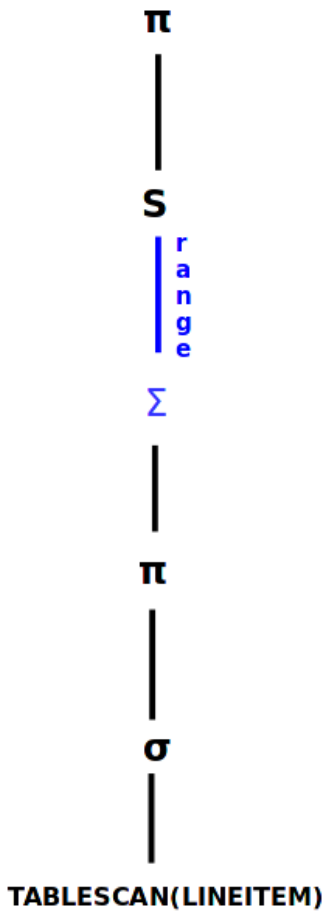

Figura 5.4: Piano di esecuzione query 1

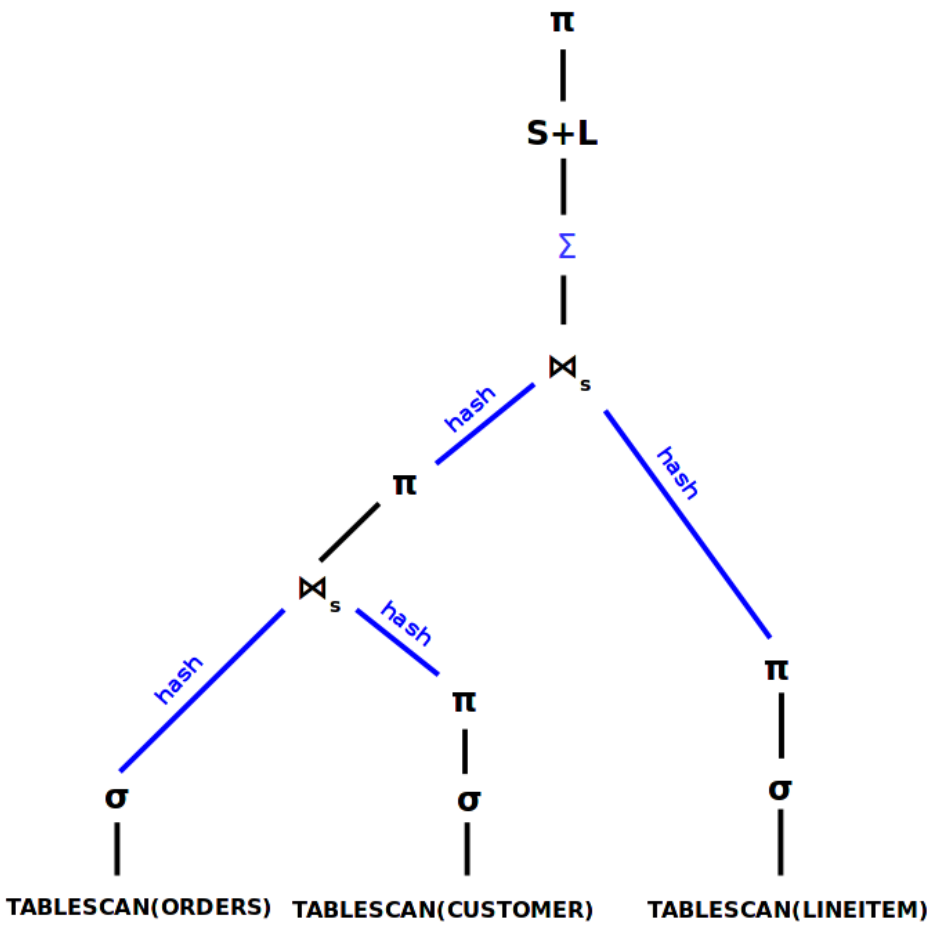

Figura 5.5: Piano di esecuzione query 3

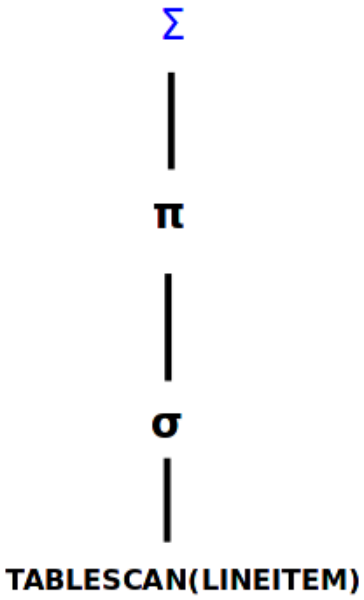

Figura 5.6: Piano di esecuzione query 6

Conclusioni

L'avvento dei Big Data ha portato negli ultimi anni alla ricerca di nuove soluzioni per la loro memorizzazione e per la loro analisi. Hadoop è un sistema software open-source molto utilizzato in questo ambito, che offre sia un filesystem distribuito per la memorizzazione di informazioni che una piattaforma per la loro elaborazione.

Vengono supportati molteplici software per l'analisi di dati, tra cui MapReduce e Spark. La differenza sostanziale tra questi due sistemi è che MapReduce obbliga a memorizzare i dati su disco dopo ogni iterazione, mentre Spark può lavorare in memoria centrale, sfruttando il disco solo in caso di necessità.

Il sistema Spark, che è un framework di alto livello, offre un insieme di moduli specifici per ogni ambito di applicazione, descritti nel secondo capitolo. Noi abbiamo preso in esame il modulo Spark SQL e lo abbiamo confrontato con Hive, un DBMS che propone un'astrazione relazionale a dati memorizzati sul filesystem distribuito di Hadoop. Entrambi i sistemi, leggendo informazioni dallo stesso metadata repository (Hive metastore), possono interrogare i database Hive tramite una variante del linguaggio SQL, HiveQL. La differenza è che Hive si appoggia a MapReduce per l'esecuzione delle query, mentre Spark SQL sfrutta il core di Spark. É in corso di implementazione il progetto Hive on Spark, che permetterà anche ad Hive di sfruttare il core di Spark per risolvere le interrogazioni.

É stato eseguito il benchmark standard TPC-H su entrambi i sistemi software, su databases da 1, 10 e 100 GB. Il benchmark è stato anche eseguito parziamente su database da 1 TB, come spiegato nell'ultimo capitolo della tesi. Dai risultati è emerso chiaramente che Spark è in media 20 volte più veloce di MapReduce, grazie al più ottimizzato utilizzo della memoria centrale.

Dall'analisi del sistema Spark e dal confronto del modulo Spark SQL con Hive è emerso che, una volta risolto il problema di Spark che non gli consente di eseguire i join in casi particolari [17, 18], esso sarà il software da prendere in considerazione per il processamento di dati a discapito di MapReduce, il quale offre un'API più complessa da utilizzare rispetto a quella di Spark. É infatti la complessità di tale API che ha fatto nascere strumenti come Hive e Pig, il

cui intento è di offrire un astrazione a MapReduce tramite linguaggi di alto livello.

Il mondo dei Big Data è in continua evoluzione, in quanto si cercano di avere performance e sicurezza sempre maggiori. Ad esempio nell'ultimo periodo si è vista ad esempio una crescente popolarità degli in-memory databases, tra cui Redis.

Ci si chiede se le piattaforme software per i Big Data soppianteranno i DBMS tradizionali. Sembra che i due tipi di sistemi coesisteranno per un lungo periodo, in quanto i sistemi per i Big Data richiedono un maggior investimento dal punto di vista infrastrutturale, non sempre motivato. Devono pure essere risolti problemi specifici dei sistemi distribuiti, come la rimozione e la modifica sincronizzata di file. Hive sfrutta infatti un DBMS tradizionale per la memorizzazione dei metadati.

Appendice A

Le interrogazioni

In questa appendice sono riportate le query 1, 3 e 6 del benchmark TPC-H, scritte in HiveQL per essere eseguite con Hive e Spark SQL.

A.1 La query 1

```
drop table if EXISTS q1_pricing_summary_report;
--create the target table
CREATE TABLE q1_pricing_summary_report ( L_RETURNFLAG STRING,
 L_LINESTATUS STRING, SUM_QTY DOUBLE,
 SUM_BASE_PRICE DOUBLE, SUM_DISC_PRICE DOUBLE, SUM_CHARGE DOUBLE,
 AVE_QTY DOUBLE, AVE_PRICE DOUBLE,
 AVE_DISC DOUBLE, COUNT_ORDER INT);
--the query
INSERT OVERWRITE TABLE q1_pricing_summary_report
select
    l_returnflag, l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
    sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(1) as count_order
from
```

```
    lineitem
  where
  l_shipdate <= '1998-09-02'
group by l_returnflag,  l_linestatus
order by l_returnflag, l_linestatus;

select * from q1_pricing_summary_report;
```

Questa query può essere vista come un'aggregazione con ordinamento.

A.2 La query 3

```
drop table if exists q3_shipping_priority;

-- create target table
create table q3_shipping_priority (l_orderkey int,
 revenue double, o_orderdate string, o_shippriority int);
-- the query
insert overwrite table q3_shipping_priority
select
   l_orderkey,
   sum(l_extendedprice*(1-l_discount)) as revenue,
   o_orderdate,
   o_shippriority
from
   customer,
   orders,
   lineitem
where
   c_mktsegment = 'BUILDING'
   and c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_orderdate < date '1995-03-15'
   and l_shipdate > date '1995-03-15'
group by
   l_orderkey,
   o_orderdate,
   o_shippriority
order by
   revenue desc,
   o_orderdate
```

```
    limit 10;

select * from q3_shipping_priority;
```

Questa query è più complessa della precedente, in quanto bisogna fare il join tra tre tabelle, oltre ad aggregare ed ordinare i dati.

A.3 La query 6

```
drop table if exists q6_forecast_revenue_change;

--create the target table
create table q6_forecast_revenue_change (revenue double);

--the query
insert overwrite table q6_forecast_revenue_change
select sum(l_extendedprice*l_discount) as revenue
from lineitem
where l_shipdate >= '1994-01-01'
    and l_shipdate < '1995-01-01'
    and l_discount >= 0.05
    and l_discount <= 0.07
    and l_quantity < 24;

select * from q6_forecast_revenue_change;
```

Questa è una query molto semplice, richiedendo la somma di una colonna selezionando alcune tuple della tabella più grande del database.

Appendice B

I piani di esecuzione

Vengono in questa appendice riportati i piani di esecuzione di Spark SQL per le query 1, 3 e 6 del banchmark TPC-H. Per una corretta comprensione dei piani, si legga il capitolo 4 di questo elaborato.

B.0.1 Piano di esecuzione della query 1

```
InsertIntoHiveTable (MetastoreRelation tpch_100gb,
   q1_pricing_summary_report, None), Map(), true
 Project [l_returnflag#404,l_linestatus#405,sum_qty#378,
     sum_base_price#379,sum_disc_price#380,sum_charge
    #381,
  avg_qty#382,avg_price#383,avg_disc#384, CAST(
     count_order#385L, IntegerType) AS count_order#412]
  Sort [l_returnflag#404 ASC,l_linestatus#405 ASC], true
   Exchange (RangePartitioning [l_returnflag#404 ASC,
      l_linestatus#405 ASC], 200)
    Aggregate false, [l_returnflag#404,l_linestatus
       #405], [l_returnflag#404,l_linestatus#405,SUM(
       PartialSum#424) AS
     sum_qty#378,SUM(PartialSum#425) AS sum_base_price
        #379, UM(PartialSum#426) AS
     sum_disc_price#380,SUM(PartialSum#427) AS
        sum_charge#381,(CAST(SUM(PartialSum#428),
        DoubleType) /
     CAST(SUM(PartialCount#429L), DoubleType)) AS
        avg_qty#382,(CAST(SUM(PartialSum#430),
```

```
    DoubleType) /
  CAST(SUM(PartialCount#431L), DoubleType)) AS
    avg_price#383,(CAST(SUM(PartialSum#432),
    DoubleType) /
  CAST(SUM(PartialCount#433L), DoubleType)) AS
    avg_disc#384,Coalesce(SUM(PartialCount#434L),0)
    AS count_order#385L]
  Exchange (HashPartitioning [l_returnflag#404,
    l_linestatus#405], 200)
   Aggregate true, [l_returnflag#404,l_linestatus
     #405], [l_returnflag#404,l_linestatus#405,COUNT
     (l_discount#402) AS
    PartialCount#433L,SUM(l_discount#402) AS
      PartialSum#432,SUM(l_extendedprice#401) AS
    PartialSum#425,COUNT(l_extendedprice#401) AS
      PartialCount#431L,SUM(l_extendedprice#401) AS
    PartialSum#430,SUM(l_quantity#400) AS PartialSum
      #424,
    COUNT(l_quantity#400) AS PartialCount#429L,SUM(
      l_quantity#400) AS PartialSum#428,
    SUM((l_extendedprice#401 * (1.0 - l_discount#402)
      )) AS PartialSum#426,
    SUM(((l_extendedprice#401 * (1.0 - l_discount
      #402)) * (1.0 + l_tax#403)))
    AS PartialSum#427,COUNT(1) AS PartialCount#434L]
    Project [l_tax#403,l_quantity#400,l_returnflag
      #404,l_extendedprice#401,l_linestatus#405,
      l_discount#402]
     Filter (CAST(l_shipdate#406, StringType) <=
       1998-09-02)
      HiveTableScan [l_tax#403,l_shipdate#406,
        l_quantity#400,l_returnflag#404,
        l_extendedprice#401,
      l_linestatus#405,l_discount#402], (
        MetastoreRelation tpch_100gb, lineitem, None
        ), None []
```

B.0.2 Piano di esecuzione della query 3

```
InsertIntoHiveTable (MetastoreRelation tpch_100gb,
  q3_shipping_priority, None), Map(), true
```

```
Project [l_orderkey#496,revenue#474,CAST(o_orderdate
    #491, StringType) AS o_orderdate#512,o_shippriority
    #494]
 TakeOrdered 10, [revenue#474 DESC,o_orderdate#491 ASC]
  Aggregate false, [l_orderkey#496,o_orderdate#491,
      o_shippriority#494], [l_orderkey#496,SUM(
      PartialSum#514) AS revenue#474,o_orderdate#491,
      o_shippriority#494]
   Exchange (HashPartitioning [l_orderkey#496,
       o_orderdate#491,o_shippriority#494], 200)
    Aggregate true, [l_orderkey#496,o_orderdate#491,
        o_shippriority#494], [l_orderkey#496,o_orderdate
        #491,o_shippriority#494,SUM((l_extendedprice#501
         * (1.0 - l_discount#502))) AS PartialSum#514]
     Project [l_orderkey#496,o_orderdate#491,
         o_shippriority#494,l_extendedprice#501,
         l_discount#502]
      ShuffledHashJoin [o_orderkey#487], [l_orderkey
          #496], BuildRight
       Exchange (HashPartitioning [o_orderkey#487],
           200)
        Project [o_shippriority#494,o_orderkey#487,
            o_orderdate#491]
         ShuffledHashJoin [c_custkey#479], [o_custkey
             #488], BuildLeft
          Exchange (HashPartitioning [c_custkey#479],
              200)
           Project [c_custkey#479]
            Filter (c_mktsegment#485 = BUILDING)
             HiveTableScan [c_custkey#479,c_mktsegment
                 #485], (MetastoreRelation tpch_100gb,
                 customer, None), None
          Exchange (HashPartitioning [o_custkey#488],
              200)
           Filter (o_orderdate#491 < 1995-03-15)
            HiveTableScan [o_shippriority#494,
                o_orderkey#487,o_orderdate#491,o_custkey
                #488], (MetastoreRelation tpch_100gb,
                orders, None), None
       Exchange (HashPartitioning [l_orderkey#496],
           200)
```

```
          Project [l_orderkey#496,l_extendedprice#501,
              l_discount#502]
           Filter (l_shipdate#506 > 1995-03-15)
            HiveTableScan [l_orderkey#496,l_extendedprice
                #501,l_discount#502,l_shipdate#506], (
                MetastoreRelation tpch_100gb, lineitem,
                None), None []
```

B.0.3 Piano di esecuzione della query 6

```
InsertIntoHiveTable (MetastoreRelation tpch_100gb,
    q6_forecast_revenue_change, None), Map(), true
 Aggregate false, [], [SUM(PartialSum#545) AS revenue
     #526]
  Exchange SinglePartition
   Aggregate true, [], [SUM((l_extendedprice#533 *
       l_discount#534)) AS PartialSum#545]
    Project [l_extendedprice#533,l_discount#534]
     Filter (((((CAST(l_shipdate#538, StringType) >=
         1994-01-01) && (CAST(l_shipdate#538, StringType)
          < 1995-01-01)) && (l_discount#534 >= 0.05)) &&
         (l_discount#534 <= 0.07)) && (l_quantity#532 <
         24.0))
      HiveTableScan [l_extendedprice#533,l_discount#534,
          l_shipdate#538,l_quantity#532], (
          MetastoreRelation tpch_100gb, lineitem, None),
          None []
```

Bibliografia

[1] Il cluster di Yahoo!, http://www.techrepublic.com/article/why-the-worlds-largest-hadoop-installation-may-soon-become-the-norm/

[2] Architettura di HDFS, https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[3] Popolarità di Spark, https://databricks.com/blog/2015/01/27/big-data-projects-are-hungry-for-simpler-and-more-powerful-tools-survey.html

[4] Record ordinamento, http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

[5] Configurazione di Spark, https://spark.apache.org/docs/1.2.1/configuration.html

[6] Spark su YARN, https://spark.apache.org/docs/1.2.1/running-on-yarn.html

[7] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

[8] Documentazione ufficiale Java API, https://spark.apache.org/docs/1.2.1/api/java/index.html

[9] Iteratori in Scala, http://www.scala-lang.org/api/2.11.5/index.html#scala.collection.Iterator

[10] Guida alla programmazione in Spark, https://spark.apache.org/docs/1.2.1/programming-guide.html

[11] Spark SQL, https://spark.apache.org/docs/1.2.1/sql-programming-guide.html

[12] Sort by (HiveQL), https://cwiki.apache.org/confluence/display/Hive/LanguageManual+SortBy

[13] Autoincrement in HiveQL, https://issues.apache.org/jira/browse/HIVE-6905

[14] Sottoquery in HiveQL, https://cwiki.apache.org/confluence/display/Hive/LanguageManual+SubQueries

[15] Funzionalità di Hive non supportate in Spark SQL, https://spark.apache.org/docs/1.2.1/sql-programming-guide.html#unsupported-hive-functionality

[16] Il tipo Row di Spark SQL, https://spark.apache.org/docs/1.2.1/api/java/org/apache/spark/sql/api/java/Row.html

[17] Skewed join, https://issues.apache.org/jira/browse/SPARK-4644

[18] Sort-merge join, https://issues.apache.org/jira/browse/SPARK-5763

[19] Descrizione formato Paquet, https://blog.twitter.com/2013/dremel-made-simple-with-parquet

[20] Specifiche del benchmark TPC-H, http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf

[21] Le query del benchmark TCP-H scritte in HiveQL, https://issues.apache.org/jira/secure/attachment/12416257/TPC-H_on_Hive_2009-08-11.pdf

Ringraziamenti

Ringrazio innanzitutto il professor Matteo Golfarelli per avermi concesso di realizzare questa tesi.

Ringrazio anche il correlatore Lorenzo Baldacci per l'energia usata per risolvere i problemi e i dubbi che sono emersi durante la realizzazione del progetto di tesi. Il suo aiuto è stato fondamentale.

Un ringraziamento speciale va alla mia famiglia ed agli amici che mi hanno supportato durante questi anni di Università. Loro mi sono stati vicini in questi tre anni e lo che lo continueranno ad essere nel proseguimento degli studi.

L'ultimo ringraziamento lo vorrei dare all'Università, che ha messo a disposizione il cluster Hadoop, senza il quale non avrei potuto svolgere questo elaborato.