ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Sviluppo di un modello di costo per Spark Sql

Tesi in
Sistemi Informativi E Business Intelligence

Relatore:
Prof. MATTEO GOLFARELLI

Presentata da:
FABIO CONSALICI

Co-relatore:
Dott. LORENZO BALDACCI

ANNO ACCADEMICO 2014-2015
SESSIONE III

Alla mia famiglia e a Valentina
per il sostegno che mi hanno dato

Sommario

Introduzione

La quantità di dati che vengono generati e immagazzinati sta aumentando sempre più grazie alle nuove tecnologie e al numero di utenti sempre maggiore. Questa tendenza è stata spinta da diversi fattori nel corso degli anni. Inizialmente dall'avvento di internet che ha permesso di condividere informazioni con tutto il mondo rivoluzionando non solo la gestione delle informazioni ma anche la loro quantità. In secondo luogo sono sorti i social network che hanno portato gli utenti a trasferire sul web una grande quantità di informazioni personali ed ora anche il mondo dell'Internet Of Things, grazie al quale anche i sensori e le macchine risultano essere agenti sulla rete. Solo negli ultimi anni però si è compreso il vero valore di questi dati. All'interno dei giganteschi dataset possono essere presenti informazioni che per un'azienda rappresentano un vero e proprio strumento non solo per migliorare il rapporto col cliente o la sua gestione interna ma anche per avere un vantaggio verso aziende avversarie. Questi dati, elaborati correttamente, permettono quindi di ottenere delle informazioni di valore strategico che aiutano nell'effettuare decisioni aziendali a qualsiasi livello, dalla produzione fino al marketing.

Il vero problema risiede però proprio nell'elaborazione di tali dati, che per le loro caratteristiche non possono essere gestiti da strumenti tradizionali. Nasce quindi la necessità di nuovi strumenti che possano sfruttare la potenza di calcolo di entità come i cluster che racchiudono sotto un unico sistema un'insieme di host che permettono di effettuarne l'analisi. Sono nati soprattutto negli ultimi anni numerosi framework proprietari e open source che vanno in questa direzione. In particolare tra i più utilizzati e attivi in questo momento a livello open source troviamo Hadoop e Spark che permettono di sfruttare la potenza di un cluster di commodity hardware per l'elaborazioni di grandi dataset. In particolare Spark garantisce elevate prestazioni date dalla possibilità di elaborare i dati direttamente in memoria centrale, e fornisce un'interfacciamento con il mondo SQL tramite un'apposito componente: Spark SQL. Questo modulo oltre a fornire un'interfaccia SQL a Spark gli permette anche di lavorare con dati strutturati ed eseguire query sugli stessi.

Proprio a causa del forte interesse della comunità e delle grandi aziende tali sistemi si evolvono molto velocemente e sono presenti numerosi studi che forniscono consigli su come affrontare la programmazione di tali sistemi molto complessi. In particolare sui framework Hadoop e Spark proprio per la loro natura open source si concentrano numerosi studi ma

spesso sono solamente consigli per comprendere il funzionamento di tali sistemi e la loro programmazione. In tutto questo manca quindi una reale funzione di costo che descriva in maniera formale e precisa il loro comportamento.

Obiettivo di questa tesi è realizzare un modello di Spark per formulare una funzione di costo che sia non solo implementabile all'interno dell'ottimizzatore di Spark SQL, che in questo momento ne è sprovvisto, ma anche per poter effettuare delle simulazioni di esecuzione di query su tale sistema. Prima di tutto si è quindi studiato il comportamento di Spark attraverso la documentazione e test effettuati su di esso per comprenderne in maniera dettagliata le principali caratteristiche. In questo modo è stato possibile elaborare un insieme di formule che permettano di definire il volume di dati generati e i tempi richiesti per l'esecuzione di una classe di query tra le più utilizzate nell'ambito OLAP: le query GPSJ. Si vuole quindi in ultimo luogo confrontare i dati ottenuti dal modello con risultati sperimentali ottenuti sfruttando il cluster messo a disposizione dal gruppo di ricerca. Con la presenza di tale modello non solo risulta possibile comprendere in maniera più approfondita il reale comportamento di Spark ma permette anche di programmare applicazioni più efficienti e progettare con maggiore precisione sistemi per la gestione dei dataset che sfruttino tali framework.

L'elaborato è strutturato in 5 capitoli. Nel primo si fa una introduzione al mondo dei Big Data che risulta essere l'ambito principale in cui si colloca la tesi, ai database NoSQL fondamentali nella gestione e nella memorizzazione dei dati a livello di cluster e ad Hadoop e Yarn, framework per la gestione dei cluster e dei Big Data sui quali Spark si poggia. Nel secondo capitolo invece si studia la composizione di Spark sia a livello di moduli da cui è costituito sia in termini di architettura run-time che permette l'esecuzione delle sue applicazioni. Nel terzo capitolo si passa invece alla realizzazione del modello e delle formule che nel loro insieme vanno a formare la funzione di costo. Si sono in particolare affrontate le principali operazioni necessarie per l'esecuzione di una query GPSJ. In questo capitolo viene anche introdotto Catalyst l'ottimizzatore di Spark SQL che determina buona parte del comportamento del sistema soprattutto a livello di ottimizzazioni del piano di esecuzione Infine nel quarto capitolo sono stati eseguiti i test per ottenere i dati sperimentali da confrontare con i dati ottenuti dal modello sia in termini di quantità di dati previsti in lettura scrittura e trasferimento sia in termini di tempi di esecuzione di determinate query.

1 I Big Data

Nel tempo i dati immagazzinati nel mondo sono aumentati sempre più soprattutto negli ultimi tempi con l'esplosione prima di internet e poi dei social network e dei dispositivi mobile che hanno portato alla creazione di raccolte di dati enormi. Un'ulteriore spinta al mondo dei big data sta avvenendo anche grazie all'avvento dell'Internet Of Things che sfruttando la connessione sulla rete di numerosi oggetti, anche di tutti i giorni come frigoriferi condizionatori e sensori, porta alla generazione di una quantità di dati enorme che possono essere memorizzati. Oltre che nella dimensione i dati sono aumentati anche nella loro importanza, proprio perché all'interno di questi dati si nascondono le vere e proprie informazioni che risultano di valore strategico come per esempio grafi di relazioni tra i clienti e i prodotti scovati attraverso l'analisi dei dati geografici e sociali delle transazioni.

Dati i tempi sempre più ristretti per muoversi sul mercato le aziende hanno necessità di effettuare decisioni puntuali e immediate a fronte di un mercato sempre più dinamico. Non c'è da stupirsi quindi che tutte le grandi aziende nel mondo dell'informazione si stiano focalizzando sempre più in questo settore data la sua grande potenzialità e gli innumerevoli vantaggi che esso porta.

I Big Data sono quindi dati che sovrastano la capacità di elaborazione dei sistemi convenzionali. La quantità di dati è diventata troppo grande, si muovono troppo velocemente o non è possibile inserirli nelle strutture del proprio database. Per ottenere valore da questi dati deve essere utilizzata una strada alternativa che permetta di processarli. I big data sono quindi diventati vitali così come la necessità di trovare approcci economici che permettano di gestirne il volume la velocità e la variabilità. Dentro a questi dati sono presenti modelli e informazioni di grande valore nascosti a causa della quantità di lavoro necessaria per poterli estrapolare. Per aziende leader come Google o Microsoft questo potere è alla portata ma considerando la tendenza dei dati ad aumentare sempre più le costringe a valutare metodi alternativi di gestione in futuro. Già oggi però grazie al comodity harware sempre più potente, alle architetture cloud e al software open-source è possibile effettuare la gestione dei big data anche per realtà con disponibilità di risorse inferiori. Anche una piccola start-up può accedere all'elaborazione dei big data attraverso l'affitto ad un prezzo accettabile di ore server sul cloud. Essere in grado di processare ogni elemento di dati in un tempo accettabile rimuove tutte le problematiche legate al campionamento e promuove un approccio investigativo sui dati, in contrasto con la natura piuttosto statica dei report predeterminati.

Le start-up che hanno avuto successo negli ultimi 10 anni sono i principali esempi di Big Data utilizzati come abilitatori per nuovi prodotti e servizi. Per esempio, combinando un grande numero di segnali dalle azioni di un utente e quelle dei suoi amici, Facebook è in grado di creare un esperienza utente fortemente personalizzata e di creare un nuovo tipo di gestione della pubblicità. Non è un caso quindi che la parte del leone nella gestione delle idee e degli strumenti alla base dei big data siano emerse da Google,Yahoo, Amazon e Facebook. Lo svilupparsi dei Big Data nelle aziende porta con se una necessaria controparte: l'agilità. Sfruttare con successo il valore di questi dati richiede esperimenti ed esplorazione. Spesso la creazione di nuovi prodotti e la ricerca di modi per ottenere un vantaggio competitivo richiedono curiosità e prospettiva imprenditoriale.

1.1 Caratteristiche

Come termine generale "Big Data" può essere piuttosto nebuloso, nello stesso modo in cui il termine "cloud" copre numerose tecnologie. I dati di input nei sistemi di gestione dei big data possono provenire da qualunque fonte come social network, log di web server, sensori di traffico, immagini satellitari, transazioni bancarie, canzoni in mp3, il contenuto delle pagine web, documenti governativi, percorsi GPS, telemetria delle automobili, dati finanziari e tante altre ancora. Ma sono tutte queste fonti la stessa cosa? Per chiarire la situazione le tre V di volume, velocità e varietà sono spesso usate per caratterizzare differenti aspetti dei Big Data. Sono delle utili lenti attraverso cui vedere e comprendere la natura dei dati e delle piattaforme software disponibili per sfruttarli. [1] Come si può vedere in Figura 1 le tre V vanno a definire le tre dimensioni che costituiscono i big data passando da valori semplici al centro (tabelle, dati nell'ordine dei Megabyte e un fattore di velocità non fondamentale) a valori più complessi da gestire come le foto i video con quantità di dati nell'ordine dei Petabyte e ad una velocità che può crescere sempre più passando da valutazioni periodiche al tempo reale. Ovviamente il mondo dei Big Data risulta essere molto eterogeneo ma questa classificazione permette una prima distinzione di massima tra le varie categorie che li possono rappresentare.

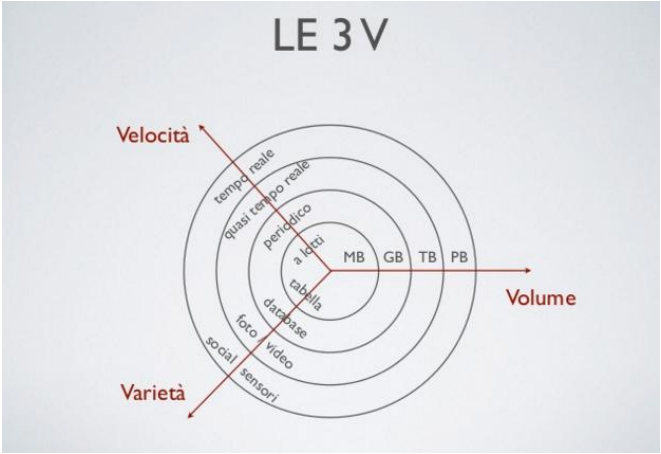

Figura 1 Rappresentazione grafica delle 3V dei Big Data

1.1.1 Volume

Il guadagno ottenuto dalla possibilità di processare grandi quantità di informazioni è la maggiore attrazione delle analisi sui big data. Chi ha più dati spesso riesce a battere anche chi ha un modello migliore: anche una matematica molto semplice può essere incredibilmente efficace se si posseggono grandi quantità di dati. Se fosse possibile effettuare la previsione con un numero di fattori centinaia di volte maggiore la previsione risulterebbe sicuramente più accurata. Il volume rappresenta la sfida più immediata alle convenzionali strutture di IT odierne. Richiede uno storage scalabile e un approccio alle interrogazioni distribuito. Molte compagnie hanno grandi quantità di dati, anche se in forma di log, ma non la capacità di processarli. Queste collezioni di dati raggiungono facilmente valori che vanno da pochi Terabyte a centinaia. Questi valori un tempo impensabili sono stati raggiunti velocemente grazie a diversi fattori. Prima di tutto il numero di utenti che genera dati è aumentato, oltre a tutti gli utenti di sistemi offline, sono aumentati in maniera esponenziale anche gli utenti collegati alla rete: si è passati da 700 milioni di utenti attivi sulla rete nel 2000 a circa 3,2 miliardi nel 2015 [2]. Per ognuno di essi bisogna poi considerare come la navigazione sia cambiata passando da una semplice fruizione di informazioni alla generazione di informazioni sulla rete grazie soprattutto ai social network che hanno portato una quantità enorme di dati condivisi dagli utenti stessi. Basta considerare alcuni esempi per capire come questo aumento di dati sia esponenziale:

- 1,4 miliardi di utenti attivi al mese su Facebook che mettono “mi piace” a 4 milioni di post ogni minuto
- 307 milioni di utenti attivi al mese su Twitter che generano347'000 tweet ogni minuto [3]

- 300 ore di video caricati su Youtube ogni minuto

In più con l'avvento dell'internet of things i dati generati e scambiati non riguardano solo utenti umani ma anche agenti sulla rete di qualsiasi tipologia (automobili, elettrodomestici, sensori ecc ecc) che immettono in maniera continuativa una grandissima mole di dati. [4]

1.1.2 Velocità

L'importanza della velocità dei dati (l'aumentare delle frequenza con cui i dati arrivano sul dataset) ha subito un andamento simile a quello del volume: problemi che precedentemente riguardavano solamente ristretti segmenti dell'industria si presentano ora in un contesto molto più ampio. L'era di internet e del mobile a portato a modificare il modo con cui noi consumiamo e inviamo prodotti e servizi creando un grande flusso di dati sui fornitori. I rivenditori online sono in grado di ricostruire la navigazione degli utenti registrando ogni azione e non solo la vendita finale. Essi sono coloro che riescono ad utilizzare in maniera molto veloce tali informazioni, ad esempio per suggerire altri acquisti. Gli smartphone hanno aumentato ancora la frequenza di dati dato che gli utenti portano con loro uno strumento che fornisce uno stream continuo di immagini, audio, geolocalizzazioni.

Non è solo la semplice velocità delle informazioni che giungono ad essere il problema: si potrebbe immagazzinare tali dati per poi eseguirne successivamente la computazione per esempio. L'importanza sta nella velocità con cui si ottiene un feedback. Nessuno vorrebbe attraversare una strada basandosi solo sulla foto del traffico di pochi secondi prima. Ci sono volte che semplicemente non è possibile attendere più di un determinato intervallo di tempo.

1.1.3 Varietà

Raramente i dati si presentano in una forma perfettamente ordinata e pronta per essere elaborata. Un tema comune nei sistemi di big data è che la sorgente dei dati è spesso diversa e non rientra nelle classiche strutture relazionali. Può essere testo dai social network, immagini, dati inviati direttamente da un sensore ecc ecc. Nessuna di queste fonti genera dati immediatamente integrabili in un'applicazione. Persino sul web, dove la comunicazione computer-computer dovrebbe dare determinate garanzie la realtà risulta piuttosto caotica. Browser differenti inviano dati differenti, gli utenti possono utilizzare diverse versioni dello stesso software e si può essere quasi sicuri che se in una parte del processo sono presenti degli utenti umani molto probabilmente ci saranno degli errori o delle inconsistenze.

Un utilizzo classico dell'elaborazione dei big data è quello di prelevare dati non strutturati e estrarre delle informazioni per essere utilizzate sia da utenti umani sia sottoforma di input strutturato per altre applicazioni. Nonostante la loro popolarità e la natura universalmente conosciuta dei database relazionali questo non è il caso per cui possono sempre essere utilizzati. Certe tipologie di dati sono più adatti a certe tipologie di database. Per esempio, documenti codificati in xml sono più versatili se memorizzati in appositi database dedicati per XML come MarkLogic, le relazioni sui social network sono per loro natura strutture a grafo e quindi database a grafo come Neo4J rendono più semplice ed efficiente la loro gestione. Anche quando non c'è un radicale mismatch tra i dati lo svantaggio dei database relazionali si evidenzia con la natura statica dei suoi schemi. In un ambiente agile e dinamico i risultati della computazione evolvono nel tempo. I database semi-strutturati NoSql forniscono queste caratteristiche di flessibilità: essi forniscono sufficienti strutture per organizzare i dati ma non richiedono lo schema esatto dei dati prima di memorizzarli.

1.2 Database NoSql

NoSql è una famiglia di database management system non-relazionali differenti dai tradizionali sistemi relazionali sotto diversi aspetti. Sono progettati per la gestione di dati distribuiti dove è richiesta una scalabilità molto alta sulla memorizzazione. Il termine venne utilizzato per la prima volta da Carlo Strozzi nel 1998 riferendosi ad un database system che non possedeva un interfaccia sql e al giorno d'oggi con NoSql si intende "not only SQL".

NoSql non è un unico prodotto, ne esistono molteplici e stanno avendo una diffusione sempre maggiore proprio perché i classici database relazionali hanno iniziato a mostrare delle debolezze in certi ambiti la cui importanza sta evolvendo nel tempo come il web e i big data. La memorizzazione di dati provenienti dalla rete e quindi molto caotici, poco strutturati e in quantità enormi poco si presta a database fortemente strutturati come i relazionali, dove per l'appunto la struttura stessa deve essere definita a priori ed in maniera statica.

Un ulteriore problema sorge considerando che in questo momento uno dei pattern più utilizzati dai linguaggi è quello object oriented. Si parla quindi di "object-relational impedence mismatch" riferendosi alle difficoltà tecniche che sorgono quando attraverso un programma scritto con uno stile di programmazione object oriented ci si interfaccia con un RDBMS, in particolare quando gli oggetti o le classi sono mappate in maniera lineare su tabelle del db o su schemi relazionali. La soluzione più utilizzata in questi casi sono gli ORM (Object Relational Mapper) che forniscono i servizi per la memorizzazione dei dati attraverso un'interfaccia orientata agli oggetti. Bisogna però considerare che essi aggiungono

una certa complessità all'applicazione e anche dell'overhead. Per questi ed altri motivi quando si trattano i big data si prediligono sistemi NoSql e tra le caratteristiche principali degli stessi abbiamo:

- Non-relazionali: non utilizzano il modello relazionale
- Open-source: la stragrande maggioranza di questi sistemi sono completamente open-source
- Schema-less: al contrario dei relazionali essi non necessitano uno schema fisso.
- Cluster-friendly: essi sono costruiti da zero per funzionare su grandi cluster realizzati con commodity hardware

1.2.1 Scalabilità

Con l'aumentare del volume dei dati risulta necessario avere anche più risorse a disposizione che permettano di elaborarli. Per fare ciò risulta necessario quindi scalare e questo si può fare su due dimensioni: scaling up o scaling out. Nel primo caso si aumenta la potenza dei singoli server (cpu,ram, rom ecc ecc) ma risulta particolarmente costoso e sono comunque presenti dei limiti fisici. Nel secondo caso invece si aumenta il numero di nodi all'interno del cluster riducendo sensibilmente i costi,purtroppo però i sistemi relazionali mal si adattano a sistemi di questo genere essendo stati progettati focalizzandosi su altre architetture al contrario dei database NoSql.

1.2.2 Categorie

I database NoSql possono essere suddivisi in base alle proprie caratteristiche in 4 principali categorie:

1.2.3 Key-Value database

Nei database key-value i dati sono memorizzati sfruttando gli hashmap si considerano quindi coppie chiave-valore. I key-value sono i più semplici data store NoSql da utilizzare dal punto di vista delle api. L'utente può ottenere un valore dalla rispettiva chiave, inserire un valore per una chiave, o eliminare la chiave dal data store. Il valore corrispondente ad una chiave viene memorizzato come un semplice blob di dati senza avere informazioni specifiche su cosa ci sia al suo interno; è compito dell'applicazione comprendere cosa esso

significhi. Come si vede in Figura 2 per ogni record che andremo a memorizzare abbiamo sulla sinistra la chiave univoca mentre sulla destra il valore vero e proprio che in questo caso è una stringa con il nome dell'utente. Dato che questa tipologia di database utilizza sempre un accesso a chiave primaria, essi risultano avere maggiori performance e permettono di scalare in maniera piuttosto semplice.

Alcuni tra i database key-value più popolari sono: Riak, Redis, Memcached ecc ecc.

Essi hanno comunque diverse differenze tra loro: ad esempio in Memcached i dati non sono persistenti, mentre in Riak si; risulta quindi importante non solo decidere se un database key-value può soddisfare i requisiti ma risulta anche necessario determinare quale specifico database sfruttare.

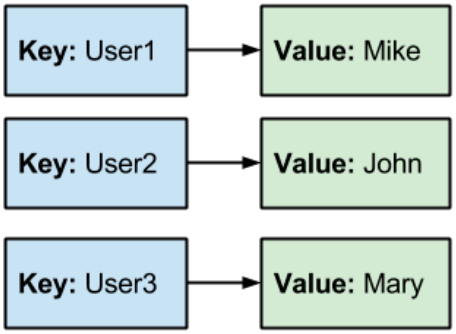

Figura 2 Esempio di gestione con modello key-value

1.2.4 Document database

In questa tipologia di database il concetto base è quello del documento. Il database memorizza e recupera documenti, che possono essere XML, JSON, BSON ecc ecc.

Questi documenti sono strutture gerarchiche ad albero autodescrittive caratterizzate da mappe, collezioni e valori scalari. Nell'esempio in Figura 3 è evidente tale struttura ad albero dove all'interno del document troviamo due attributi (costumer e billingaddress) caratterizzati da ulteriori sotto attributi. I documenti memorizzati sono simili tra loro ma non devono essere necessariamente gli stessi. I document-based db memorizzano i documenti nel campo valore della coppia chiave-valore. I document database come MongoDB forniscono un ricco linguaggio di query e costrutti che permettono una transizione più semplice dai database relazionali.

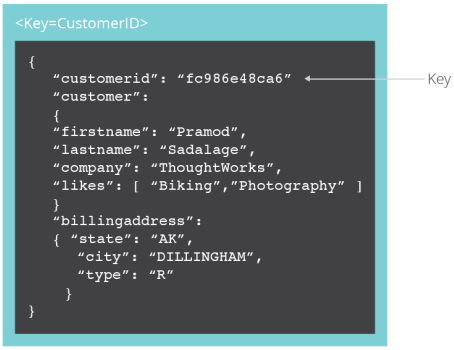

Figura 3 Esempio di gestione con modello document

1.2.5 Column family database

In questa tipologia di database le tuple sono costituite da coppie chiave valore dove la chiave è mappata ad un valore costituito da un insieme di colonne. Le column family sono gruppi di dati relazionati tra loro che vengono spesso utilizzati assieme. Ad esempio per un utente accederemo spesso alle informazioni del suo profilo e più raramente ai suoi ordini. Ogni column family può essere vista come un container di righe in una tabella RDBMS dove la chiave funge da identificativo della riga e la riga consiste in diverse colonne. La differenza è che diverse righe non devono avere le stesse colonne e le colonne possono essere aggiunte in ogni momento senza che debbano essere aggiunte alle altre righe. Quando una colonna è composta da un insieme di colonne allora si ha una super colonna. Una super colonna è costituita da un nome e un valore che è un insieme di colonne. In Figura 4 viene evidenziata tale divisione per cui sotto un'unica column family troviamo più colonne con i loro nomi e valori rispettivamente allo specifico record.

Cassandra è uno dei database column-family più famosi ma ce ne sono tanti altri come HBase, Hypertable e DynamoDb. Cassandra risulta molto veloce e scalabile con le operazioni di scrittura sparse sul cluster. Il cluster non ha un master node e di conseguenza ogni operazione di scrittura e lettura può essere gestita da ogni nodo del sistema.

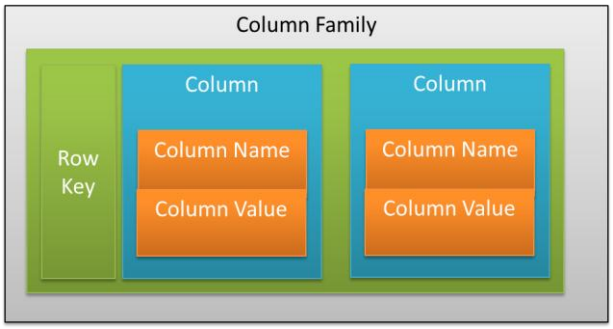

Figura 4 Modello di gestione per categoria column family

1.2.6 Graph database

I database graph permettono di memorizzare le entità e le relazioni tra di esse. Le entità vengono anche viste come nodi con determinate proprietà. Si può pensare un nodo come un'istanza di un oggetto nell'applicazione. Le relazioni possono invece essere viste come archi con le loro proprietà. Come si può notare in Figura 5 gli archi dipendono dalla direzione infatti quelli entranti nel nodo con id=3 indicano l'appartenenza di un membro a tale gruppo mentre quelli uscenti indicano quali membri fanno parte del gruppo. I nodi sono organizzati attraverso le relazioni stesse e ciò permette di trovare dei pattern significativi tra di essi. L'organizzazione del grafo permette che i dati siano memorizzati una sola volta per poi essere interpretati in modi differenti in base alle loro relazioni. Normalmente quando salviamo una struttura a grafo su di un RDBMS, questo avviene per un solo tipo di relazione (es. chi è il padre di Aldo?). Aggiungere un'altra relazione significherebbe modificare e cambiare un grosso numero di schemi mentre ciò non avverrebbe nel caso di database graph. In questi database percorrere le join o le relazioni è molto veloce. Le relazioni tra i nodi non sono calcolate durante la query ma vengono memorizzate e quindi risulta molto più veloce percorrerle in questo caso piuttosto che doverle ricalcolare ogni volta. I nodi possono avere diversi tipi di relazioni tra di essi, permettendo di rappresentare sia relazioni tra le entità di dominio sia relazioni secondarie come categorie, percorsi, spatial indexing o liste ordinate per accessi ordinati. Dato che non c'è limite al numero e al tipo di relazioni che i nodi possono avere esse possono essere rappresentate nello stesso database graph.

Dato che la potenza di questi database deriva dalle relazioni e dalle loro proprietà è necessario un grosso lavoro di modellazione per rappresentare e comprendere le a pieno le relazioni nel dominio su cui andremo a lavorare. Ci sono diversi database graph come Neo4J, Infinite Graph, OrientDb o FlockD.

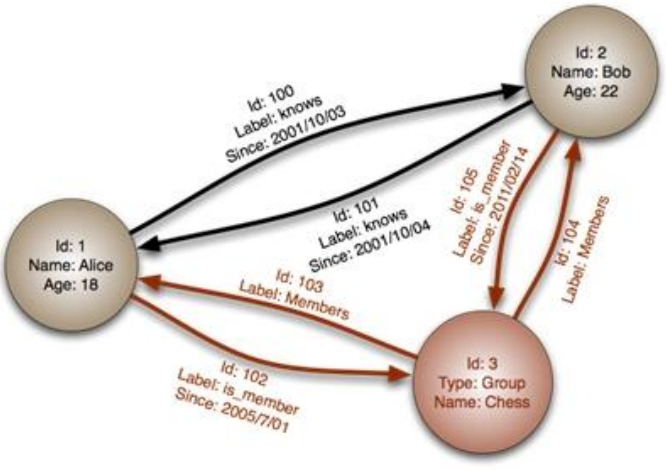

Figura 5 Esempio di gestione dati con modello Graph

1.2.7 Teorema CAP

In un sistema distribuito gestire la consistenza (tutti i nodi vedono gli stessi dati nello stesso momento), la disponibilità (la garanzia che ogni richiesta riceva una risposta positiva o negativa che sia) e la tolleranza alla partizione (capacità di gestire l'aggiunta o la rimozione un nodo nel sistema o la perdita di messaggi sulla rete) risulta molto importante.
All'inizio degli anni 2000 però Eric Brewer ha pubblicato, da prima come congettura e poi come teorema, il postulato che afferma che in ogni sistema distribuito possiamo scegliere solo due delle caratteristiche sopra elencate.
In Figura 6 vediamo come queste tre proprietà si intersechino e come solo nei casi in cui ci sia una doppia intersezione si possa avere un sistema reale rendendo impossibile avere tutte le caratteristiche ma costringendo a scegliere tra due di esse.

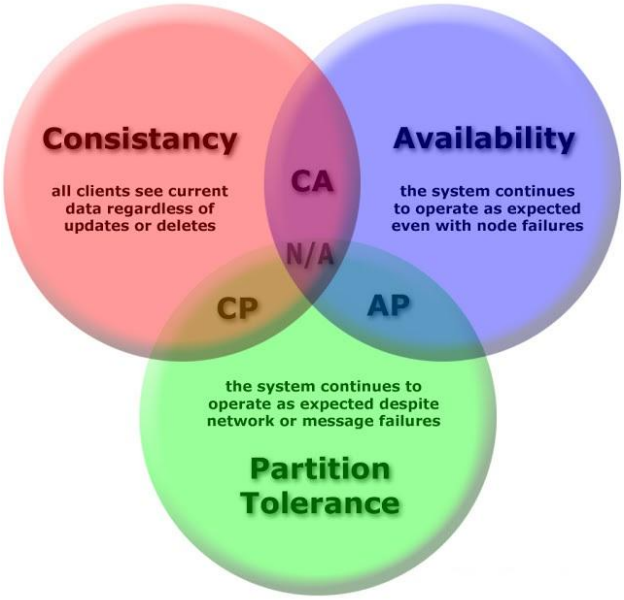

Figura 6 Intersezioni tra le tre proprietà del teorema CAP

Diversi sistemi NoSql cercano però di fornire delle opzioni con cui lo sviluppatore può modellare il database in base ai propri bisogni. Per esempio considerando Riak si hanno 3 variabili:

- r = numero di nodi che devono rispondere ad una richiesta di lettura prima che possa essere considerata riuscita
- w = numero di nodi che devono rispondere ad una richiesta di scrittura prima che sia considerata riuscita
- n = numero di nodi dove i dati sono replicati

In un cluster con 5 nodi possiamo rendere il sistema consistente settando r=5 e w=5 ma in questo modo il cluster risulta essere suscettibile alla tolleranza di partizione dato che basta

che non sia disponibile un solo nodo perché un'operazione di lettura non sia considerata riuscita. Possiamo invece focalizzarci sulla disponibilità del sistema settando r=1 e w=1 ma ora la consistenza è compromessa dato che alcuni nodi possono non avere la copia più recente dei dati. Il teorema CAP dice quindi che se si vuole una tolleranza alla partizione si deve scegliere un compromesso tra tra la disponibilità e la consistenza.
Una feature fornita di default negli RDBMS sono le transizioni e anche il modo di sviluppare da parte dei programmatori si è talmente abituato a considerare questa caratteristica di default che non contempla più il problema di cosa fare quando invece il database non fornisce tale proprietà. La maggior parte dei sistemi NoSql non fornisce le transazioni, è quindi compito dello sviluppatore pensare a come implementarle [5].

1.3 Apache-Hadoop

Apache Hadoop è framework open source per la memorizzazione e l'elaborazione su larga scala di data-set su cluster costituiti da commodity hardware. Hadoop è un progetto Apache costruito e utilizzato da una comunità mondiale di collaboratori e utenti sotto la licenza Apache 2.0 ed è stato creato nel 2005 da Doug Cutting e Mike Cafarella. Venne inizialmente sviluppato come supporto al progetto Nutch search engine, un motore di ricerca basato sulla piattaforma Lucene. E'stato progettato per scalare dal singolo server fino a migliaia di macchine ognuna delle quali fornisce dell'elaborazione e dello spazio di archiviazione locali. Invece di fare affidamento ad un hardware che fornisca alta disponibilità e affidabilità, Hadoop è stato creato per individuare e gestire le failures del livello di applicazione ottenendo quindi un servizio altamente disponibile sfruttando un cluster costituito da macchine che possono essere soggette a guasti [6].

1.3.1 Moduli di Hadoop

Tutti i moduli di Hadoop sono progettati con l'assunzione fondamentale che gli hardware failures (di singole macchine o di interi rack) possono essere comuni e che questi devono essere gestiti automaticamente lato software dal framework. Questa assunzione risulta fondamentale proprio per le caratteristiche del cluster utilizzato costituito da commodity hardware.

1.3.2 Hadoop common

Hadoop common è costituito da una collezione di utility e librerie che forniscono accesso al file system e supporto per gli altri moduli del sistema. Esso è considerato il core del framework perché fornisce servizi e processi essenziali come l'astrazione dal sistema operativo su cui risiede il cluster e il suo file system. Hadoop common contiene anche i file jar e gli script necessari per avviare il sistema Hadoop oltre a fornire il codice sorgente e la documentazione così come una sezione che include diversi progetti che derivano dal contributo della comunità di Hadoop.

1.3.3 Hadoop-Distributed-File-System (HDFS)

L'Hadoop Distributed File System (HDFS) è un file system distribuito progettato per girare sfruttando del commodity hardware. Sono presenti alcune analogie con i file system distribuiti esistenti, ma le differenze sono significative.
HDFS si può definire highly fault-tollerant cioè anche in caso di guasti o errori esso continua a fornire i suoi servizi attraverso la gestione di eventi imprevisti che rende trasparenti i guasti che possono avvenire sul cluster. Esso fornisce anche un efficiente accesso ai dati delle applicazioni e risulta ideale per quelle applicazioni che devono gestire grandi data set e rilassa alcuni requisiti POSIX per permettere un accesso a streaming ai dati del file system.
HDFS venne originariamente realizzato come infrastruttura per il progetto Apache Nucth web search engine, ma ora è un sottoprogetto di Hadoop.
Ogni nodo in un istanza Hadoop possiede un solo NameNode (contenitore dei metadati del cluster) e generalemente un DataNode (contenitore dei blocchi di dati), anche se un nodo non necessariamente deve possedere un DataNode per essere presente. Generalmente vengono memorizzati file di dimensioni elevate (solitamente nell'intervallo tra i Gb e Tb) su diverse macchine, spezzandoli in blocchi (di default da 128 Mb); in questo modo si riesce ad ottenere un alta affidabilità attraverso la replicazione dei dati su diversi host e quindi non richiedere l'archiviazione RAID sugli host. I NameNode monitorano attivamente il numero di repliche di ogni blocco, quando una replica di un blocco è persa a causa di una failure su un DataNode o di un disco il NameNode crea un'altra replica. Il NameNode mantiene anche l'albero dei NameSpace, il mapping tra i blocchi e i DataNode e mantiene l'immagine di tutto il NameSpace sulla RAM.

In Figura 7 si può notare l'architettura HDFS dove abbiamo diversi client che possono fare richieste di lettura o scrittura di blocchi di dati che risiedono nei DataNode e come sia presente un NameNode che immagazzina i metadati e interagisce con i DataNode.

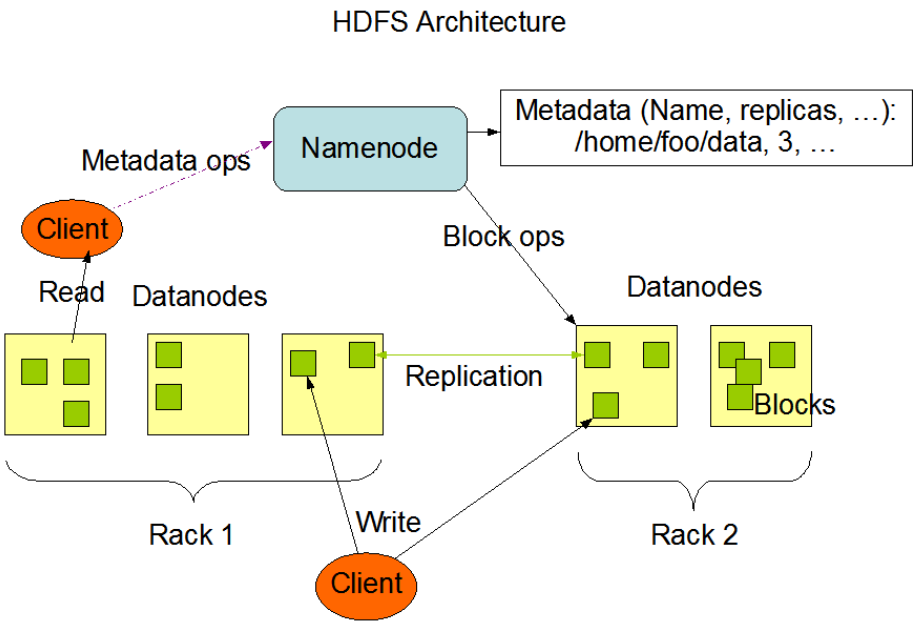

Figura 7 Architettura e gestione dei blocchi dati da parte di HDFS

I NameNode non inviano direttamente richieste ai DataNode ma inviano istruzioni come risposta ad un segnale di integrità inviato dallo stesso DataNode. Le istruzioni inviate possono essere le seguenti:

- Replicare il blocco su un altro nodo
- Rimuovere una replica locale del blocco
- Inviare un report sul blocco
- Spegnere il nodo

Il valore di replicazione di default delle repliche è 3, quindi i dati sono salvati su 3 nodi: il primo sul primo nodo selezionato mentre gli altri due su due nodi differenti di un altro rack distinto da quello del primo nodo. HDFS include anche un NameNode secondario che si connette regolarmente con il NameNode primario e ne crea una copia delle informazioni. Queste informazioni possono essere utilizzate per ripristinare il NameNode primario in caso di crash senza dover ricaricare tutte le azioni del file system. Dato che il NameNode è un singolo punto di gestione dei file può risultare un collo di bottiglia nel caso si debbano utilizzare un grosso numero di piccoli file. Esiste però l'HDFS federation che permette di

ovviare a questo problema utilizzando diversi NameSpace che utilizzano NameNode differenti. Un altro vantaggio nell'utilizzo di HDFS deriva dalla possibilità di sfruttare il principio di località:

Quando l'elaborazione coinvolge grandi insiemi di dati è più economico (più veloce) spostare il codice verso i dati piuttosto che i dati verso il codice

HDFS considera la posizione dei dati sui vari nodi e invia i job che devono essere eseguiti in base a questa conoscenza, quindi se sul nodo A avremo i dati (x,y,z) e sul nodo B i dati (a,b,c) il gestore dei job li schedulerà di modo che sul nodo A siano eseguite le operazioni di map o reduce sui dati (x,y,z) mentre sul nodo B le operazioni sui dati (a,b,c). In questo modo si limita la quantità di dati che devono essere inviati sulla rete migliorando le performance diminuendone il traffico.
Tutti i protocolli di comunicazione di HDFS fanno affidamento al protocollo TCP/IP. Il client stabilisce una connessione ad una porta TCP configurabile della macchina su cui risiede il NameNode e comunica attraverso un clientProtocol con esso, mentre i DataNode comunicano col NameNode attraverso uno specifico DataNode protocol.

1.3.4 Map Reduce

Sopra al file system si trova il Map Reduce engine, un framework software costituito da uno scheduler chiamato JobTraker che si occupa della gestione dei job che vengono inviati dal client. Il job è costituito da un serie di dati in input ognuno dei quali viene elaborato tramite un processo Map che genera delle coppie chiave/valore. Tali dati possono essere inviati sulla rete tramite shuffle di modo che possa poi essere eseguita l'operazione di reduce la quale elabora le coppie ricevute sulla base del loro valore di chiave. Il JobTraker quindi invia i job ad uno o più TaskTraker (nodi che accettano task cioè operazioni di map, reduce e shuffle) cercando di mantenere il principio di località. Sfruttando un file system che conosce la topologia del cluster come HDFS il JobTraker conosce quali nodi contengono i dati e quali altri sono vicini a tali dati. Se il lavoro non può essere ospitato sul nodo dove risiedono i dati viene data priorità ai nodi che risiedono sullo stesso rack, permettendo di ridurre il traffico di rete sulla backbone principale. Se un TaskTraker genera una failure o va in time-out quella parte di job viene rischedulata. Il TaskTraker comunque crea su ogni nodo una Java Virtual Machine separata per evitare di andare in errore nel caso che il job faccia crashare la JVM.

Ad un intervallo di alcuni minuti viene inviato un segnale di integrità da parte del TaskTraker al JobTraker per verificarne lo stato che può essere monitorato tramite browser web come è possibile vedere in Figura 8.

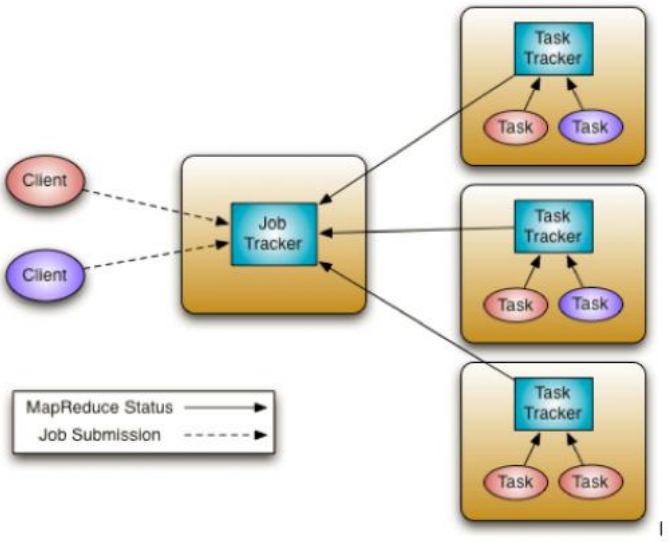

Figura 8 Gestione di un Job MapReduce su Hadoop

Se il JobTraker andava in crash dalle versioni di Hadoop precedenti alla 0.20 tutto il lavoro in corso andava perso mentre dalle versioni successive vengono aggiunti dei checkpoint al processo di modo che quando il JobTraker viene riavviato controlla se sono presenti tali checkpoint e nel caso riparte dal punto in cui si era fermato.

1.3.5 YARN

MapReduce è stato fortemente revisionato a partire da hadoop0.23 passando al cosiddetto MapReduce 2.0 o YARN. YARN (Yet Another Resource Negotiator) è un sottoprogetto di Hadoop all'Apache Software Founation introdotto in Hadoop 2.0 e che separa la gestione delle risorse dai componenti di elaborazione. Esso è nato per la necessità di fornire una più ampia gamma di modelli di interazione per i dati salvati in HDFS oltre a MapReduce e quindi l'architettura di Hadoop 2.0 basata su YARN fornisce una piattaforma di elaborazione che non è vincolata al solo MapReduce.

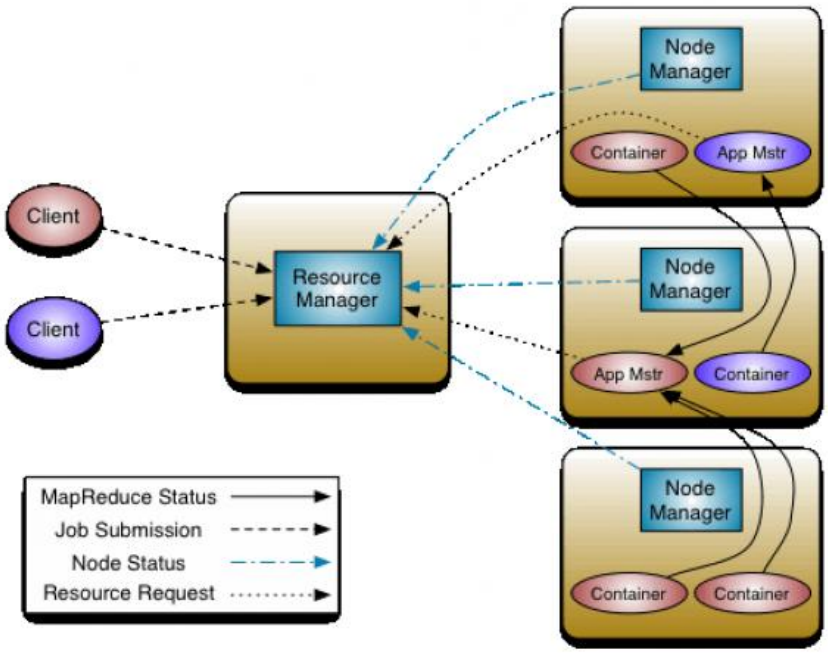

Figura 9 Gestione di MapReduce con YARN

L'intento originale di YARN era per l'appunto dividere le due maggiori responsabilità del JobTraker/TaskTracker in entità separate come è anche possibile vedere in Figura 9:

- Un ResourceManager globale
- Un ApplicationMaster per ogni applicazione
- Un NodeManager per ogni nodo slave
- Un container per ogni applicazione che gira su un NodeManager

Il ResourceManager col NodeManager forma il nuovo sistema per la gestione delle applicazioni in maniera distribuita. Il ResourceManager è il componente che comanda e decide come le risorse devono essere utilizzate nel sistema. L'ApplicationMaster è un entità specifica del framework che negozia le risorse dal ResourceManager e collabora con il NodeManager per eseguire e monitorare i task. Il ResourceManager ha uno scheduler che è responsabile dell'allocazione delle risorse alle varie applicazioni che risultano in esecuzione sul cluster, considerando sempre i vincoli quali ad esempio le capacità delle code e le limitazioni dei vari utenti. Lo scheduler lavora basandosi sulle risorse richieste da ogni applicazione. Ogni ApplicationMaster ha la responsabilità di negoziare dei container di risorse appropriati con lo scheduler, tracciando il loro stato e monitorandone i progressi. Dal punto di vista del sistema l'ApplicationMaster gira come un normale container. Il NodeManager è un'entità su ogni nodo slave che è responsabile per l'avvio degli applications container, del monitoraggio dell'utilizzo delle loro risorse (cpu,memoria, disco,

rete) e di riportare tutto al ResourceManager. In Figura 9 si possono osservare vari moduli di Hadoop e si può notare come alla base di tutto lo stack si trovino i moduli HDFS e YARN su cui poi si basano i livelli successivi come MapReduce.

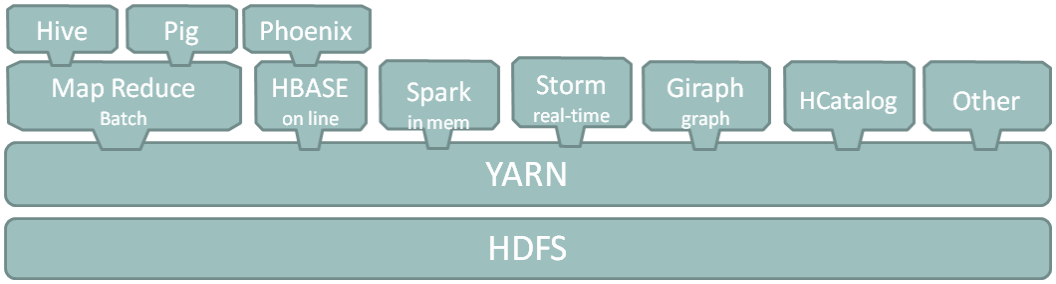

Figura 10 Stack dei moduli Hadoop

2 Apache Spark

2.1 Introduzione a Spark

Apache Spark è una piattaforma per l'elaborazione di dati su cluster progettata per essere particolarmente veloce e general-purpose. Inizialmente venne sviluppato nel 2009 al Berkley's AMPLab per poi diventare nel 2010 un progetto Apache open-source. Dal punto di vista della velocità Spark estende il popolare modello MapReduce per supportare in maniera efficiente più tipi di elaborazione incluse le interactive queries e lo stream processing. La velocità è fondamentale per processare grandi dataset e per esplorare i dati in maniera interattiva piuttosto che aspettare minuti o ore. Una delle caratteristiche più importanti che Spark offre per aumentare la velocità è la possibilità di eseguire l'elaborazione dei dati in memoria centrale, anche se il sistema resta comunque più efficiente dei tradizionali sistemi MapReduce anche per applicazioni complesse che utilizzano il disco. Dal punto di vista general-purpose, Spark è progettato per coprire una vasta gamma di applicazioni che prima avrebbero richiesto diversi sistemi distribuiti separati. La generalità del motore di Spark risulta importante proprio perché gran parte degli utenti combinano assieme diverse tipologie di data processing nelle loro applicazioni [7]. Supportando queste applicazioni sullo stesso motore, Spark rende semplice e poco costosa la combinazione di diverse tipologie di processi che sono spesso necessari per l'analisi dei dati, in più riduce la complessità della manutenzione del sistema non dovendo considerare diversi strumenti separati tra loro.

Spark è progettato per essere facilmente utilizzabile offrendo numerose api in Scala, Java, Python e SQL oltre a diverse librerie. Si integra molto bene con altri strumenti di lavoro sui BigData, infatti Spark può girare su cluster Hadoop e accedere ad ogni sua sorgente dati.

2.2 I Componenti

Il progetto Spark contiene diversi componenti fortemente integrati. Nel core di Spark troviamo un motore di elaborazione che è responsabile dello scheduling, della distribuzione e del monitoraggio di applicazioni costituite da diversi task su diverse macchine. Essendo l'engine di Spark sia veloce che general-purpose permette di utilizzare anche diversi componenti di alto livello (vedi Figura 11) specializzati per diversi carichi di lavoro come SQL e Machine Learning. Tutti questi componenti sono fortemente integrati e questo porta a diversi vantaggi: tutte le librerie e i componenti dei livelli superiori dello stack usufruiscono delle ottimizzazioni apportate ai livelli inferiori. Per esempio quando viene aggiunta un ottimizzazione all'engine di Spark automaticamente le librerie SQL e di machine learning migliorano le prestazioni allo stesso modo. In secondo luogo il costo associato all'avvio dello stack è minimizzato perché invece di far partire numerosi sistemi indipendenti è sufficiente avviarne solamente uno.

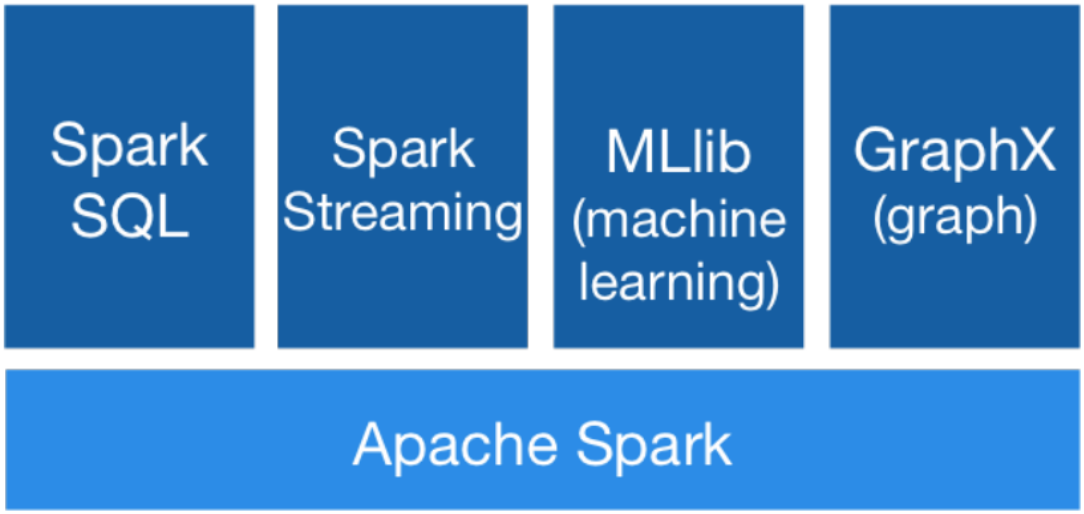

Figura 11 Stack di Spark

2.2.1 Spark core

Lo Spark Core contiene le funzionalità di base di Spark compresi i componenti per lo scheduling dei task, la gestione della memoria e il fault recovery, l'interazione con i sistemi di storage e altro. Nello Spark Core risiedono anche le API che definiscono i Resilient Distributed Dataset (RDD) che sono la principale astrazione della programmazione di Spark. Gli RDD rappresentano una collezione di oggetti distribuiti su diversi nodi che possono

essere manipolate in parallelo e le API presenti in questo componente permettono di gestire tali collezioni.

2.2.2 Spark SQL

Spark SQL è un componente di Spark che permette di lavorare con dati strutturati ed è stato aggiunto a Spark a partire dalla versione 1.0. Prima era presente Shark, un vecchio progetto per avere SQL su Spark che modificava Hive per permettere di farlo girare su Spark. Permette di interrogare i dati tramite SQL e supporta diverse sorgenti dati come ad esempio tabelle Hive, Parquet e JSON. Oltre a fornire un'interfaccia SQL a Spark, questo modulo permette agli sviluppatori miscelare query SQL assieme alla gestione dei dati tramite RDD sfruttando linguaggi di programmazione come Java, Scala e Python tutto nella stessa applicazione.

2.2.3 Spark Streaming

Spark Streaming è un componente di Spark che permette di elaborare stream di dati.
Esempi di dati in stream possono essere i file di log generati dai server web o code di messaggi che contengono aggiornamenti di stato aggiornati dagli utenti di un web service.
Spark Streaming fornisce un'insieme di API per manipolare gli stream di dati molto simile alle API presenti nello Spark Core per la gestione degli RDD rendendo più semplice agli sviluppatori la comprensione del progetto e passare da una tipologia di applicazione ad un'altra. Lo Spark Streaming è stato progettato per avere lo stesso grado di foult tollerance, throughput e scalabilità dello Spark Core.

2.2.4 MLib

Installando Spark è già presente una libreria che contiene le funzionalità più comuni di machine learning (ML) chiamata MLib. Essa fornisce diverse tipologie di algoritmi di machine learning nell'ambito della classificazione, della regressione e del clustering; inoltre supporta diverse funzionalità come la valutazione dei modelli e l'import dei dati.
Fornisce anche diverse primitive di basso livello per il Machine Learning come ad esempio l'ottimizzazione tramite gradiente. Tutte queste funzionalità sono comunque progettate per scalare su di un cluster.

2.2.5 GraphX

GraphX è una libreria per la gestione dei grafici (ad esempio grafici di amicizie in un social network) ed eseguire computazioni sui grafici in parallelo. Come lo Spark Streaming e lo Spark SQL GraphX estende le API sugli RDD permettendo di creare un grafo con tutte le proprietà che si desidera definire sui vertici e sugli archi. GraphX fornisce anche diversi operatori per manipolare i grafi e una libreria di algoritmi su di essi(ES PageRank e triangle counting)

2.2.6 Cluster Managers

Alla sua base Spark è progettato per scalare in maniera efficace da uno fino a migliaia di nodi di computazione. Per raggiungere questo obbiettivo massimizzando la flessibilità Spark può essere eseguito su una vasta gamma di cluster managers quali Hadoop YARN, Apache Mesos e anche un semplice cluster manager incluso in Spark chiamato Standalone Scheduler. Se si vuole installare Spark su di una o più macchine vuote lo Standalone Scheduler fornisce un modo semplice per iniziare a conoscere Spark; in ogni caso se sono già presenti Hadoop Yarn o Mesos Spark supporta questi cluster manager permettendo alle applicazioni di essere eseguite su di essi. [8]

2.3 Architettura

Spark può lavorare sia su una singola macchina in local mode (soprattutto per il debug e per fini istruttivi) sia su di un cluster. Uno dei benefici di scrivere applicazioni in Spark è l'abilità di scalare la computazione in maniera semplice aggiungendo dei nodi e far girare tali applicazioni su di essi (cluster mode). Allo stesso modo con cui vengono utilizzate le API in local mode è possibile riutilizzarle allo stesso modo nel caso di utilizzo di un cluster, rendendo possibile agli utenti di eseguire in maniera molto semplice e veloce dei prototipi su piccoli dataset per poi eseguire la stessa applicazione su un cluster grande a piacere senza dover modificare il codice.

2.3.1 Architettura a Runtime

Nell'architettura runtime di Spark entrano in gioco diversi elementi che andremo a trattare in questa sezione:

- Il driver: processo di coordinazione centrale (uno per ogni applicazione Spark eseguita)
- Gli executor: processi slave sui quali vengono eseguite le operazioni richieste dall'applicazione
- I task: l'unità di esecuzione fisica più piccola che esegue il fetch dei dati, la loro elaborazione e la scrittura del risultato
- Le trasformation: operazioni che restituiscono un nuovo dataset a partire dall'elaborazione del dataset di partenza
- Le action: operazioni che ritornano il loro risultato al driver centrale dopo aver eseguito una computazione sul dataset
- Gli stage: collezione di task indipendenti che eseguono diverse transformation senza necessità di scambio dati in rete
- I job: insieme di stage per la risposta ad un action invocata dall'applicazione

In modalità distribuita Spark utilizza un'architettura master/slave con un coordinatore centrale e diversi workers distribuiti. Come si può vedere in Figura 12 il driver funge da coordinatore centrale e comunica con i numerosi workers che costituiscono gli executors. Il driver viene eseguito su un suo personale processo Java e ogni executor è un processo separato. L'insieme costituito dal driver e i suoi executor costituisce un'applicazione Spark.

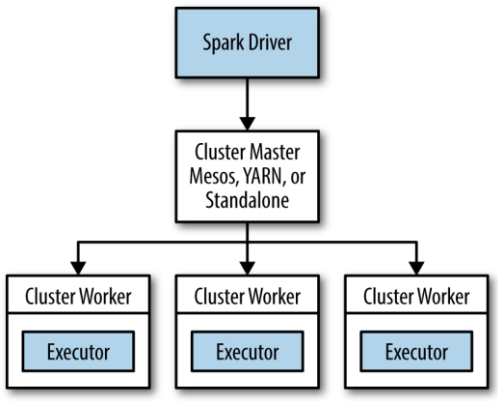

Figura 12 I componenti di un applicazione Spark distribuita

Un'applicazione Spark viene lanciata su un insieme di macchine utilizzando un servizio esterno chiamato cluster manager. All'interno di Spark troviamo il suo cluster manager (lo Standalone cluster manager) anche se generalmente viene utilizzato assieme ad Hadoop YARN e Apache Mesos.

2.3.2 Driver

Il driver è il coordinatore centrale dell'applicazione ed è il processo in cui viene eseguito l'avvio dell'applicazione (il metodo main()). Esso è quindi il processo su cui viene eseguito il codice dell'utente che crea lo SparkContext, gli RDD ed esegue le transformation e le action. In Spark possiamo trovare anche una shell (Spark-shell) cioè una console simile a quelle utilizzate in Scala o R che permette di eseguire comandi e analisi ad hoc sui dati e a differenza delle altre shell non lavora sulla singola macchina ma su tutto il cluster; per questo quando essa viene avviata si crea un programma driver. Quando il driver viene chiuso allora anche tutta l'applicazione termina. Una volta che il driver entra in esecuzione esso fornisce due servizi:

- Converte il programma dell'utente in task

  Il driver Spark è responsabile della conversione del programma in unità di esecuzione fisica (i tasks). Ad un livello più alto tutti i programmi Spark utilizzano la stessa struttura: creano uno o più RDD da un qualche tipo di input, ricavano nuovi RDD dai precedenti utilizzando delle transformation ed eseguono delle azioni per ottenere un risultato e salvare i dati. Un programma Spark crea implicitamente un grafo aciclico diretto (DAG) di operazioni. Quando il driver viene eseguito esso converte il grafo logico in un piano di esecuzione fisico. Spark opera anche alcune ottimizzazioni come ad esempio il "pipeling" per mappare le transformation assieme e unirle, in più converte il grafo in un insieme di stages.

  Ogni stage è costituito da diversi task i quali sono preparati e inviati al cluster. Come detto i task sono la più piccola unità di lavoro in Spark ne segue che una classica applicazione può lanciare da alcune centianaia fino a migliaia di task.

- Schedula i task sugli executor

  Dato un piano di esecuzione fisico, il driver deve coordinare lo scheduling di ogni task su di un executor. Quando gli executor vengono avviati essi si

registrano al driver di modo che esso abbia una completa visione di tutti gli executor dell'applicazione in ogni momento. Ogni executor può essere visto come un processo in grado di eseguire task e memorizzare dati degli RDD.
Il driver considera l'insieme di executor registrati e cerca di schedulare ogni task nella giusta locazione considerando di sfruttare il più possibile il principio di località dei dati. Quando i task vengono eseguiti essi possono avere dei side-effect come ad esempio memorizzare dei dati che avevano in cache, in questo caso il driver prende nota della posizione in cui questi dati sono salvati per avere maggiori informazioni su come schedulare i successivi task che accedono a questi dati. Viene anche fornita un'interfaccia web su cui possono essere consultate alcune informazioni riguardanti l'esecuzione delle applicazioni.

2.3.3 Executor

Gli Spark executor sono processi sui quali vengono eseguiti i task di un job. Essi vengono avviati all'inizio dell'applicazione e generalmente restano in vita fino a che l'applicazione non viene terminata, in ogni caso per garantire una maggiore fault tollerance anche se un executor dovesse fallire l'applicazione continuerebbe a girare. Gli executor hanno due ruoli:

- Eseguono i task che gli sono stati assegnati e restituiscono i risultati al driver.
- Forniscono uno spazio di memorizzazione degli RDD che sono stati inseriti in cache dal programma dell'utente attraverso un servizio chiamato Block Manager che si trova all'interno di ogni executor. Dato che gli RDD vengono salvati direttamente dentro agli executor i task possono lavorare direttamente sui dati in cache.

Gli executor 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.
E'possibile specificare quante risorse assegnare agli executor tramite degli appositi parametri. Tali parametri possono essere specificati sia tramite console all'invocazione dello "spark-submit" per lanciare l'applicazione, sia da codice quando vengono specificati i parametri dello SparkContext attraverso lo SparkConf. Il numero di core utilizzati da ogni Executor viene specificato attraverso il parametro *executor.core*; questa proprietà controlla il numero di task concorrenti che un executor può eseguire. Ad esempio specificare

executor.core = 5 significa che ogni executor può eseguire al massimo 5 task nello stesso momento.

Gli executor hanno una certa quantità di memoria assegnata, che gli permette di memorizzare i dataset in memoria se richiesto dall'applicazione utente (tramite l'istruzione cache su un RDD).La memoria assegnata può essere configurata tramite il parametro *executor.memory*. [9] Dentro ad ogni executor la memoria è utilizzata per diversi scopi:

- RDD Storage: quando viene chiamata la funzione persist o cache() di un RDD, le sue partizioni vengono memorizzate in buffer di memoria. Spark limita però la quantità di memoria utilizzata per questa operazione ad una determinata frazione della memoria totale dello heap della JVM (*spark.storage.memoryFraction*). Se tale limite viene superato le vecchie partizioni verranno cancellate dalla memoria.
- Shuffle e Aggregation Buffer: quando vengono eseguite delle operazioni di shuffle, Spark crea dei buffer intermedi per memorizzare i dati di output. Questi buffer sono anche utilizzati per memorizzare i risultati intermedi dell'aggregazione. La memoria per tali buffer può essere gestita attraverso il parametro *spark.shuffle.memoryFraction*.
- User Code: Spark esegue qualsiasi tipo di codice utente quindi possono esserci funzioni che richiedono una grande quantità di memoria ad esempio se vengono allocati array molto grandi. Il codice dell'utente ha accesso a tutto lo spazio di memoria heap della JVM lasciato dalle gestione degli RDD e dello Shuffle.

Di default Spark lascia il 60% dello spazio per la memorizzazione degli RDD, il 20% per i buffer dello shuffle e il restante 20% per il codice utente [8].

2.3.4 Cluster manager

Spark dipende dalla presenza di un cluster manager che gestisca le risorse, ad esempio è lui che si occupa di avviare gli executor e il driver. Il cluster manager viene considerato da Spark come un componente collegabile e questo gli permette di utilizzare non solo il suo cluster manager (standalone cluster manager) ma anche di sfruttarne altri esterni come YARN e Mesos. [8]

2.4 Hadoop and Spark

Hadoop è una tecnologia per il processamento dei Big Data molto utilizzata per l'elaborazione di grandi data-set. Essa sfrutta MapReduce che è un ottima soluzione in determinati casi ma quando sono necessari computazioni particolarmente pesanti e la necessità di sfruttare determinati algoritmi, non risulta particolarmente efficiente. Ad ogni passo nell'elaborazione dei dati c'è una fase di "map" e una di "reduce" e quindi risulta necessario convertire ogni soluzione al pattern MapReduce per ottenere un risultato. I dati del job in output devono essere memorizzati nel file system distribuito prima che il passo successivo possa iniziare, questo approccio risulta però essere lento a causa della replicazione e della necessità di scrivere su disco. In più le soluzioni Hadoop tipicamente sfruttano cluster che sono difficili da configurare e gestire e richiede un integrazione tra diversi strumenti in base ai differenti casi d'uso dei Big Data (come ad esempio Mahout per il Machine Learning e Storm per l'elaborazione di dati in stream). Per realizzare applicazioni complesse risulta necessario unire tra loro una serie di job e eseguirli in sequenza. Ogni job risulta avere grosse latenze e ognuno non può iniziare prima che il precedente non sia concluso.
Spark permette ai programmatori di sviluppare applicazioni complesse utilizzando il directed acyclic graph (DAG) pattern e supporta anche la condivisione di dati in memoria attraverso i DAG cosicchè diversi job possano lavorare con gli stessi dati. Spark lavora sopra all'infrastruttura fornita da HDFS per fornire funzionalità aggiuntive e permette di sviluppare ed eseguire applicazioni Spark su di un cluster Hadoop v1, Hadoop v2 YARN. [11]

2.5 Modello di programmazione

Spark fornisce principalmente due astrazioni per gestire la programmazione parallela: i Resilient Distributed Datasets (RDDs) e operazioni parallele su tali dataset (richiamate attraverso il passaggio di funzioni da applicare ai dataset). In più Spark supporta due tipi di variabili condivise che possono essere utilizzate nelle funzioni che vengono eseguite sul cluster.

2.5.1 Resilient distributed dataset

Un Resilient Distributed Dataset (RDD) è una collezione read-only di oggetti partizionati attraverso un insieme di macchine (si veda Figura 13) e le cui parti possono essere ricostruite nel caso vadano perse. Non è necessario che gli elementi di un RDD siano presenti nello spazio di archiviazione fisico; infatti un handler dell'RDD contiene sufficienti informazioni per elaborare un RDD a partire dai metadati che lo descrivono. Questo significa che un RDD può sempre essere ricostruito se un nodo fallisce.

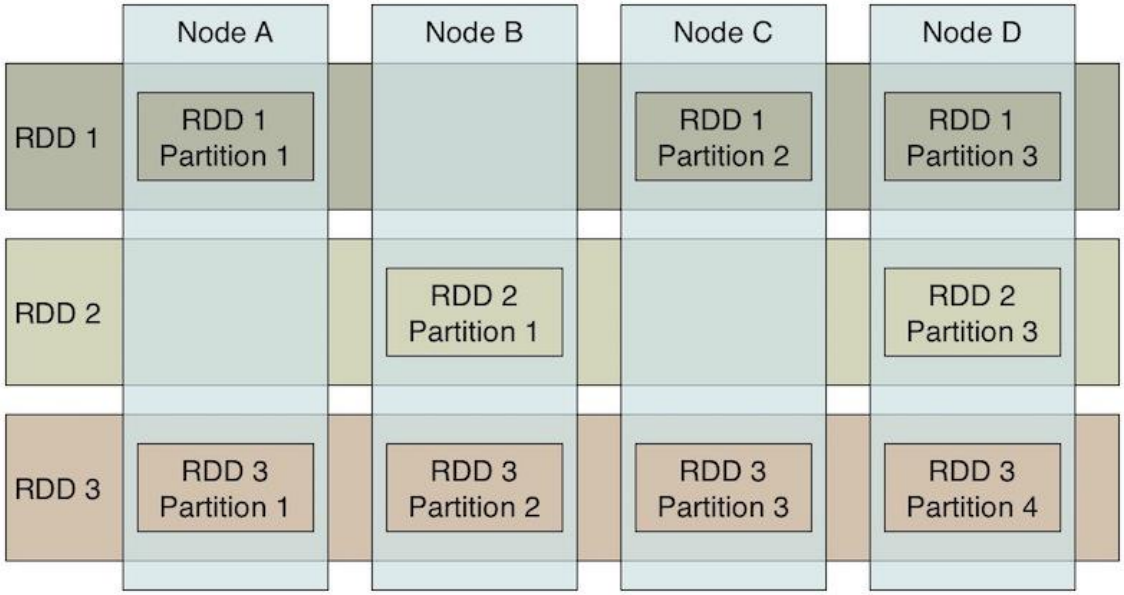

Figura 13 Esempio di partizionamento di 3 RDD su di un cluster

In Spark ogni RDD è rappresentato da un oggetto Scala e viene permesso agli sviluppatori di creare degli RDD in 4 modi:

- A partire da un file presente in un file system condiviso come ad esempio HDFS
- Attraverso l'operazione "parallelize" effettuata su di una collection di Scala (ad esempio un array) sul driver dividendola quindi in una serie di parti che vengono inviate a diversi nodi
- Eseguendo una trasformation su un RDD esistente. Un dataset con elementi di tipo A può essere trasformato in un dataset di elementi di tipo B utilizzando un operazione chiamata flatMap che processa ogni elemento del primo dataset attraverso una funzione definita dall'utente per trasformarlo in un altro tipo oppure semplicemente attraverso un operazione di filtering (vengono prelevati solo gli elementi per i quali vale una determinata proprietà).

- Modificando la persistenza di un RDD esistente. Di default gli RDD sono cosiddetti lazy, cioè le partizioni di un dataset sono materializzate su richiesta solo quando sono utilizzate in operazioni parallele (ad esempio quando si passa un blocco di un file attraverso una funzione di map) e vengono eliminate dalla memoria subito dopo l'utilizzo.

  In ogni caso un utente può modificare la persistenza di un RDD attraverso due azioni:

  - L'azione "cache" lascia il dataset lazy ma suggerisce di lasciarlo in memoria non appena avviene la prima computazione perché dovrà essere riutilizzato.
  - L'azione "save" forza l'elaborazione del dataset e lo scrive su di un file system distribuito come ad esempio HDFS.

In ogni caso si deve considerare che l'azione di cache è solo un suggerimento: se non c'è sufficiente spazio in memoria per effettuare il cache di tutte le partizioni del dataset, Spark le dovrà comunque rielaborare quando saranno riutilizzate.
Questo comportamento è dovuto alla volontà di far continuare a lavorare Spark (anche se con performance ridotte) anche nel caso in cui dei nodi vadano in fail o se un dataset risulti troppo grande. L'obiettivo è quello di garantire agli utenti un compromesso tra il costo per la memorizzazione degli RDD, la velocità per accedervi, la probabilità di perderne una parte e il costo per rielaborali [12].

2.5.2 Operazioni sugli RDD

Gli RDD supportano due tipi di operazioni : le trasformazioni e le azioni. Le trasformazioni (transformation) sono operazioni sugli RDD che ritornano un nuovo RDD (ad esempio operazioni come map e filter) mentre le azioni (action) sono operazioni che ritornano un risultato al driver o lo salvano su disco (ad esempio operazioni come count o first). In particolare in Figura 14 possiamo vedere le due categorie in cui si dividono le trasformation:

- Narrow Transformation: processi in cui la logica di elaborazione dipende solo dai dati che già risiedono nella partizione e non è necessario lo shuffeling dei dati. ES filter(), sample(), map(), flatMap() ecc ecc
- Wide Transformation: processi in cui la logica di elaborazione dipende dai dati che risiedono su diverse partizioni e quindi risulta necessario effettuare uno shuffeling dei dati per poterli riportare assieme in un unico punto. Es. groupByKey(), reduceByKey() [13]

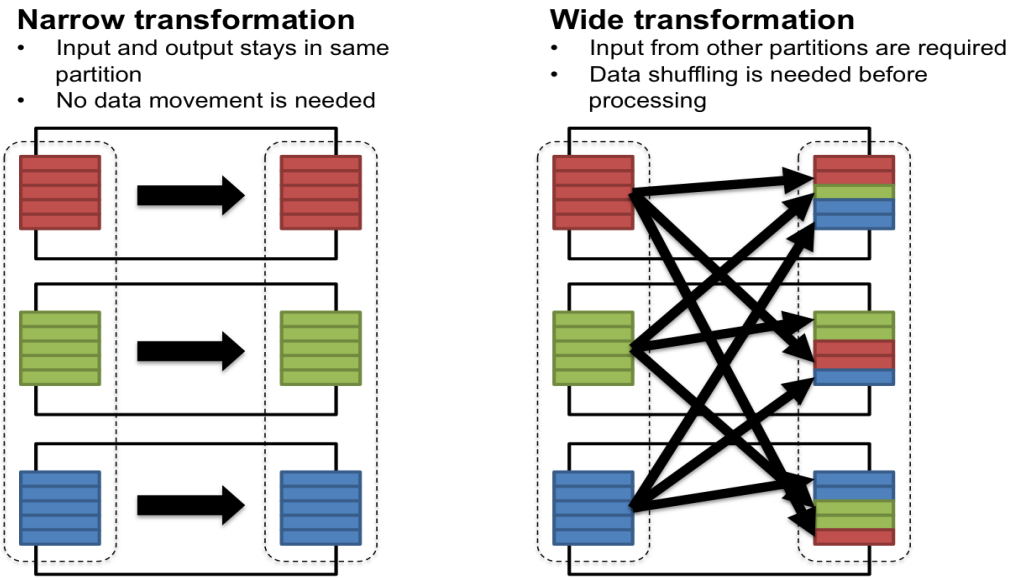

Figura 14 Differenze tra Narrow trasformation e Wide trasformation

In alcuni casi le transformation vengono riordinate per limitare lo shuffeling dei dati. Nell'esempio di Figura 15 abbiamo due grandi RDD in join tra loro seguiti da un filtro.

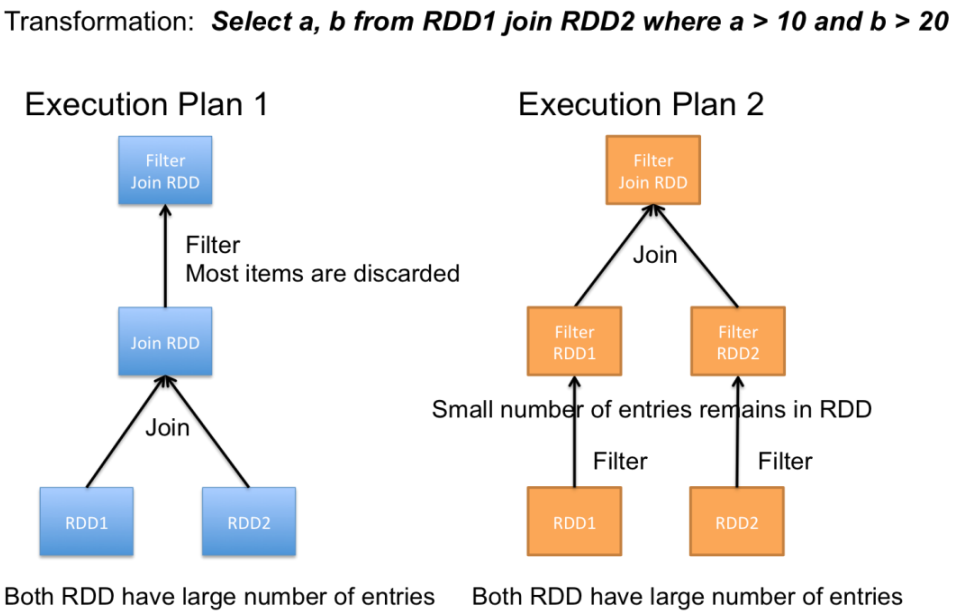

Figura 15 Esempio di ottimizzazione di query

Il piano1 è un'implementazione naive che segue questo ordine: per prima cosa fa il join tra i due RDD e poi applica il filtro al risultato della join. Questo causa un pesante shuffeling dei

dati perché i due RDD sono molto grandi anche se il risultato dopo il filtraggio risulta piccolo. Il piano due invece offre un modo più intelligente di usare la tecnica del push-down per cui viene applicato il filtro prima di effettuare la join. Dato che il filtro riduce il numero di elementi di ogni RDD in maniera significativa il processo di join risulterà molto più conveniente. [12] Risulta possibile distinguere il tipo di un operazione che viene eseguita su di un RDD semplicemente sul tipo di ritorno di tale funzione (RDD nel caso delle trasformazioni e qualsiasi altro tipo nel caso di azioni). Risulta importante saper distinguere quale tipo di operazione si sta eseguendo per comprendere appieno il comportamento del nostro programma, infatti Spark tratta in maniera differente le une dalle altre.

2.5.3 Lazy Evaluation

Come detto in precedenza gli RDD sono considerati oggetti la cui valutazione (delle trasformazioni) viene effettuata in maniera "lazy" cioè Spark non inizierà la loro esecuzione fino a che non verrà richiesta un'azione. La lazy evalutation fa si che quando viene chiamata una trasformazione su di un RDD (ad esempio chiamando map()) l'operazione non venga eseguita immediatamente mentre invece internamente Spark registra i metadati per indicare che questa operazione è stata richiesta. Piuttosto che considerare un RDD come un contenitore di dati è più corretto pensarlo come un insieme di istruzioni su come elaborare i dati che vengono definiti dalla varie trasformazioni. Anche il caricamento dei dati su di un RDD è valutato in maniera lazy così come lo sono le trasformazioni, quindi quando viene richiesta l'operazione per caricare tali dati su di un RDD essi non vengono realmente caricati finche non risulta strettamente necessario.
Spark sfrutta la lazy evalutation per ridurre il numero di volte che deve scorrere tutti i dati raggruppando le operazioni assieme ottimizzandone la gestione. [13]

2.6 Join in Spark

In Spark i dataset analizzati come detto sono spesso molto grandi e le tabelle stesse risultano essere di dimensioni sostenute. Per velocizzare l'accesso a tali dati Spark utilizza gli RDD che vengono caricati in memoria centrale, ma essendo spesso di dimensioni elevate essi risultano partizionati su diversi nodi sfruttando la memoria di diverse macchine. Qui nasce quindi uno dei principali colli di bottiglia che caratterizzano non solo Spark ma la programmazione su cluster in generale: lo shuffeling dei dati, cioè l'invio dei dati sulla rete

del cluster per fare in modo di ottenere i risultati di query che interessano informazioni presenti su diverse macchine. Nella valutazione delle performance compare quindi un nuovo parametro che indica la quantità di dati che devono essere inviati sulla rete.
Tra le varie operazioni più pesanti dal punto di vista prestazionale che richiedono lo shuffle dei dati troviamo i join che devono essere trattati in maniera particolare proprio per la natura partizionante dei cluster. In un join tra due tabelle risulta necessario scorrere tutti i record per trovare i match tra le chiavi di join dei record, ma in un cluster una singola macchina possiede solo una parte di queste tabelle. Diventa quindi necessario prima di eseguire l'operazione di join vera e propria inviare i dati delle tabelle sulle macchine di modo che possano elaborare una sotto parte della join in maniera completa e corretta.
L'esecuzione delle join in Spark viene gestita in due modi:

2.6.1 Shuffled Hash Join

In questo caso i record delle due tabelle vengono inviate ad una determinato worker basandosi su di una funzione di hash applicata all'attributo di join.

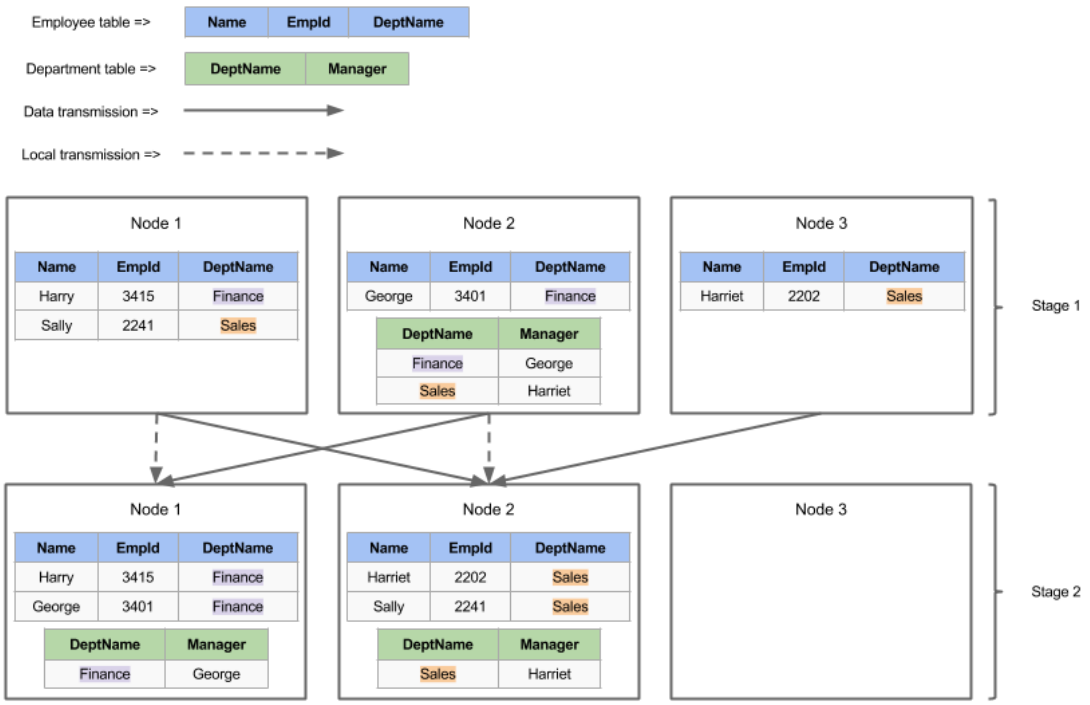

Figura 16 Esempio di Shuffle Join. Si può notare come entrambe le tabelle siano state partizionate sui nodi sulla base del campo di join

Ogni executor carica in memoria i dati presenti sul suo disco locale ed elabora record per record attraverso una funzione di hash che indichi l'executor a cui tale record è destinato.

Tale funzione di hash permette di dividere i record di entrambe le tabelle su tutti gli executor avendo la sicurezza che i record che devono essere uniti in join poichè possiedono lo stesso valore sull'attributo di join siano inviati allo stesso executor che li possa processare localmente. Come si può notare nell'esempio in Figura 16 i dati vengono suddivisi in base al valore dell'attributo di join "DeptName" di modo che sul primo nodo troveremo i record che hanno come valore per questo attributo "Finance" mentre sul secondo nodo i record con valore "Sales" (nel nodo 3 non è presente nessun record perché in questo semplice esempio i valori distinti dell'attributo di join sono solo 2)

2.6.2 Broadcast Hash Join

Il caso in cui viene utilizzata un Broadcast Hash Join invece avviene quando una delle due tabelle risulta essere piccola e quindi può essere facilmente mantenuta in memoria da ogni executor. In questo caso infatti la tabella che risulta essere più piccola viene prima caricata sul driver centrale (ogni executor invia la parte di tabella caricata dal disco locale) il quale poi si incarica di inviarla in broadcast a tutti gli executor.

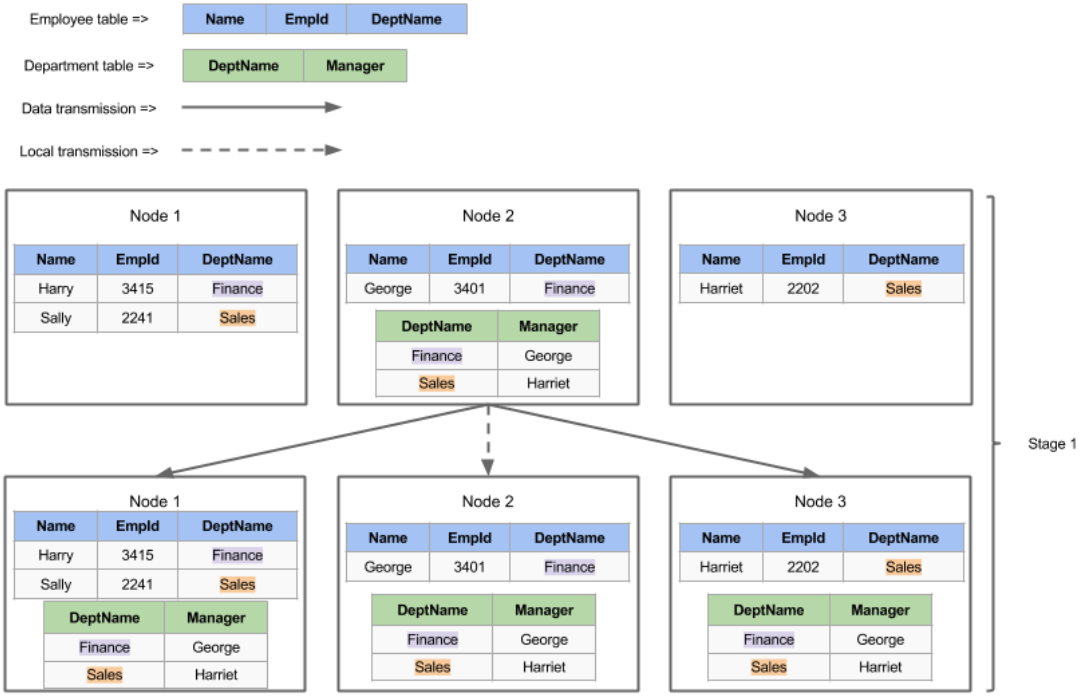

Figura 17 Esempio di Broadcast Join. Si può notare come la prima tabella non sia stata partizionata, così come la seconda tabella che è stata inviata in broadcast a tutti i nodi

In questo modo ogni executor può semplicemente caricare in memoria centrale i dati presenti sul disco locale relativi alla tabella di grandi dimensioni ed eseguire il join per ognuno dei suoi record con la certezza di trovare il match (se presente) sulla seconda tabella più piccola

dato che risulta essere presente per intero sulla sua memoria. In questo modo il traffico sulla rete risulta essere ridotto non avendo necessità di inviare la tabella più grande sulla rete.
Nell'esempio di Figura 17 si può osservare proprio come ogni nodo carichi in locale la sua partizione della tabella grande (quella in blu) mentre la tabella piccola (quella in verde) che si trova sul nodo2 viene inviata in broadcast a tutti gli altri nodi. Per semplificare in questo esempio si vede l'invio della tabella piccola direttamente dal nodo2, in realtà si consideri che prima la tabella viene inviata al driver il quale poi si occupa del suo invio in broadcast.

2.6.3 Valutazione

Risulta quindi necessario comprendere quale tipo di join utilizzare per migliorare le prestazioni. Nel caso in cui le due tabelle abbiano entrambe dimensioni considerevoli (e quindi nessuna delle due possa essere caricata in memoria centrale sul singolo executor) risulta obbligatorio utilizzare un Shuffled Hash Join che premette di suddividere entrambe le tabelle. Nel caso in cui invece una delle due tabelle risulti sufficientemente piccola per essere caricata in memoria occorre valutare attentamente la situazione: se la tabella inviata in broadcast risulta essere effettivamente piccola le prestazioni migliorano rispetto ad un caso di Shuffled Hash Join grazie alla quantità di dati inviati sulla rete notevolmente inferiore.
Nel caso in cui invece la differenza di dati inviati sulla rete non sia sufficientemente alta tra i due casi il Broadcast Hash Join risulta essere addirittura più lento rispetto allo Shuffled Hash Join, questo perché nonostante il traffico in rete possa risultare minore del caso broadcast le comunicazioni risultano essere più pesanti avendo un collo di bottiglia iniziale che riguarda l'invio di tutti i dati della tabella piccola sul driver centrale.
In ogni caso per definire quando una tabella può risultare "piccola" Spark si basa sul valore del parametro *spark.sql.autoBroadcastJoinThreshold* che di default è settato a 10 Megabyte ma può essere modificato dall'utente. Spark confronta quindi la dimensione della tabella più piccola in join con tale parametro e se risulta essere più piccola viene utilizzata un Broadcast Hash Join altrimenti viene utilizzato uno Shuffled Hash Join.

3 Gestione delle query

3.1 Struttura di una query GPSJ

Utilizzeremo come modello di query le cosiddette GPSJ (Generalized Projection-Selection-Join) perché rappresentano la classe di query più utilizzate nell'ambito OLAP (On-Line Analytical Processing). Si considera quindi un cubo OLAP cioè una struttura di memorizzazione dei dati su diverse dimensioni che può essere modellato a partire da due distinti modelli logici:

- MOLAP: i dati vengono salvati utilizzando strutture intrinsecamente multidimensionali
- ROLAP: si utilizza il modello relazionale per andare a rappresentare i dati

In particolare la modellazione su sistemi relazionali è basata sul cosiddetto schema a stella e sulle sue varianti. Uno schema a stella è costituito da:

- Un insieme di relazioni chiamate dimension table ognuna delle quali modella una dimensione del cubo. Per ognuna di esse si ha una chiave primaria e un insieme di attributi che descrivono le dimensioni di analisi a diversi livelli di aggregazione
- Una relazione chiamata fact table al cui interno troviamo tutte le chiavi importate dalle varie dimension table le quali nel loro insieme vanno a costituirne la chiave primaria. In oltre per ogni attributo la fact table contiene un attributo per ogni misura

Una query q di tipo GPSJ è una proiezione generale su di una selezione su di uno o più join; può quindi essere espressa in algebra relazionale su di uno schema a stella in questo modo:

$$q = \pi_{G,M}\, \sigma_{Pred}\, (FT^{(0)} \bowtie DT_1^{(0)} \bowtie \ldots \bowtie DT_n^{(0)})$$

Dove Pred è una congiunzione di semplici predicati di range sugli attributi della dimension table, G è un insieme di attributi della dimension table e M è un insieme di misure aggregate ognuna definita applicando un operatore di aggregazione ad una misura in $FT^{(0)}$. La proiezione generale $\pi_{G,M}$ dal punto di vista dell'SQL corrisponde ad un group by sugli attributi in G e inserire G e M nelle clausole di selezione. [15] Per la gestione di tali query Spark utilizza un piano di esecuzione costituito da Job, stage e task.

3.2 Piano di esecuzione logico

3.2.1 Job

In cima alla gerarchia di esecuzione ci sono i Job. Invocare una action in un applicazione Spark innesca il lancio di un Job per poterla realizzare, viene quindi creato almeno un Job per ogni action richiesta dal programma dell'utente. Un Job è associato ad una catena di dipendenze tra RDD organizzate su di un grafo aciclico diretto (DAG). Quindi per specificare come questo Job deve essere costituito, Spark esamina il grafo degli RDD da cui l'action dipende e crea un piano di esecuzione. Questo piano parte dai primi RDD, cioè da quelli che non dipendono da altri RDD (ad esempio quando i dati vengono letti da un file in HDFS o su di una tabella hive) o che referenziano dati già presenti in cache e si conclude con l'ultimo RDD richiesto per produrre il risultato della action. [16] All'interno di un'applicazione Spark (un'istanza dello SparkContext) possono essere lanciati più Job (Spark action) in maniera concorrente se vengono innescati da thread differenti. Questo può essere comune laddove l'applicazione stia rispondendo a richieste sulla rete; ad esempio il server Shark lavorava in questo modo. Un caso particolare è invece quello del Broadcast Hash Join il quale implica una prima fase di collect sul driver anche se non esplicitamente specificato dal programma dell'utente e che quindi va a creare un nuovo Job in maniera implicita.

L'esecuzione di una query su di un'applicazione Spark può quindi essere vista come un job costituito da più stage dove ogni stage ha il compito di eseguire una determinata operazione andando a realizzare uno schema ad albero nel quale l'esecuzione dei vari stage porta al risultato finale del job e quindi della query. Nel caso specifico delle GPSJ una classica divisione in stage del job è quella in Figura 18

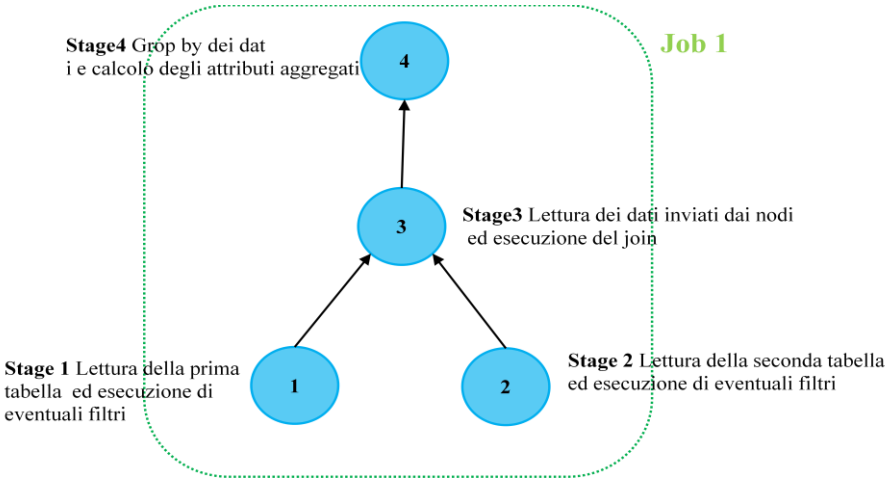

Figura 18 Struttura ad albero del job e degli stage che caratterizzano l'esecuzione di una GPSJ

Di default lo scheduler Spark esegue i job con una metodologia FIFO. Il primo Job ha priorità su tutte le risorse utilizzabili quando i suoi stage hanno task da lanciare, poi il secondo Job prende la priorità e così via. Se i Job in testa alla coda non hanno bisogno di utilizzare tutto il cluster, i Job successivi possono partire subito, se invece i Job in testa sono molto pesanti i successivi possono subire significativi ritardi. [16]

3.2.2 Stage

Uno Stage corrisponde ad una collezione di task indipendenti che eseguono tutti lo stesso codice su di un diverso subset di dati. Il piano di esecuzione consiste nell'assemblare le trasformation all'interno di un Job in stage. Ogni stage contiene una sequenza di transformation che possono essere completate senza che sia necessario effettuare uno shuffeling dei dati. [15] Si può vedere in Figura 19 come lo shuffleing (causato dall'operazione di group by o dal join) porti alla creazione di nuovi stage mentre operazioni come il map e la union possano essere fatte all'interno dello stesso stage.

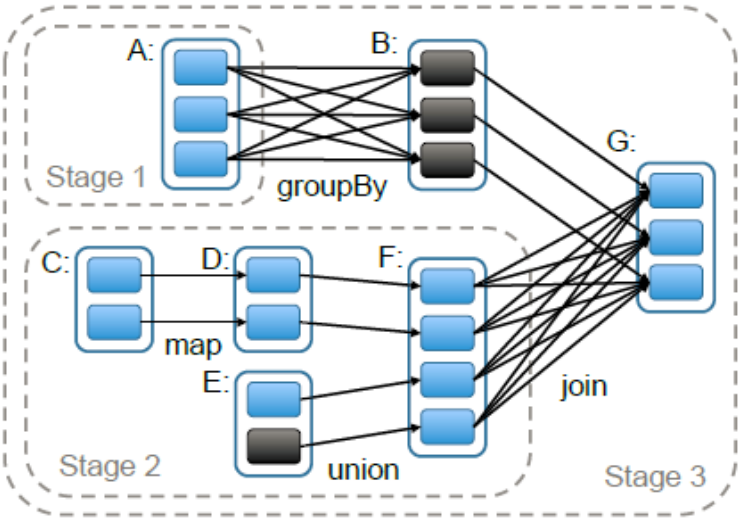

Figura 19 Esempio di suddivisione di un job in stage e suddivisione degli stessi

Ogni DAG eseguito dallo scheduler viene quindi diviso in stage laddove avvenga uno shuffeling dei dati permettendo allo scheduler di eseguire tali stage in ordine topologico.
Gli stage si differenziano in due tipologie:

- Shuffle Map Stage: in essi i risultati dei task diventano input per gli stage successivi
- Result Stage: in essi i task elaborano direttamente l'action che ha avviato il Job (ad esempio count(), save()) e ritornano un risultato.

Ogni stage ha anche un JobId il quale indentifica il Job che per primo ha innescato lo stage.
Quando viene utilizzato uno scheduling FIFO questo permette agli stage dei primi Job di essere elaborati per primi o comunque di essere recuperati per primi in caso di failure.

3.2.3 Task

Un task è un'unità di esecuzione indipendente; un comando inviato dal driver all'executor serializzando la funzione che deve eseguire. L'executor deserializza la funzione e la esegue attraverso il task. Anche i task come gli stage sono divisi in due tipologie:

- ShuffleMapTask: il task esegue la sua funzione e divide il suo output in diverse parti (basandosi su di un partizionatore specificato in ShuffleDependency) le quali dovranno poi essere inviate ai vari executor che li richiedono
- Result Task: esegue la sua funzione e invia il suo output al driver

L'ultimo stage di un Job è costituito da una serie di ResultTask, mentre gli stage precedenti sono costituiti da ShuffleMapTask. [17] Come si evidenzia in Figura 20 ogni task può essere visto essenzialmente come costituito da 3 fasi principali:

- Fetch input: caricamento dei dati da file o da cache nel caso di dati già elaborate da task precedenti
- Execute the task: esecuzione vera e propria della funzione del task
- Write output: l'output del task viene materializzato come dato di shuffle o come risultato per il driver

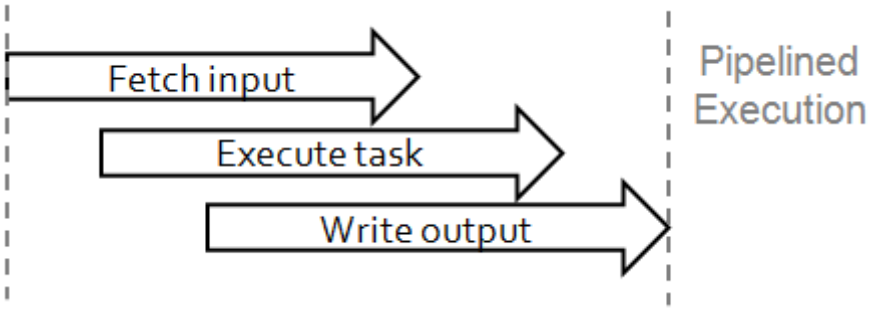

Figura 20 Principali fasi di un task Spark

3.2.4 Catalyst

Le query in SQL specificano quali dati si vogliono ottenere ma non specificano il come; quindi il database è libero di scegliere la strada migliore attraverso un processo di ottimizzazione. Un buon ottimizzatore di query è capace in maniera automatica di riscrivere le query stesse ed eseguirle in maniera più efficiente utilizzando diverse tecniche come anticipare la fase di filtraggio dei dati, utilizzare gli indici disponibili o assicurarsi che data source differenti siano messe in join nella maniera più efficiente. Attraverso queste trasformazioni l'ottimizzatore non solo migliora il tempo di esecuzione delle query ma in più permette agli sviluppatori di focalizzarsi sulla semantica delle loro applicazioni piuttosto che sulle performance. [18]

Quando è stato realizzato SparkSQL è stato progettato anche un ottimizzatore estendibile chiamato Catalyst che si basa sui costrutti del linguaggio funzionale Scala (come ad esempio i quasiquotes e il pattern matching). Catalyst è stato realizzato fin dal principio con l'idea di base di poterlo estendere questo per raggiungere due obiettivi:

- Rendere semplice l'aggiunta di tecniche di ottimizzazione per SparkSQL
- Rendere possibile e semplice la sua estensione anche a sviluppatori esterni per esempio attraverso l'aggiunta di regole su specifici data source che permettano il push down delle operazioni di filtering e di aggregation in sistemi di storage esterni, oppure aggiungendo il supporto a nuovi data type

Il cuore di Cataylst contiene una libreria generale per la rappresentazione degli alberi e l'applicazione di regole per manipolarli. Gli alberi sono composti da oggetti nodo , ognuno di essi può avere da uno a molti figli e la definizione di un nuovo tipo di nodo avviene mediante la definizione di una nuova classe che erediti dalla classe TreeNode. Essi sono oggetti immutabili e possono essere manipolati utilizzando trasformazioni funzionali. Sugli alberi si possono eseguire delle manipolazioni tramite l'utilizzo di regole le quali sono funzioni che creano un albero a partire da un altro albero. Anche se una regola può eseguire codice sul suo albero di input (essendo tale albero un oggeto Scala) l'approccio più comune è quello di utilizzare un insieme di funzioni di pattern matching che trovano e sostituiscono i sotto-alberi con una specifica struttura.
Sopra a questo framework di gestione degli alberi sono state poi realizzate delle librerie specifiche per il processing delle query relazionali (es. espressioni, piani logici) e un insieme di regole che gestiscono diverse fasi dell'esecuzione delle query: analisi, ottimizzazione logica, pianificazione fisica e code generation per la compilazione di parti di query in Java bytecode. [19]

3.3 Struttura del cluster

Per far si che il cluster effettui decisioni appropriate sulle azioni da intraprendere esso deve essere a conoscenza della sua topologia che viene definita durante la fase di configurazione.
La memorizzazione dei blocchi e l'allocazione dei processi (data locality) sono funzionalità che necessitano di questa informazione. Un cluster è costituito da un certo numero di nodi ognuno dei quali è una macchina elaborativa ovvero un server fisico o virtuale che prende parte al cluster stesso. Ogni nodo in un cluster Hadoop quindi possiede:

- un disco locale su cui sono memorizzati i dati salvati sul cluster (HDFS) ed eventuali dati intermedi (ad esempio i dati di shuffeling prima di essere inviati)
- Un certo numero di core su cui eseguire le elaborazioni in parallelo
- Una connessione alla rete per comunicare con gli altri nodi.
- Un node manager incaricato di gestire le risorse del nodo utilizzate dai container

I nodi sono poi organizzati in rack e un insieme di rack costituisce un data center. Un cluster Hadoop modella questi concetti in una struttura ad albero ed elabora la distanza tra i nodi come distanza sull'albero. Nell'esempio di Figura 21 viene mostrato un semplice cluster costituito da un unico data center, due rack distinti e 4 nodi in ogni rack.

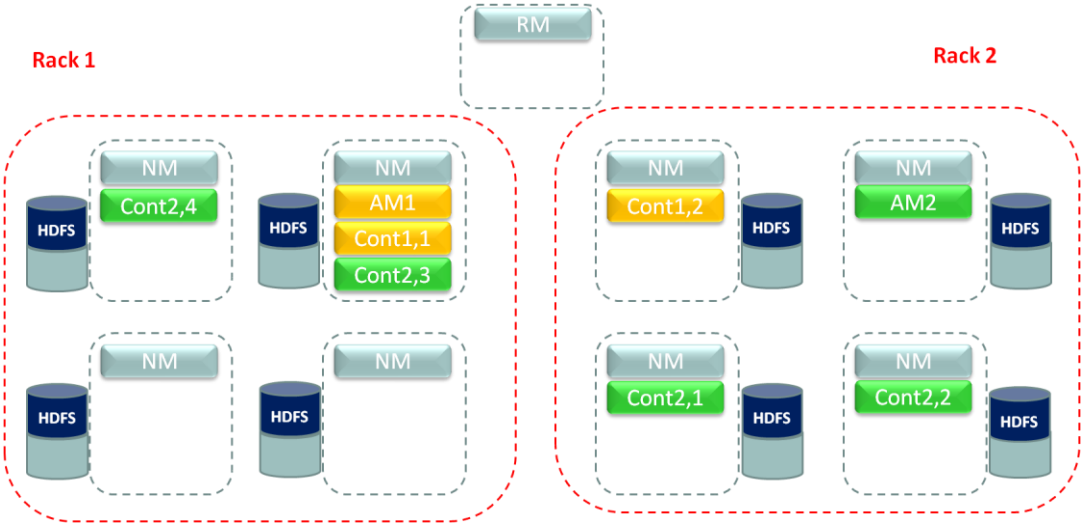

Figura 21 Esempio di cluster hadoop

Per sfruttare la data locality risulta quindi necessario definire un concetto di distanza all'interno del cluster. Tale distanza viene realizzata considerando il cluster come un albero e andando a utilizzare la distanza su di esso come riferimento per la distanza tra nodi come si può vedere nell'esempio di Figura 22

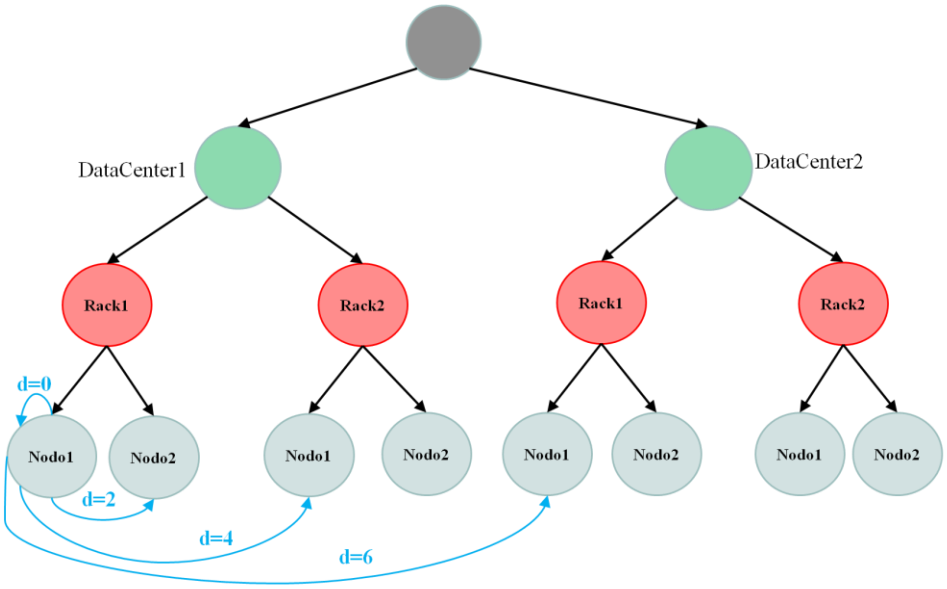

Figura 22 Rappresentazione ad albero del cluster e distanza dei nodi

Tipicamente ci sono dai 30 ai 40 nodi su ogni rack con uno switch da 1Gb per ogni rack.

Il punto saliente è che la banda tra nodi sullo stesso rack risulta essere molto maggiore piuttosto che tra nodi su rack differenti. [20]

3.4 Dal Piano di esecuzione logico al piano di esecuzione fisico

Tutta la gestione del piano di esecuzione (logico e fisico) è affidata a Catalyst il quale realizza i piani e ne sceglie il migliore.

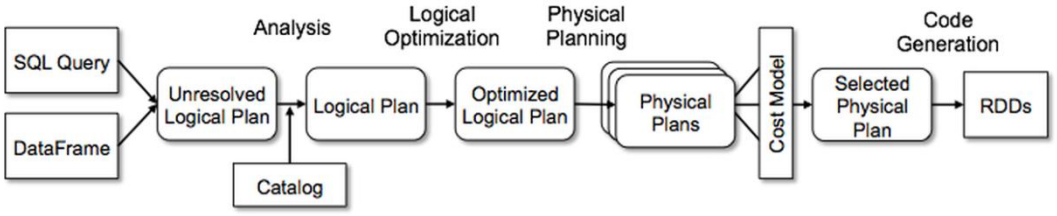

Figura 23 Flusso per la gestione di una query da parte dell'ottimizzatore Catalyst

La fase di ottimizzazione logica applica ottimizzazioni basate su regole standard al piano logico. Questo include il costant folding, il push-down dei predicati, il pruning projection, la null propagation, la semplificazione delle espressioni booleane e diverse altre regole.

Nella fase di realizzazione del piano fisico Catalyst può generare diversi piani e compararli sulla base del costo. In questa fase Spark SQL prende un piano logico e genera uno o più piani fisici utilizzando degli operatori fisici che combaciano con quelli del motore di esecuzione di Spark. Successivamente Catalyst seleziona un piano basandosi su di un modello di costo, ma in questo momento l'ottimizzazione basata su tale costo è utilizzata solo per selezionare l'algoritmo di join: per tabelle che Spark riconosce essere piccole (una delle due in join) viene utilizzata una broadcast join utilizzando la distribuzione broadcast peer-to-peer disponibile in Spark, mentre per tabelle troppo grandi per essere gestite in memoria per intero viene utilizzata uno shuffle join che suddivide entrambe le tabelle sui nodi. Nella fase di creazione del piano fisico vengono eseguite ottimizzazioni fisiche basate su regole come per esempio effettuare il pipelining delle projection (select) e dei filtri (where) cioè facendo un merge di queste due operazioni di trasformation all'interno di una singola map permettendo quindi di evitare l'esecuzione di due fasi di map e reduce e quindi evitando di scrivere e rileggere i dati intermedi per due volte . In più viene anche effettuato un push down di determinate operazioni portando ad esempio il filtro dei dati sulle operazioni iniziali non appena i dati vengono prelevati permettendo quindi di dover gestire una quantità di dati inferiore sia nella fase di reduce ma anche durante la di trasmissione in rete. [19] Allo stesso modo viene effettuato un push down dell'operazione di calcolo dei valori aggregati nel caso di group by. Dato che normalmente i dati si trovano su diversi nodi è necessario effettuare lo shuffling di tali dati anche nel caso del group by. Ogni mapper quindi preleva i dati della propria partizione e quando va a scriverne il risultato per effettuare

lo shuffeling, consapevole della necessità di effettuare successivamente un'operazione di group by non solo li organizza in gruppi basati sull'attributo di group by ma ne calcola già il valore aggregato degli attributi selezionati permettendo quindi di scrivere e inviare sulla rete una quantità di dati minore. Nella successiva fase di reduce quindi sarà necessario solamente ottenere tutti i dati dei gruppi che devono essere inviati a quel task di reduce (in base alla funzione di hash) e calcolarne i dati aggregati ma a partire da dati già parzialmente aggregati. In Figura 24 si ha un esempio di group by sull'"Attributo di Group By" e si richiede il valore massimo dell'"Attributo Selezionato"; si può osservare come il mapper pre aggreghi e calcoli il valore massimo dei dati che legge dal blocco in input.

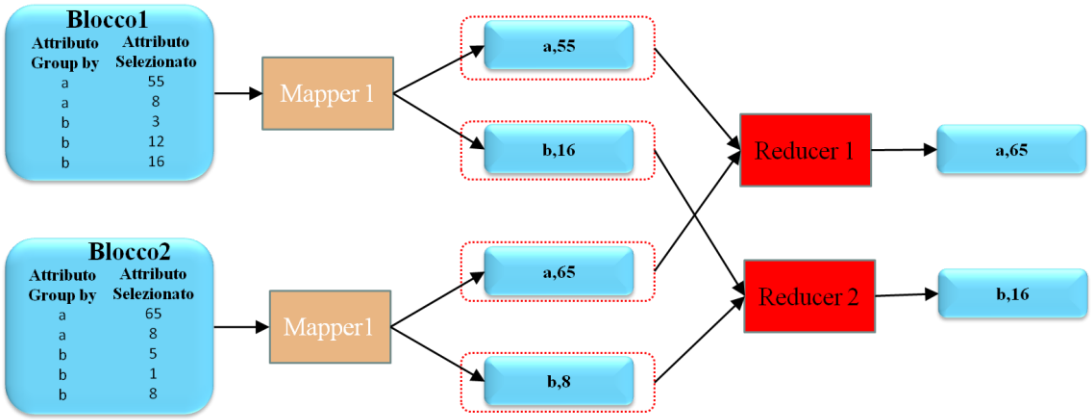

Figura 24 Esempio di push down dell'operazione di calcolo del valore aggregato sul mapper in caso di group by

Per realizzare una funzione di costo consistente si è cercato quindi di comprendere appieno il funzionamento prima logico e poi fisico dei piani realizzati da Spark per l'esecuzione delle query GPSJ.

3.4.1 Lettura tabelle

La prima fase che deve essere considerata nella valutazione di una funzione di costo nell'esecuzione di una query GSPJ è quella relativa al caricamento dei dati. I dati che vengono letti sono salvati in memoria centrale andandoli a caricare su uno o più RDD. Gli RDD possono avere diverse sorgenti dati che possono essere sia relative a strutture dati realizzate dal programma utente (es. array) su cui viene applicato il metodo parallelize() oppure possono essere caricate direttamente dal disco. Si è quindi analizzato il caso di caricamento dei dati da disco considerandolo come il caso più comune (es. caricamento in memoria dei dati di una tabella Hive) e il più pesante avendo la necessità di effettuare letture

su disco e in diversi casi anche di un invio sulla rete di quei dati che sono presenti su altri nodi.

Prima di tutto è necessario comprendere come è strutturata una tabella Hive memorizzata su HDFS. HDFS è un file system distribuito e quindi partizioni della tabella in questione sono sparse sui vari nodi del cluster come si può vedere in Figura 25. Il numero di blocchi ($TableBlock$) è dato dalla dimensione della tabella diviso la dimensione di tali blocchi (di default 128Mb).

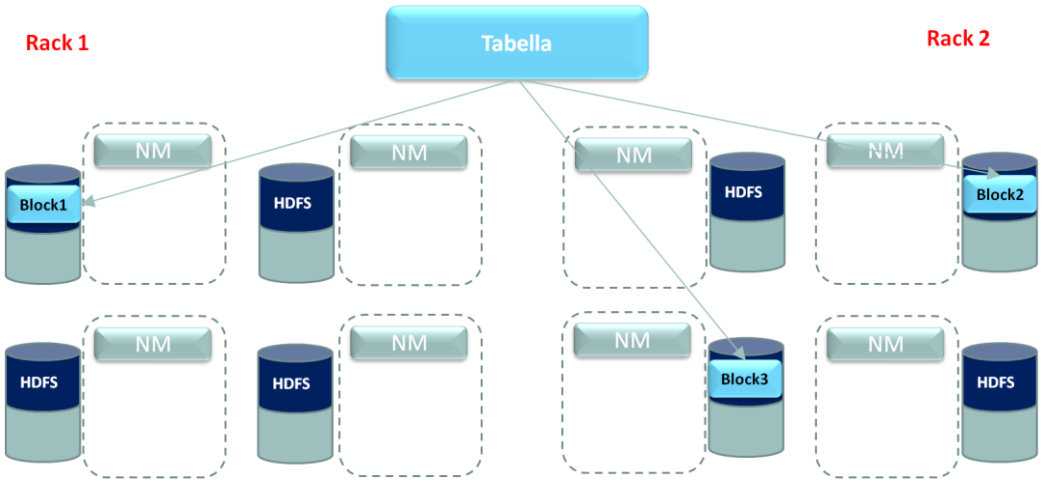

Figura 25 Distribuzione dei blocchi in cui è divisa la tabella sul cluster

Supponendo che ogni executor si trovi in un nodo differente sarà necessario caricare in memoria ed elaborare per ogni executor un numero di blocchi pari a

$$ExecBlocks = TableBlock/Exec$$

blocchi dove $Exec$ indica il numero di executor utilizzati dall'applicazione. In più è necessario considerare che tali blocchi sono replicati all'interno del cluster in base ad un fattore di ridondanza ($BlockRedundancy$).Nell'esempio in Figura 26 si ha un valore di ridondanza di 3 e quindi si può notare come per ogni blocco siano creati altri due blocchi su due nodi differenti di un altro rack. Il numero totale di blocchi presenti sul cluster diventa quindi:

$$TotalTableBlock = BlockTablex * BlockRedundancy$$

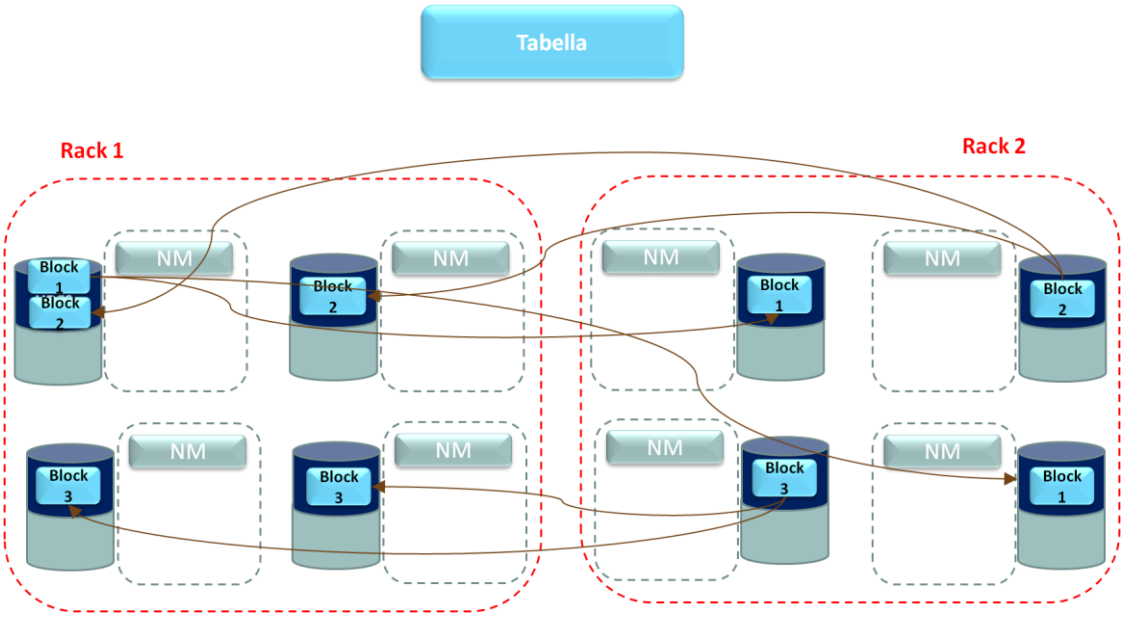

Figura 26 Distribuzione dei blocchi della tabella sul cluster considerando una ridondanza 3

I blocchi che devono essere caricati su ogni nodo dove risiede un executor possono essere suddivisi in 3 categorie in base alla loro provenienza:

- Blocchi che risiedono sul nodo dove l'executor è in esecuzione
- Blocchi che risiedono sullo stesso rack del nodo
- Blocchi che risiedono su di un rack differente.

Blocchi sul nodo locale

Supponendo una distribuzione uniforme sui nodi del cluster di tali blocchi avremo su ogni nodo un numero di blocchi pari a:

$$LocalBlock = (TableBlock * BlockRedundancy)/HDFSNodes$$

dove $HDFSNodes$ indica il numero di nodi da cui è formato il cluster e su cui possono risiedere blocchi HDFS. Come si può osservare dalla Figura 27 tali blocchi sono memorizzati su HDFS ma trovandosi sul disco locale del nodo possono essere letti direttamente dall'executor. Questo permette di avere prestazioni migliori dato che non si devono considerare tutti i tempi relativi all'invio dei dati sulla rete e alle diverse richieste che devono essere fatte dagli executor

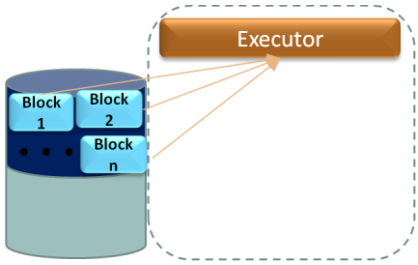

Figura 27 Blocchi della tabella caricati dal nodo locale

Ora è possibile considerare il numero di blocchi che verranno caricati localmente su ogni nodo:

$$LocallyLoadedBlocks = MIN(ExecBlocks, LocalBlocks)$$

Cerchiamo il minimo tra $ExecBlocks$ e $LocalBlocks$ perché, se il numero di blocchi richiesti da ogni executor risulta minore dei blocchi presenti sul nodo, allora il valore dei blocchi caricati localmente sarà dato solo da $ExecBlocks$. Su questi blocchi dovremo quindi considerare come tempo per la loro lettura solo il tempo di lettura da disco ($DiskSpeed$).

Blocchi sullo stesso rack

Nel caso in cui $ExecBlocks$ sia maggiore di $LocalBlocks$ allora vuol dire che sarà necessario far arrivare sul nodo dell'executor blocchi presenti su altri nodi:

$$NONLocallyLoadedBlocks = MAX(0, ExecBlocks - LocallyLoadedBlocks)$$

Allo stesso modo con cui è stato calcolato il numero di blocchi su ogni nodo possiamo valutare il numero di blocchi sul rack dividendo il numero totale di blocchi della tabella per il numero di rack:

$$BlockPerRack = (BlockRedundancy * TableBlock) \,/\, Rack$$

dove $Rack$ indica il numero di Rack del cluster.
Risulta necessario anche considerare quanti executor su ogni rack siano presenti considerando una distribuzione uniforme degli stessi:

$$ExecPerRack = Exec/\, Rack.$$

Dove $Exec$ indica il numero di executor utilizzati.

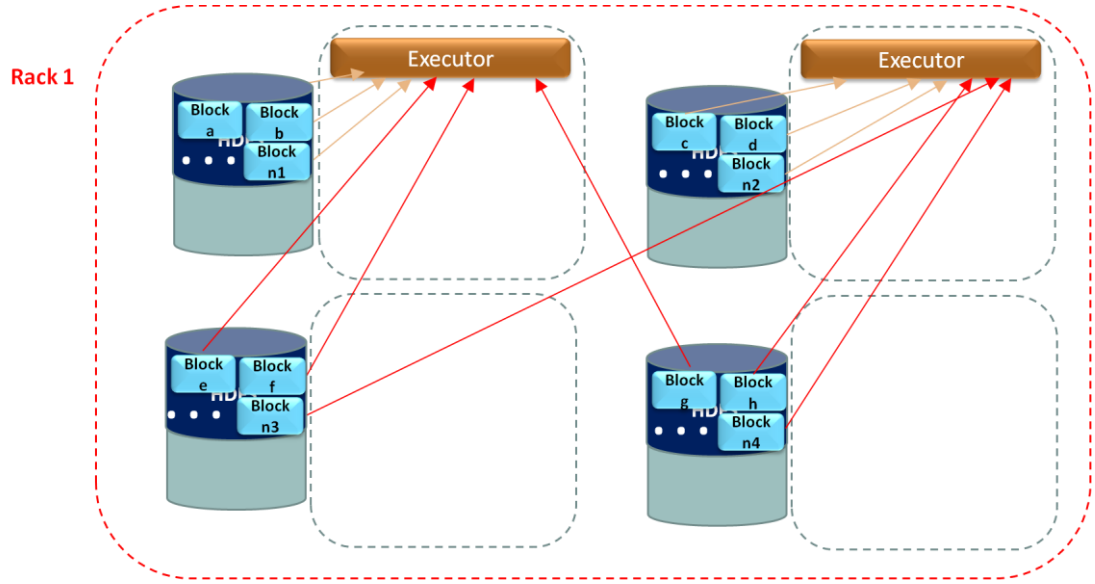

Figura 28 Trasferimento dei blocchi all'interno di un rack. In rosso le frecce che indicano un trasferimento intra-rack

In Figura 28 viene raffigurato con le frecce rosse l'invio dei blocchi all'interno dello stesso rack ma su nodi differenti da quelli dove risiedono gli executor. Possiamo ora calcolare il numero di blocchi inviati all'interno del rack:

$$RackLoadedBlocks = MIN(ExecBlock - LocallyLoadedBlocks; (BlockPerRack - (ExecPerRack\ x\ LocallyLoadedBlocks))$$

Nella formula abbiamo due componenti di cui prenderne il minimo:

- Differenza tra i blocchi che devono essere caricati per l'executor e quelli caricati in locale (questo ci indica quanti blocchi devono essere caricati al di fuori del nodo locale dell'executor)
- Differenza tra i blocchi presenti sul rack ($BlockPerRack$) e il totale dei blocchi caricati dal nodo locale su tutti i nodi del rack ($ExecPerRack$ x $LocallyLoadedBlocks$)

Il minore tra i due componenti ci restituisce i blocchi inviati all'interno del rack ($RackLoadedBlocks$). Se il valore minimo è il primo allora significa che tutti i blocchi di cui abbiamo necessità si trovano sul nostro rack, mentre quando il valore minimo è il secondo significa che la quantità massima dei blocchi che devono ancora essere caricati supera quella dei blocchi presenti sul rack e quindi essa è la quantità massima di blocchi che possiamo prelevare dal rack.

Sui blocchi che vengono trasferiti all'interno del rack applicheremo quindi una velocità di trasferimento di tali blocchi sulla rete interna di un rack ($IntraRackTransferSpeed$).

Blocchi sullo stesso data-center

Considerando un cluster costituito da più rack e un solo data-center possono sussistere casi in cui i dati da caricare siano molti (non bastano quelli presenti sul rack dell'executor) e risulta quindi necessario che vengano inviati da altri rack come si può vedere in Figura 29 dove parte dei blocchi necessari per l'esecuzione dei task degli executor (sul rack1) deve essere prelevata dal rack2.

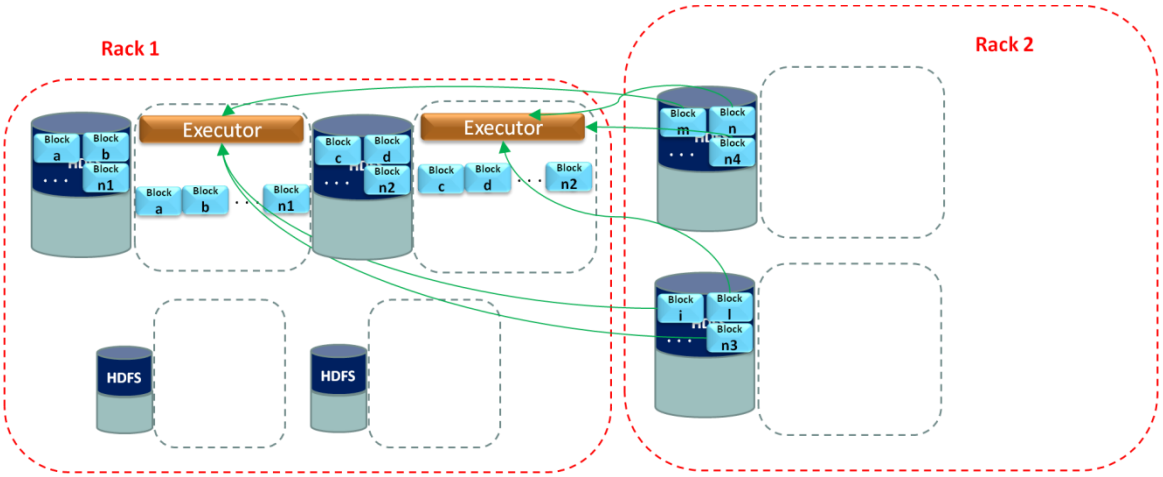

Figura 29 Trasferimento dei blocchi inter rack

Per calcolare questa quantità sarà sufficiente togliere al valore totale dei blocchi che devono essere caricati su ogni executor i blocchi già caricati localmente o intra-rack:

$$DataCenterLoadedBlocks = MAX(0; ExecBlock - LocallyLoadedBlocks - RackLoadedBlocks)$$

In questo caso utilizziamo il valore massimo tra zero e la differenza appena descritta per i casi in cui non ci sia necessità di caricare blocchi inter-rack (il secondo termine risulta minore di zero). Per calcolare il tempo di invio di tali blocchi applicheremo una velocità di trasferimento dei dati inter-rack ($InterRackTansferSpeed$).

Tempi e parallelizzazione

Ora che è nota la distribuzione dei blocchi nelle tre categorie possiamo calcolarne il tempo di lettura totale. Come detto nelle sezioni precedenti Spark utilizza diverse forme di ottimizzazione del piano logico e del piano fisico soprattutto indirizzate a diminuire la quantità di dati che viaggiano nella rete e che devono essere scritti su disco essendo questi i principali colli di bottiglia durante l'esecuzione delle query. In questo caso ad esempio è

necessario considerare che viene effettuato un push down della projection e quindi fin dai primi stage Spark sa già quali attributi serviranno evitando di andare a prendere tutta la tupla, diminuendo quindi la quantità di dati interessati. Considerando che con una memorizzazione dei dati basata su parquet il blocco è partizionato in colonne, la quantità di dati di un singolo blocco che devono essere lette è una frazione del totale essendo il blocco stesso suddiviso come si può notare in Figura 30 . Per ogni colonna si deve considerare un peso dato dal tipo di dato che tratta (ES int 4 byte,double 8 byte ecc ecc) e si sommano tutti i pesi delle colonne che sono selezionate nella query. Per ottenere la frazione di dati del blocco che devono essere letti è sufficiente dividere la somma dei pesi delle colonne selezionate per il totale dei pesi di tutte le colonne:

$$QueryAttr = \frac{\sum_{i=1}^{i=m} Ai * peso(Ai)}{\sum_{j=1}^{j=n} Aj * peso(Aj)}$$

Dove:

- m = numero di attributi selezionati
- n = numero totale degli attributi della tabella
- Ai= i-esimo attributo selezionato
- Aj= j-esimo attributo della tabella
- $peso(Ai)$ = peso dell'attributo

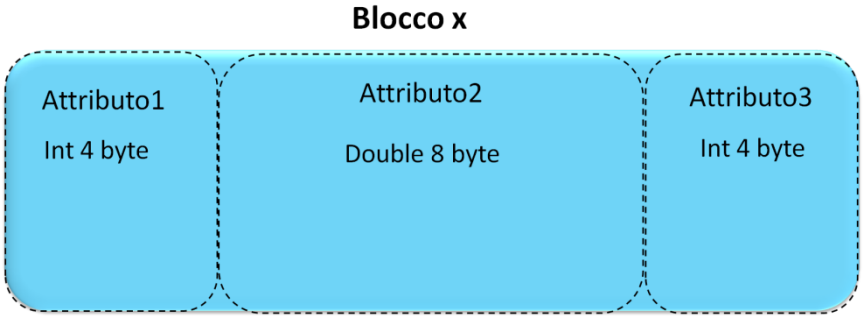

Figura 30 Suddivisione di un blocco HDFS nel caso si utilizzi parquet

Moltiplicando la dimensione del blocco per $QueryAttr$ si ottiene la frazione di dati del blocco che devono essere caricati in memoria.

$$BlockPortionRead = BlockSize * QueryAttr$$

Ora è possibile calcolare il tempo di fetch dei dati considerando il fetch dei blocchi che già si trovano sul nodo locale:

$$FetchTimeLocal = \frac{LocallyLoadedBlock * DiskOverloading * BlockPortionRead}{DiskSpeed * VCore}$$

Consideriamo poi il tempo di fetch dei dati sui nodi remoti per preparali all'invio dopo che sono stati richiesti dall'executor che ne ha necessità e non li possiede sul nodo locale:

$$FetchTimeRack = \frac{RackLoadedBlock * ExternalDiskOverloading * BlockPortionRead}{DiskSpeed * VCore}$$

In questo caso si indica con ExternalDiskOverloading un overloading del disco differente da quello locale dato che in un cluster piuttosto grande può risultare improbabile che vengano richiesti molti blocchi nello stesso momento sullo stesso nodo ma potrebbe succedere che siano in esecuzione altre applicazioni che vadano e intaccare la velocità di lettura del disco del nodo remoto. Allo stesso modo ma considerando i blocchi prelevati extra rack si può calcolare il $FetchTimeExtraRack$. Tali dati mentre vengono letti dal disco vengono anche inviati sulla rete; possiamo calcolare quindi il tempo di trasferimento per i dati che si trovino sullo stesso rack come:

$$ExecTransferTimeIntraRack = \left(\frac{RackLoadedBlock}{IntraRackTrasferSpeed}\right) * BlockPortionRead * NetworkOverloading$$

Mentre quelli che si trovano al di fuori del rack dell'executor:

$$ExecTransferTimeInterRack = \left(\frac{DataCenterLoadedBlocks}{InterRackTransferSpeed}\right) * BlockPortionRead * NetworkOverloading$$

Per semplificare si è considerato come tempo per ottenere i dati che si trovano su nodi remoti il massimo tra il tempo di trasferimento e il tempo necessario al nodo remoto per leggerli dal disco, questo perché in casi in cui la rete sia molto veloce l'unico collo di bottiglia sarebbe rappresentato dal disco (e viceversa) permettendo comunque alla rete di procedere all'invio dei dati mentre il disco continua a leggere:

$$FetchTimeRackFinal = MAX(FetchTimeRack; ExecTransferTimeIntraRack)$$

Allo stesso modo si calcola il tempo finale di fetch dei dati da nodi al di fuori del rack:

$$FetchTimeDataCenterFinal = MAX(FetchTimeExtraRack; ExecTransferTimeIntraRack)$$

In questo modo possiamo ora calcolare il tempo totale di lettura dei dati:

$$TotalReadTime = FetchTimeLocal + FetchTimeRackFinal + FetchTimeDataCenterFinal$$

3.4.2 Shuffle Join

Tra le varie operazioni che possono essere eseguite sugli RDD hanno un ruolo particolare le wide trasformation: processi in cui la logica di elaborazione dipende da dati che risiedono su diverse partizioni e quindi risulta necessario effettuare una movimentazione di tali dati per portarli assieme in un unico punto. L'operazione di shuffle è una delle più costose e che più degradano le performance dei job e in Spark sono presenti anche numerose sorgenti di inefficienza che possono caratterizzare la fase di shuffeling. Anche se Spark cerca di evitare il più possibile lo shuffeling dei dati alcune operazioni richiedono per forza il trasferimento dei dati per poter ottenere il risultato. Queste operazioni richiedono che ogni nodo faccia il fetch dei dati dagli altri nodi per avere sufficienti dati per l'elaborazione del risultato. [21]
Nei framework tradizionali che utilizzano MapReduce la fase di shuffeling viene spesso nascosta dalle fasi di Map e di Reduce, infatti la fase di shuffle dei dati viene spesso integrata dentro la fase di Reduce anche se ha poco a che vedere con la semantica dei dati. [22]

Esecuzione

Nella fase di Map i task di Spark invece di conservare un buffer in memoria centrale scrivono il loro risultato direttamente su disco [22]. In particolare ogni Map task scrive un file di shuffle per ogni Reduce task il quale corrisponde al blocco logico di Spark. Il concetto di blocco è riferito al modello dei dati di Spark dove ogni data set (RDD e file di shuffle) è composto al livello di granularità più fine da blocchi individuali.

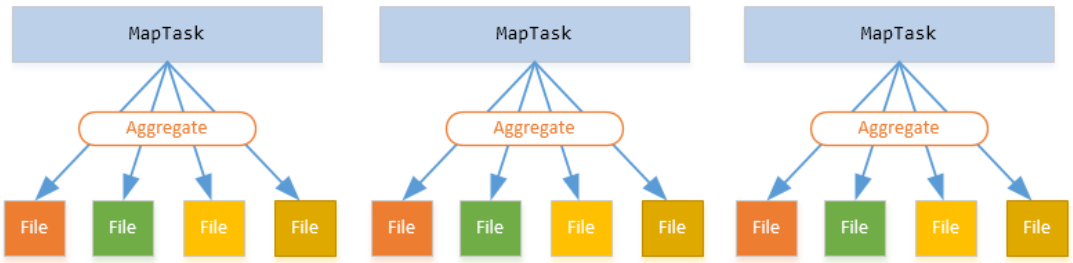

Figura 31 Suddivisione dell'input da parte di ogni map task in più output file

Quindi ogni map task M scrive R shuffle file (si veda Figura 31) dove R indica il numero di reduce task che verranno utilizzati nella fase di reduce. Questo porterebbe alla creazione di un numero di file molto elevato (M*R) che potrebbe degradare le performance a causa delle

numerose richieste di I/O sul disco. Per questo è possibile utilizzare la shuffle file consolidation attraverso il parametro *spark.shuffle.consolidateFiles* il quale settato a true fa in modo che sia mantenuto un solo shuffle file per ogni partizione riducendo fortemente il numero di file che devono essere elaborati come si può vedere in Figura 32 dove i vari bucket creati dai mapper vengono uniti in un unico file per ogni reducer.

Un'altra ottimizzazione fornita da Spark consiste nella possibilità di comprimere i dati che devono essere inviati attraverso il parametro *spark.shuffle.compress* per diminuire la quantità di traffico sulla rete. Nella fase di Reduce i dati vengono letti e caricati in memoria centrale per essere elaborati.

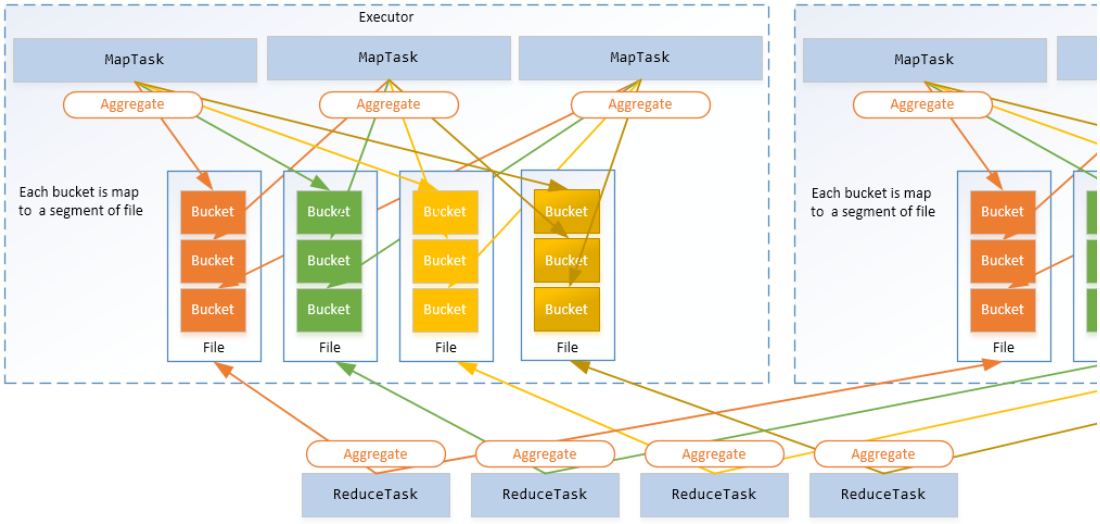

Figura 32 Shuffeling dei dati utilizzando la consolidation

Per far questo ogni reduce task mantiene in memoria un network buffer per fare il fetching dei risultati della fase di map. Anche la dimensione di questo buffer può essere specificata attraverso l'apposito parametro *spark.shuffle.maxMbInFlight*. In particolare i dati shuffle vengono letti mediante una connessione TCP tra un executor e l'altro. Se l'executor A richiede dei dati shuffle all'executor B viene prima di tutto inviato un OpenBlocks message nel quale viene specificata una lista di tutti i blocchi che A necessita.

Tale messaggio fa si che l'executor B inizi a mettere in memoria centrale i dati prelevandoli dal disco. Successivamente A richiede un blocco alla volta a B utilizzando un ChunkFetchRequest message anche se è possibile che più blocchi siano inviati in maniera concorrente tra A e B a causa dei diversi task in esecuzione nello stesso momento. [23]

Il più classico caso in cui si necessita di effettuare lo shuffeling dei dati avviene proprio per l'esecuzione dei join. Nel caso di shuffle join tra due tabelle entrambe dovranno essere partizionate sull'attributo di chiave. In questo modo ogni partizione nella quale il campo di

join avrà lo stesso valore come risultato della funzione di hash verrà inviata allo stesso task e questo permetterà di eseguire quella porzione di join.

Per realizzare un modello di tale operazione è necessario quindi stimare la quantità di dati che devono essere prima scritti sul disco nella fase di map e poi inviati sulla rete nella fase di reduce. Come nei casi precedenti in questo modello si considerano solo i tempi di scrittura del disco e invio sulla rete dato che il tempo di elaborazione dei dati non viene considerato un collo di bottiglia avendo generalmente tempi inferiori rispetto a quelli delle altre due operazioni. Per valutare quindi la quantità di dati si è deciso prima di tutto di stimare in maniera puntuale i dati che devono essere scritti su disco lato mapper. Per far questo si è deciso di considerare dapprima il numero di tuple che sono interessate dall'operazione e quindi tutte quelle tuple che nella fase precedente sono state lette da disco e hanno dato esito positivo agli eventuali filtri posti. In più per ogni tupla è necessario anche considerare quali attributi interesseranno le fasi successive dato che come detto l'ottimizzazione applicata da Spark tramite Catalyst permette già in fase di fetch dei dati di essere a conoscenza di quali attributi siano necessario andando a diminuire la quantità di informazioni che devono essere scritte e inviate. Questo chiaramente influisce anche nella fase di shuffle dato che la tupla che andremo a considerare avrà quindi una dimensione differente.

Il task preposto alla scrittura dei dati di shuffle quindi ha in input una serie di tuple che dovrà scrivere su disco per essere inviati sulla rete, risulta quindi necessario una serializzazione di tali dati. Attraverso un parametro di Spark (*spark.serializer*) può essere specificato il tipo di serializer utilizzato. Di default viene utilizzato il serializer di Java (JavaSerializer) che risulta essere più lento ma permette di avere la certezza di serializzare correttamente tutti gli oggetti Java. Un altro serializer che viene messo a disposizione dal framework è invece KryoSerializer che risulta essere più veloce ma non fornisce le stesse sicurezze riguardo la serializzazione di ogni tipologia di classe. In ogni caso può essere settato un serializer che erediti dalla classe org.Apache.spark.Serializer. Utilizzando quindi un serializer la quantità di dati scritti risulta essere maggiore rispetto alla semplice somma degli attributi della tupla considerati. E'stato quindi necessario come primo passo valutare tale quantità e come essa possa variare al variare degli attributi selezionati. Ovviamente nel caso di un join, qualsiasi siano gli attributi selezionati sarà necessario scrivere in shuffle write anche il valore dell'attributo di join dato che viene utilizzato per fare prima l'hash in fase di map e poi il join vero e proprio in fase di reduce. Partendo quindi da questa considerazione si è osservato come ci sia sempre una quantità fissa di byte che devono essere scritti per ogni tupla ($FixedTupleSize$) dovuta alla serializzazione della classe java che modella tale tupla e alla serializzazione dell'attributo di join. Si è potuto poi osservare come aggiungendo attributi in

selezione i dati scritti siano dipendenti dal loro tipo: se l'attributo in selezione aggiunto è dello stesso tipo della chiave di join o di altri attributi che sono selezionati allora la quantità di dati aggiunti alla tupla sarà la quantità stessa del suo tipo di dato (AttributeSize) mentre se il tipo di attributo risulta essere differente verrà aggiunta una quantità di byte superiore ($FixedTypeSize$) per la serializzazione del nuovo tipo. Come si può osservare nell'esempio di Figura 33 il task prende in input i dati dal blocco assegnatogli (prelevando solo le colonne che gli interessano, in questo caso la chiave di join e l'attributo selezionato). Esegue poi tutte le operazioni di elaborazione di tale input come ad esempio il filtro di tali tuple per poi generare in output le tuple serializzate che andranno a costituire l'input del successivo task reducer.

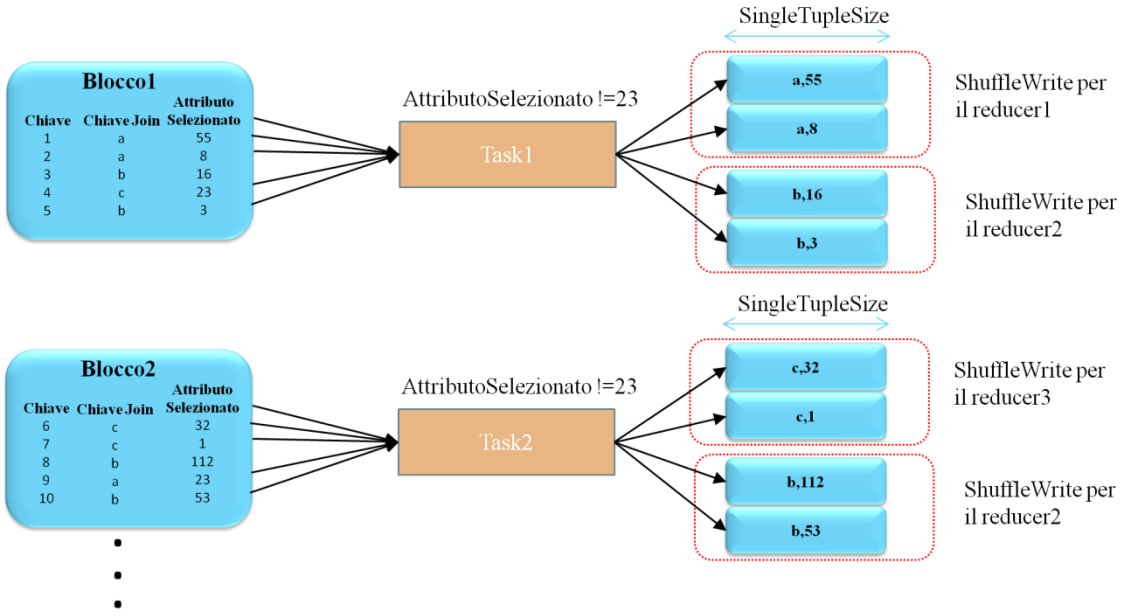

Figura 33 Esempio lettura di un blocco e scrittura dei dati in shuffle write nel caso di un shuffle hash join da parte di un task

Considerando quindi la divisione dell'output sui vari reducer basandosi su di una funzione di hash sull'attributo di join si può osservare come le tuple siano aggregate sotto lo stesso file in base a tale attributo per ogni reducer. Ogni tupla avrà una dimensione specifica ($SingleTupleSize$) che può essere calcolata con la formula di seguito.

Per ottenere la quantità di byte scritti per una singola tupla è quindi necessario applicare la seguente formula:

$$SingleTupleSize = FixedTupleSize + AttributeTypeNumer * FixedTypeSize + \sum_{i=0}^{i=n} Ai * Peso(Type(Ai))$$

Dove:

- $FixedTupleSize$: quantità di byte fissa dovuta alla serializzazione della classe java che modella la tupla e dell'attributo di join.
- $AttributeTypeNumer$: numero di tipi di dati distinti tra gli attributi selezionati
- $FixedTypeSize$: quantità di byte che devono essere aggiunti per ogni nuovo tipo di dato aggiunto alla tupla
- Ai: i-esimo attributo selezionato
- $Type(Ai)$: tipo dell'i-esimo attributo selezionato
- $Peso(Type(Ai))$: peso del tipo dell'i-esimo attributo selezionato (AttributeSize)

Per rendere più semplice e testabile il modello si sta considerando che i dati scritti in shuffle write dopo la serializzazione non vengano compressi; cosa che spesso viene fatta per rendere minore la quantità di dati inviati sulla rete e che devono essere scritti su disco. Senza utilizzare la compressione però è possibile avere un riscontro più preciso dei dati scritti e delle ipotesi fatte. In questo modo ora risulta semplice calcolare la quantità di dati scritti in shuffle write considerando le tuple che passano la fase di filtering e che dovranno essere considerate nel join (TupleJoinNumber) moltiplicandole per la quantità di byte che ogni tupla porta con se:

$$ShuffleWriteSize = TupleJoinNumber * SingleTupleSize$$

Questo risulta essere la quantità di dati scritti a livello dell'intero cluster per una singola tabella che deve essere inviata in shuffle. Ovviamente nel caso di un shuffle join le tabelle interessate risultano essere più di una e quindi il valore totale dei dati scritti in shuffle ($TotalShuffleWriteSize$) risulta essere:

$$TotalShuffleWriteSize = \sum_{i=0}^{i=n} ShuffleWriteSize_i$$

Ora abbiamo ottenuto il valore totale dello shuffle write per un'operazione di join tra due o più tabelle su tutto il cluster, considerando però una suddivisione uniforme del carico di lavoro e quindi dei dati stessi sui diversi executor che vengono settati per l'applicazione Spark possiamo ottenere la quantità di dati che ogni executor scrive:

$$ShuffleWriteSizePerExecutor = \frac{TotalShuffleWriteSize}{Exec}$$

Questi dati come detto vengono scritti su disco e quindi sarà sufficiente dividere tale quantità per la velocità di scrittura del disco per ottenere il tempo di scrittura dei dati da parte dei task mapper ($ShuffleWriteTime$). Risulta comunque necessario considerare che tale operazione viene eseguita in parallelo su più task; questo,come nel caso della lettura dei dati dalla tabella porta alla necessità di considerare una diminuzione dei tempi data dal parallelismo di tali operazioni ma anche un overloading delle risorse del disco che si possono andare a saturare soprattutto nel caso che siano assegnati diversi core per ogni executor.
Al contrario però del caso della lettura dei dati l'unica forma di overloading da considerare in questo caso è quella sul disco locale senza necessità di fare valutazioni sulla rete in questa fase dato che tale operazione avviene per l'appunto completamente sul disco locale.

$$ShuffleWriteTime = \frac{ShuffleWriteSizePerExecutor * DiskOverloading}{DiskSpeed * VCore}$$

Anche nella pratica si può osservare come con questa operazione si concluda il primo stage nel caso di una query che necessiti effettuare un join. Infatti i dati vengono scritti in shuffle write proprio perché risulta necessaria un'operazione di shuffle, la quale determina (come detto nelle sezioni precedenti) la realizzazione di un nuovo stage.
In questo secondo stage saranno presenti tutti i task reducer che avranno come input i dati precedentemente scritti in shuffle write i quali saranno poi elaborati per poi essere riscritti in forma di risultato finale nel caso si tratti dell'ultimo stage o nuovamente in forma di shuffle write nel caso in cui successivamente debbano essere eseguite altre operazioni. Nel caso di un shuffle join è necessario comunque considerare che le tabelle che devono essere suddivise ed inviate sulla rete sono almeno due. Questo comporta il fatto che i task reducer dello stage successivo dovranno andare a prelevare in input i dati (a loro assegnati tramite funzione di hash) da entrambe. Come detto i dati scritti in fase di shuffle write vengono scritti su disco locale quindi il fetch dei dati da parte dei task reducer riguarderà una parte dei dati che si trova sul disco locale data dai dati dei mapper che sono stati eseguiti su quel specifico nodo. In Figura 34 è presentato l'esempio di uno specifico nodo che esegue una parte dei mapper e una parte dei reducer necessari per eseguire il join. Si può osservare in particolare come i mapper prelevino i dati in input da diversi nodi (dato che vengono prelevati dati da HDFS) e e vadano a scrivere il risultato (200 file, cioè tanti quanti sono i reducers) sul nodo locale. Anche i reducers successivamente vanno a prelevare i dati assegnati ma in questo caso dai dischi locali del proprio nodo o da quello degli altri nodi dove sono stati eseguiti altri mapper. Si osservi anche come il numero di reducers siano 200 (valore di dafault utilizzato da Spark per le operazioni di shuffle) su tutto il cluster e non 200 su ogni nodo. Si fa quindi

l'assunzione che questi task siano distribuiti in maniera uniforme su tutti i vari executor dell'applicazione così come avviene per i mappers.

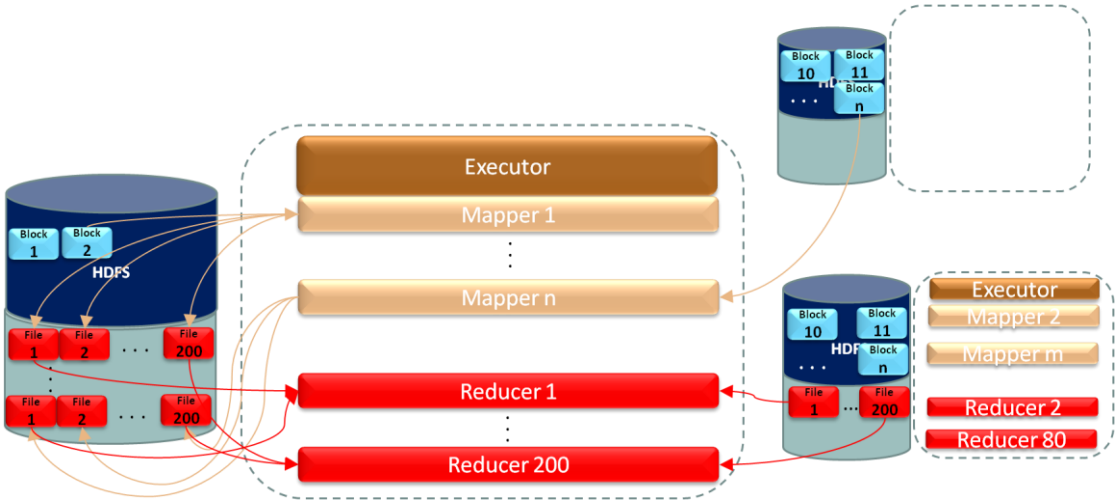

Figura 34Fetch dei dati da parte dei task reducers sia in locale che su altri nodi del rack

Considerando sempre una distribuzione uniforme dei mapper risulta quindi che i dati che si trovano già in locale e che non necessitano di essere trasferiti sulla rete siano:

$$LocalReduceSize = TotalShuffleWriteSize * \frac{1}{Exec}$$

Come al solito consideriamo la quantità di dati prelevati in locale per ogni executor:

$$LocalReduceSizePerExecutor = \frac{LocalReduceSize}{Exec}$$

Su tale porzione di dati sarà necessario applicare solamente il tempo di lettura degli stessi dal disco locale con il solito parametro $DiskSpeed$ e il corrispettivo overloading:

$$LocalReduceFetchTime = \frac{LocalReduceSizePerExec * DiskOverloading}{DiskSpeed * VCore}$$

Questo permette di avere un'ulteriore diminuzione dei dati che vengono inviati sulla rete ottimizzando in parte i tempi. Ora bisogna però considerare i dati che vengono invece richiesti su altri nodi. Per semplicità verranno considerati solo nodi che si trovano sullo stesso rack. Chiaramente questa quantità è data dalla differenza della quantità di dati totale di shuffle write e i dati in shuffle write che si trovano in locale:

$$NonLocalReduceSize = TotalShuffleWriteSize - LocalReduceSize$$

Per poi calcolare i dati richiesti da ogni singolo executor:

$$NonLocalReduceSizePerExecutor = \frac{NonLocalReduceSize}{Exec}$$

Ora è possibile considerare il tempo di fetch dei dati dal disco del nodo remoto a cui sono stati richiesti:

$$NonLocalReduceDiskTimePerExecutor = \frac{NonLocalReduceSizePerExecutor * ExternalDiskOverloading}{DiskSpeed * Vcore}$$

Questi dati devono poi essere trasferiti sulla rete e quindi è necessario considerare un tempo di trasferimento dato da:

$$NonLocalReduceTransferTimePerExecutor = \frac{NonLocalReduceSizePerExecutor * NetworkOverloading}{IntraRackTransferSpeed}$$

Ora è possibile calcolare dapprima il tempo di fetch dei dati che non si trovano in locale che come nel caso della lettura dei dati sarà dato dal massimo tra il tempo di lettura dei dati dal disco del nodo remoto e il loro invio:

$$NonLocalReduceFetchTime = MAX(NonLocalReduceDiskTimePerExecutor; NonLocalReduceTransferTimePerExecutor)$$

Per poi ottenere dalla somma dei tempi di fetch dei dati locali e non locali il tempo totale di fetch dei dati per i reducer:

$$TotalReduceFetchTime = NonLocalReduceFetchTimePerExecutor + LocalReduceFetchTime$$

Una volta che il task reducer ha ottenuto i suoi dati in input può procedere alla elaborazione del join ma come detto il tempo di elaborazione della cpu non viene considerato in questo modello.

3.4.3 Group By

Altra componente fondamentale di una GPSJ è la fase di group by nella quale i dati con lo stesso valore per l'attributo specificato vengono aggregati. In particolare in Spark anche

questa fase può risultare particolarmente costosa a causa della necessità di fare shuffeling dei dati così come avviene nel caso del Shuffle Hash Join.

In questo caso però bisogna considerare che nella fase di map lo shuffle write avviene in maniera leggermente diversa. Restano comunque valide le regole che riguardano la creazione dei file e l'utilizzo dei parametri ad esse riferiti ma in questo caso si deve considerare che per ogni gruppo di dati verrà salvata la chiave corrispettiva una sola volta e quindi non un attributo per ogni record così come avviene nel caso del shuffle join. In più nel caso siano da calcolare valori aggregati su altri attributi tali valori saranno già calcolati sulla partizione del singolo task di modo che solo il risultato finale sarà inviato sulla rete invece di dover inviare tutti i valori per ogni record diminuendo la quantità di dati che devono essere prima scritti in shuffle write e poi inviati sulla rete ai reducer. In Figura 35 un esempio di come cambia rispetto alla shuffle join la shuffle write avendo eseguito sul task oltre all'operazione di filtro anche quella di group by. Nello specifico la query eseguita sarebbe del tipo: SELECT AttributoGroupBy,MAX(AttributoSelezionato) FROM Table WHERE AttributoSelezionato <> 23 GROUP BY AttributoGroupBy. Si consideri che in questo semplice esempio per ogni reducer troviamo una sola tupla ma, soprattutto nei casi in cui l'attributo di group by abbia una cardinalità molto grande, potremmo trovare altre tuple per lo stesso reducer con un differente valore di attributo ma che la funzione di hash ha comunque assegnato a lui.

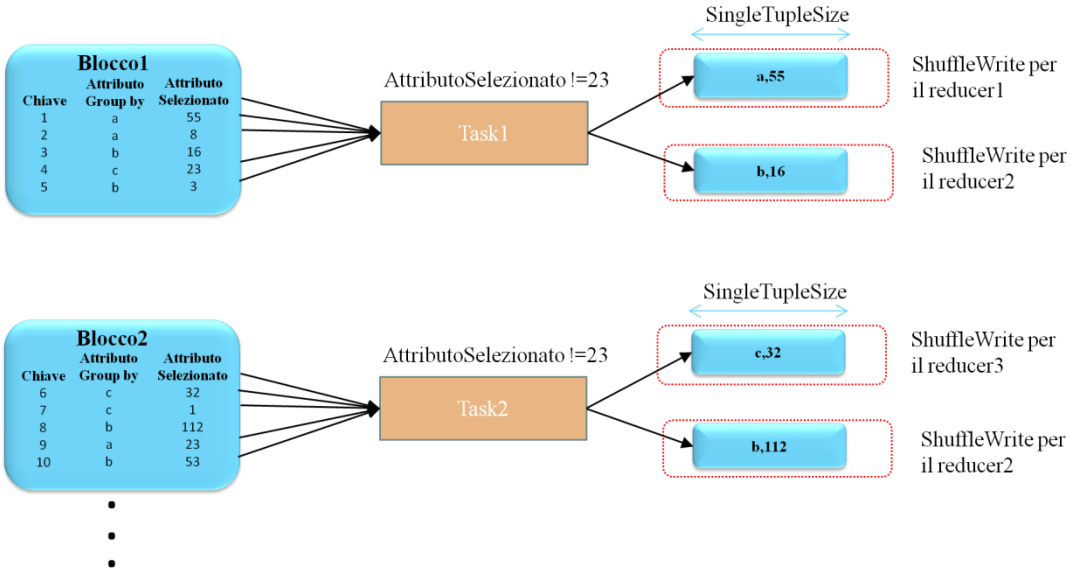

Figura 35 Esempio lettura di un blocco e scrittura dei dati in shuffle write nel caso di un'operazione di group by da parte di un task. In questo caso viene richiesto il valore massimo dell'attributo selezionato

Per comprendere quindi la quantità di dati che interessano questa fase si è deciso come nel caso del Shuffle Hash Join di stimare tale quantità in maniera puntuale valutando la dimensione delle tuple scritte in shuffle write e la loro dimensione. Per la dimensione della singola tupla si può utilizzare la stessa formula derivata dalla fase di Shuffle Hash Join ($SingleTupleSize$) ma per ottenere il valore totale dei dati scritti in questo caso non è sufficiente un semplice prodotto tra tale valore e il numero di tuple ottenute dopo il filtro a causa dell'aggregazione dei dati. Risulta quindi necessario fare una valutazione a livello di singolo blocco di dati in input che vengono elaborati da un task. Come prima operazione è necessario ottenere il numero medio di tuple presenti in ogni blocco:

$$TuplePerBlock = \frac{Card(Table) * Filter}{TableBlock}$$

Dove filter indica la percentuale di filtro applicata alla query. $TuplePerBlock$ risulta quindi essere il numero totale di tuple che ogni task dovrà elaborare e sulle quali eseguirà una prima operazione di group by e valutazione degli attributi aggregati. Per valutare il numero esatto di tuple che dovranno essere scritte si deve allora considerare il numero di valori distinti dell'attributo di group by che sono presenti nell'insieme di quelle trattate dal task. Per far questo è stata utilizzata la formula di Cardenas utilizzata nei sistemi informativi per effettuare una valutazione del costo di accesso di un RDBMS nella ricerca di un record su differenti blocchi. Nel nostro caso tale formula permette di dare una stima sul numero di tuple che il task dovrà andare a scrivere:

$$GroupByTupleNumber = AttributeDistinctValue * \left(1 - \left(1 - \frac{1}{AttributeDistinctValue}\right)^{TuplePerBlock}\right)$$

Dove il valore $AttributeDistinctValue$ indica il numero di valori distinti (su tutta la tabella) per l'attributo di group by. Si può osservare come logicamente il numero di tuple ottenute risulti essere molto vicino a $AttributeDistinctValue$ nel caso in cui la cardinalità di un blocco risulti essere molto maggiore rispetto a $AttributeDistinctValue$ dato che risulta maggiormente probabile che tra tutte le tuple presenti almeno una faccia parte di tale gruppo, mentre allo stesso modo se la cardinalità del blocco risulta essere molto minore di $AttributeDistinctValue$ il numero di tuple ottenute si avvicinerà maggiormente alla cardinalità del blocco. Va comunque considerato che tra le ipotesi di base della formula di Cardenas si considerano i vari record equiprobabili, questo si traduce nel nostro caso con la necessita di avere una distribuzione uniforme dei valori dell'attributo di group by per poter

ottenere dei buoni risultati. Ottenuto il numero effettivo di tuple scritte da ogni blocco possiamo ora procedere al calcolo della quantità di dati scritti considerando di moltiplicare per il numero di blocchi da cui è costituita la tabella dato che abbiamo considerato le tuple elaborate da ogni singolo task:

$$GroupByShuffleWrite = GroupByTupleNumber * SingleTupleSize * TableBlock$$

Come al solito tale valore si riferisce a tutti i dati scritti a livello di cluster, quindi per ottenere la quantità di dati scritti dal singolo executor dobbiamo dividere tale valore per il numero di executor utilizzati:

$$GroupByShuffleWritePerExecutor = \frac{GroupByShuffleWrite}{Exec}$$

Essendo anche questa un'operazione di shuffle sappiamo che tali dati andranno scritti su disco quindi possiamo fare le stesse considerazioni fatte nelle sezioni precedenti ottenendo il tempo di scrittura di tali dati da parte di un singolo executor:

$$GroupByShuffleWriteTime =$$

$$\frac{GroupByShuffleWritePerExecutor * DiskOverloading}{DiskSpeed * VCore}$$

Ora che abbiamo ottenuto il tempo per il shuffle write dei dati è possibile considerare la fase di reduce. Valgono in questo caso tutte le formule che sono già state definite in fase di shuffle join per quanto riguarda il fetch dei dati da parte dei task reducer essendo anche questa un'operazione di shuffle. In fase di reduce l'unica differenza sarà data dal fatto che il task in fase di elaborazione dei dati non dovrà eseguire un'operazione di join ma un'operazione di group by sui singoli sottogruppi che riceverà, ma come detto in precedenza il tempo cpu non viene considerato nella formulazione del modello

4 Risultati sperimentali

4.1 Strumenti

Per la realizzazione dei test sono stati necessari diversi strumenti che hanno permesso l'esecuzione delle applicazioni Spark e la loro diagnostica di modo che fosse possibile ottenere i dati necessari per fare un confronto con i dati ottenuti dalle formule realizzate nel capitolo precedente.

4.1.1 Cluster

Tutte le applicazioni sono state testate sul cluster messo a disposizione dall'università. Su di esso girano diversi servizi e framework tra i quali in particolare Yarn, Spark, Hive e HDFS che sono stati sfruttati prima di tutto per comprendere il comportamento del sistema e poi in secondo luogo per eseguire i test. Il cluster è costituito da 7 nodi di commodity hardware forniti di due dischi fissi da 2 Terabyte,una CPU Intel i7-4790 con 8Vcore da 3.6 Ghz, una RAM da 32 Gigabyte e una connessione punto a punto da 1 Gigabit tra i nodi. Tutti i nodi si trovano all'interno dello stesso rack e il valore di ridondanza dei blocchi HDFS è quello di default (3). Il sistema operativo è un CentOS 6.6 (Linux).

4.1.2 Spark ui

Ogni applicazione Spark che viene eseguita sul cluster genera una certa quantità di dati di log che vengono salvati in un apposita directory specificata dal parametro *spark.eventLog.dir*. In questa directory si possono trovare tutti i log di tutte le applicazioni eseguite su Spark. Per una valutazione puntuale dei dati di esecuzione tali file possono essere processati (sono in formato JSON) ed analizzati. Per una visualizzazione più chiara e veloce però Spark mette a disposizione un'interfaccia grafica chiamata Spark UI che mostra a video sfruttando un browser tutti i dati di log relativi ad una specifica applicazione:

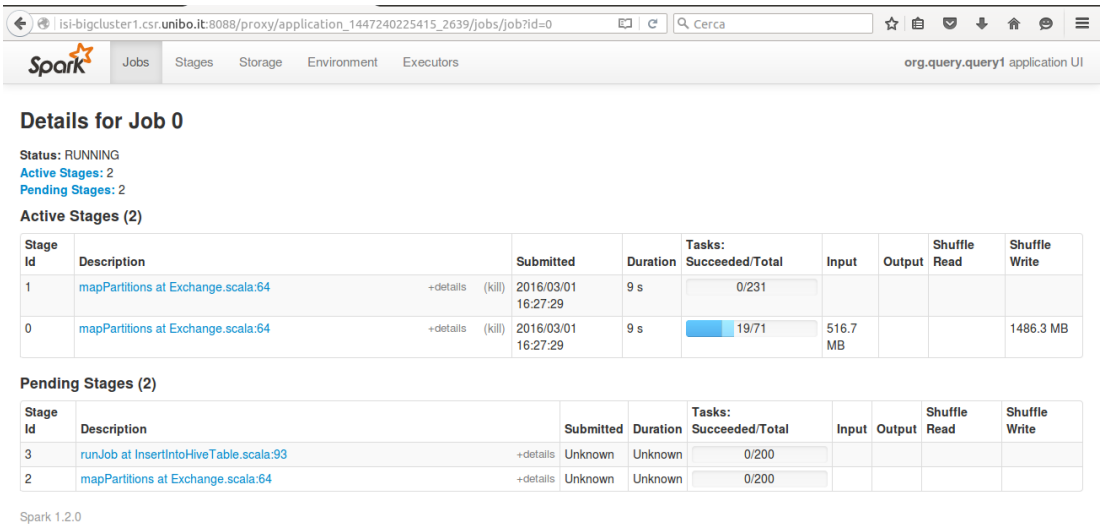

Figura 36 Interfaccia grafica Spark UI

Come si può osservare in Figura 36 Interfaccia grafica Spark UIviene data la possibilità all'utente di visualizzare numerose informazioni fondamentali riguardanti l'esecuzione dell'applicazione. Prima di tutto permette di vedere i job completati, in esecuzione o ancora da eseguire (dato che il piano fisico è già stato formulato da Spark) e i loro tempi. In più è possibile scendere ulteriormente nel dettaglio osservando i tempi degli stage fino ad arrivare al livello dei task con informazioni che riguardano anche le quantità di dati che sono stati letti in input oppure scritti per esempio in shuffle write. Nelle altre schede sono anche presenti dati aggregati che riguardano gli executor (dati scritti e letti, numero di task eseguiti, memoria assegnata ecc ecc) oppure dati che si riferiscono all'ambiente in cui viene eseguita l'applicazione (es. valori di parametri).

4.1.3 Cloudera manager

Claudera manager è un'applicazione end-to-end per la gestione di un cluster CDH (Claudera Distribution including apache Hadoop). Attraverso Claudera manager è possibile gestire in maniera semplice e centralizzata tutto lo stack del CDH e diversi altri servizi inoltre, grazie a diversi strumenti di diagnostica permette anche di avere una visione real time dello stato di tutto il cluster sia a livello di singoli host sia ad un livello aggregato. Il cuore di questa applicazione è il ClouderaManagerServer nel quale si trova l'admin console web server e la logica applicativa. Esso è responsabile dell'installazione del software, delle fasi di configurazione e avvio dei servizi e del loro stop. Esso interagisce con diversi altri componenti come gli agenti installati su ogni host che gli permettono di monitorarne le

performance, i database dove vengono memorizzati tutti i dati di configurazione e di monitoring dei servizi e i client che mettono a disposizione dell'utente finale un'interfaccia web. Come si può osservare in Figura 37 l'interfaccia web permette (oltre a visualizzare lo stato dei vari servizi in esecuzione) anche di osservare il comportamento del cluster e delle sue risorse attraverso diversi grafici che riguardano l'utilizzo della rete, dei dischi, della cpu e altro ancora. Attraverso tali grafici è stato possibile avere un'idea di massima sul comportamento di una specifica applicazione in particolare su come le risorse assegnatele siano state sfruttate.

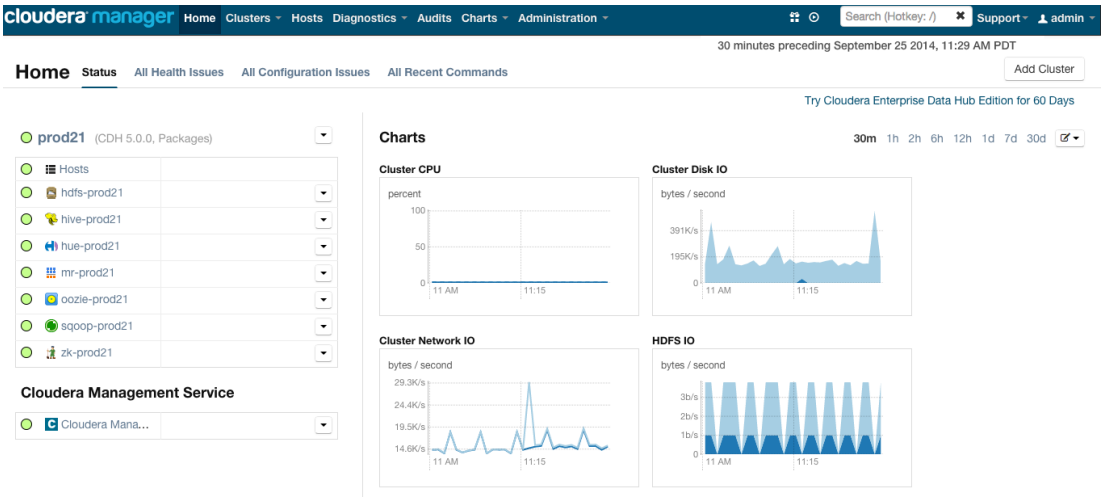

Figura 37 Interfaccia web di Claudera Manager

4.1.4 Programmazione dell'applicazione Spark

Spark risulta essere scritto in Scala ma permette comunque di eseguire su di esso applicazioni Java e Python. Si è scelto quindi di realizzare un'applicazione Java. Per poter realizzare l'applicazione in locale e avviarla poi in remoto sul cluster si è realizzato un progetto Maven che ha permesso di specificare le dipendenze anche di librerie (ad esempio quelle relative a Spark e Hive) che la nostra applicazione avrebbe trovato solo una volta eseguita sul cluster. Come per ogni applicazione Spark si deve prima di tutto inizializzare uno SparkConf, un oggetto che viene utilizzato per definire tutte le configurazioni dei parametri con cui l'applicazione deve essere eseguita. Per eseguire le query scrivendo direttamente dei comandi Hive è stato utilizzato uno JavaHiveContext, un wrapper del JavaSparkContext (il quale permette a Spark di avere le informazioni necessarie per riuscire a connettersi al cluster). Con pochissime righe di codice è quindi possibile lanciare sul cluster una query hive di qualsiasi tipo. Una volta scritta la query il progetto Maven può

essere esportato come jar e quindi inviato sul cluster di modo che la nostra classe sia riconosciuta e lanciata.

4.1.5 Realizzazione tabelle

Per effettuare i test si è deciso di realizzare delle tabelle apposite con le caratteristiche necessarie all'analisi che si è deciso di effettuare. Considerando quindi delle query GPSJ risulta necessario avere a disposizione una fact table e una dimension table su cui effettuare un join e un group by. Sono state quindi realizzate sul cluster due tabelle apposite su di un database utilizzato in fase di test. Dato che l'ambito è quello dei big data si è voluto realizzare due tabelle di dimensioni piuttosto elevate sia per quanto riguarda il numero di tuple sia per quanto riguarda l'effettiva dimensione. In oltre le considerazioni fatte nella formulazione del modello sono state considerate basandosi su tabelle in formato colonnare, quindi le tabelle realizzate sono in formato parquet.

Per quanto riguarda la fact table sono stati considerati 3 campi chiave che rappresenterebbero le chiavi importate dalle dimension table ad essa collegata e che insieme ne costituiscono una chiave univoca per ogni fatto presente sulla tabella. Per semplicità le prime due chiavi sono due semplici progressivi che partono da 1 fino al numero di tuple presenti sulla tabella. La terza chiave invece è una chiave importata dalla dimension table realizzata appositamente. Il valore di tale attributo viene quindi valorizzato in maniera casuale uniforme all'interno del range di valori della chiave della dimension table. Infine come ultimo attributo consideriamo una misura che rappresenta l'effettiva misura calcolata in maniera casuale come un valore double da 0 a 1 milione. La struttura della fact table risulta quindi essere quella in Figura 38.

La dimension table è stata realizzata come detto con un attributo chiave anche in questo caso incrementale da 1 al numero di tuple della tabella. I successivi attributi sono dei campi stringa con un numero fisso di caratteri ma con un numero di valori distinti crescente (Attributo0 1 solo valore distinto, Attributo1 10 valori distinti ecc ecc) questo per permettere di avere diversi valori di aggregazione in una query GPSJ. I valori di ogni attributo sono quindi stati selezionati in maniera casuale uniforme tra tutti i possibili valori distinti per quello specifico attributo. La struttura della dimension table risulta quindi essere la seguente:

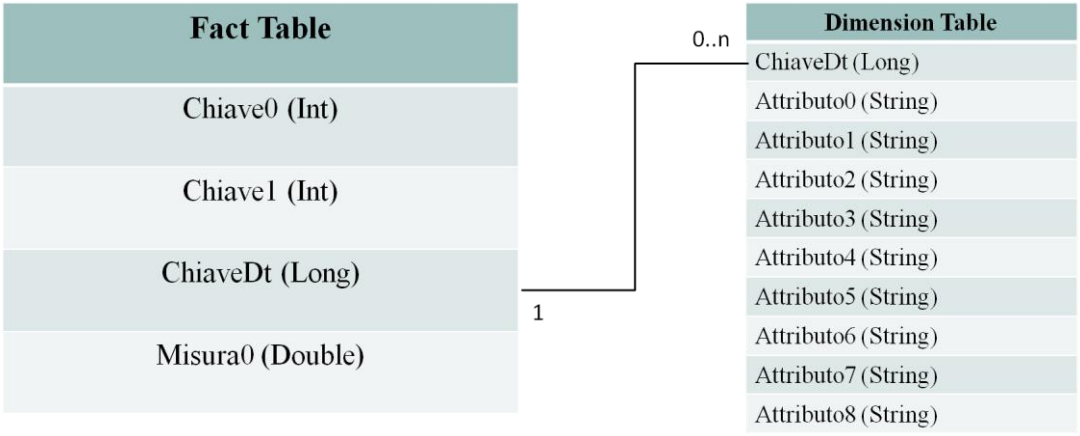

Figura 38 Schema delle tabelle utilizzate

Nel caso specifico il numero di tuple inserite nella fact table sono 1 miliardo andando ad occupare uno spazio totale di 22,35 Gigabyte suddivisi in 231 blocchi su HDFS e quindi sparsi su tutti i nodi del cluster (il tutto senza considerare il fattore di replicazione). Il numero di tuple inserite invece nella dimension table è pari a 100 milioni per un totale di 6,5 Gigabyte di spazio su HDFS con 71 blocchi. Tali tabelle sono state inserite direttamente su HDFS utilizzando un apposita applicazione Spark scritta ad hoc. Effettuando una semplice insert attraverso l'applicazione Spark, a causa dell'elevato numero di task eseguiti il numero di blocchi realizzati risultava essere troppo elevato (nell'ordine delle migliaia) andando a inficiare i risultati dei test con un valore di fetch dei file troppo elevato. Per questo è stato necessario reinserire i dati sfruttando Hive il quale facendo il merge di tali file cerca di saturare il più possibile lo spazio di un blocco (128 Megabyte).

4.2 Risultati sui volumi di dati

Come precisato nel capitolo precedente il modello si focalizza principalmente sui tempi che riguardano la lettura da disco e l'invio di dati sulla rete. Risulta prima di tutto fondamentale testare tali quantità di dati. Sono presenti 3 principali fasi in cui risulta necessario valutare tali quantità:

- dati letti dal disco
- dati scritti in shuffle write
- dati letti in shuffle read

In tutti i casi i dati a cui si fa riferimento sono valori aggregati cioè la somma dei dati letti o scritti da tutti i task di tutti gli executor. Per quanto riguarda i parametri di Spark sono stati specificati in questo modo:

- *spark.driver.memory* = 15Gb
- *spark.executor.memory* = 20Gb
- *spark.shuffle.compress* = "False"

I valori di memoria del driver e degli executor sono stati impostati vicino al valore massimo che poteva essere utilizzato per diminuire la possibilità di incorrere in fail dei task. Per quanto riguarda invece la compressione è stata impostata a false per far si che le valutazioni possano essere più precise non dovendo considerare un fattore di compressione che può dipendere dal tipo di dati inviati.

Come detto precedentemente la valutazione degli RDD in Spark è lazy e quindi solo se è presente una action alla fine le richieste vengono realmente elaborate. Per questo ogni volta che si effettua una query si richiede di creare una tabella sul database di modo che tale richiesta forzi l'elaborazione dell'RDD (Create table tmp as select ….) per andare a scrivere il risultato della query su di una tabella temporanea.

Per quanto riguarda invece il calcolo della dimensione delle tuple ci si è basati su dati ricavati da diversi test:

- FixedTupleSize: 211 nel caso di shuffle join, 40 nel caso di group by
- FixedTypeSize: 60 nel caso di shuffle join, 0 nel caso di group by

4.2.1 Dati letti dal disco

Consideriamo inizialmente la quantità di dati letti durante lo scan di una tabella (nei test è stata utilizzata la fact table) e come tali quantità variano al variare degli attributi selezionati eseguendo la seguente query:

SELECT Chiave0
FROM Ft

Partendo da questa query si sono poi aggiunti altri campi nella selezione (Chiave1,Misura0,ChiaveDt). Questo perché la tabella considerata risulta essere in formato parquet e quindi colonnare; questo ci permette di andare a leggere solamente le colonne che ci interessano e non tutte le tuple per intero permettendoci di valutare la correttezza della formula impostata precedentemente. Di seguito i risultati sperimentali dell'esecuzione di tale query:

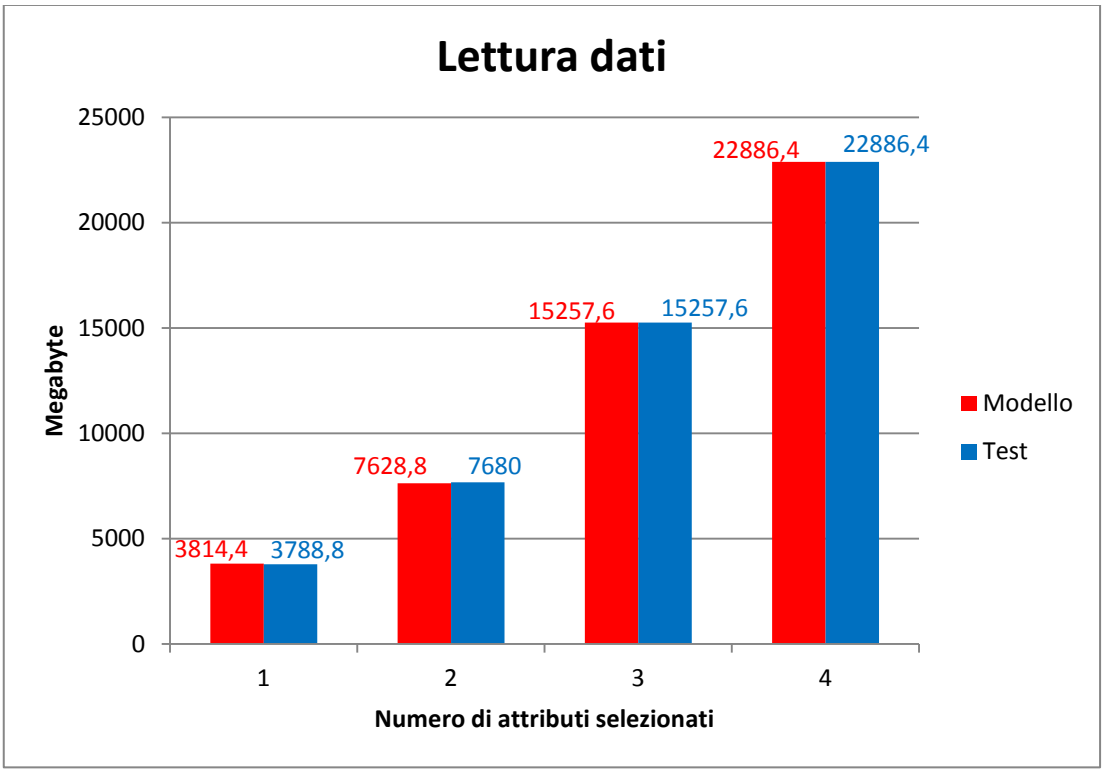

Figura 39 Confronto sulla quantità di dati letti in fase di scan

Come si può osservare i valori attesi e quelli osservati risultano essere molto simili fino ad essere ovviamente uguali nel caso in cui si prelevi tutta la tabella. Si può anche osservare come ci sia un forte sbalzo tra il caso con 2 attributi e quello con 3 (così come tra 3 e 4). Questo risulta semplicemente dovuto al fatto che sono stati aggiunti in selezione attributi di tipo double (o long nel caso 4) i quali, essendo modellati con 8 byte invece che 4 come gli int (caso dei primi due attributi), risultano avere un volume maggiore. Non si sono considerati casi con filtro dato che in ogni caso per il comportamento di Spark esso sarebbe stato valutato in una fase successiva dal task lasciando quindi invariati le quantità di dati letti.

4.2.2 Dati scritti in shuffle write

Consideriamo ora la quantità di dati che devono essere scritti in shuffle write nei casi in cui sia necessario un invio di dati sulla rete. In questo caso dobbiamo fare una distinzione tra la valutazione del join e la valutazione del group by dato che sono state fatte considerazioni differenti. Nel caso di join eseguiamo la seguente query:

SELECT f.Chiave0
FROM Ft as f, Dt as d
WHERE f.Chiave0<100000000 AND d.ChiaveDt<20000000AND f.ChaiveDt=d.ChiaveDt

E’stato anche inserito un filtro per evitare che la quantità di dati scritti in shuffle write risulti essere eccessiva a causa del mancato utilizzo della compressione, in ogni caso si è osservato che la quantità di dati scritti in shuffle write cambia in maniera direttamente proporzionale rispetto al filtro. Anche in questo caso valutiamo come si comportano i dati all’aumentare degli attributi selezionati (Chiave1,Misura0,Attributo0).

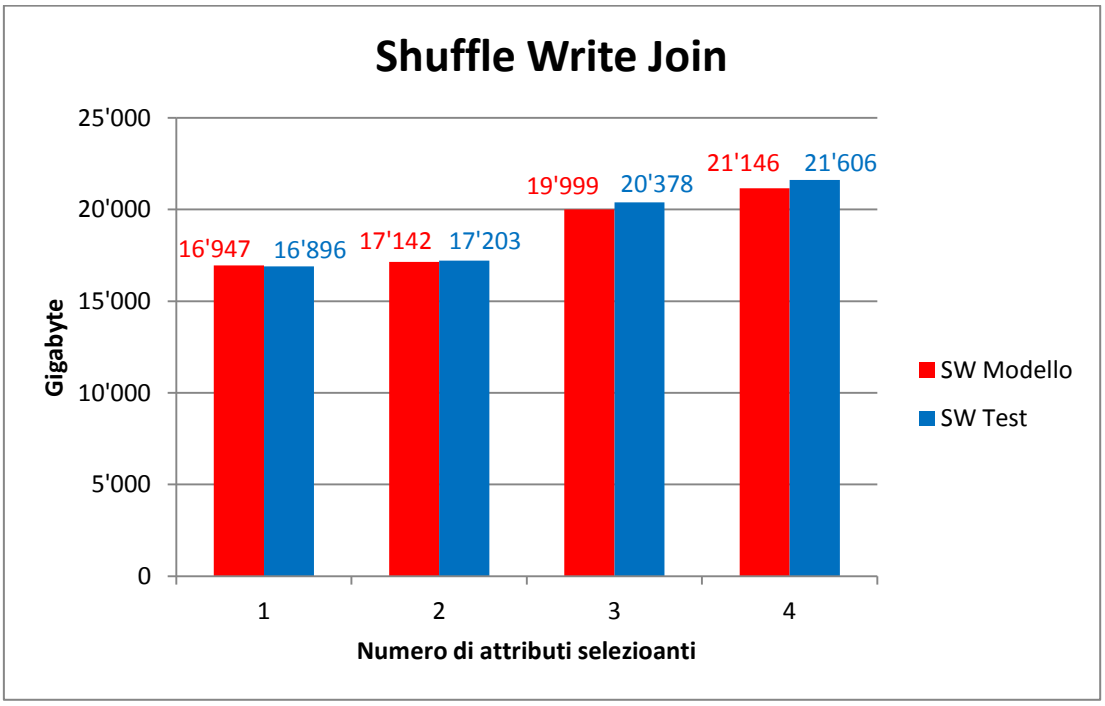

Figura 40 Confronto sulla quantità di dati scritti in shuffle write in caso di join

Anche in questo caso si può osservare come i valori del modello e quelli osservati siano molto vicini. Al contrario della lettura da disco all’aumentare degli attributi selezionati la quantità di dati non aumenta allo stesso modo ma più lentamente. Questo è dovuto al fatto che i dati scritti essendo prima serializzati non aumentano in maniera rilevante se viene aggiunto un attributo dello stesso tipo del precedente risulta essere più rilevante invece il caso in cui l’attributo aggiunto sia di tipo diverso ma partendo comunque da una base comune di FixedTupleSize piuttosto elevata risulta incidere poco sul valore finale.

Passando invece al caso del group by si deve considerare come vari la quantità di dati scritti al variare del numero di valori distinti dell’attributo di group by. I test sono stati quindi effettuati sulla dimension table nella quale sono presenti attributi con quantità di valori distinti differenti considerando la seguente query:

SELECT Attributo0
FROM dtuniforme100m
GROUP BY Attributo0

In ogni test si è quindi modificato l'attributo di group by per osservare se la quantità di dati scalasse correttamente

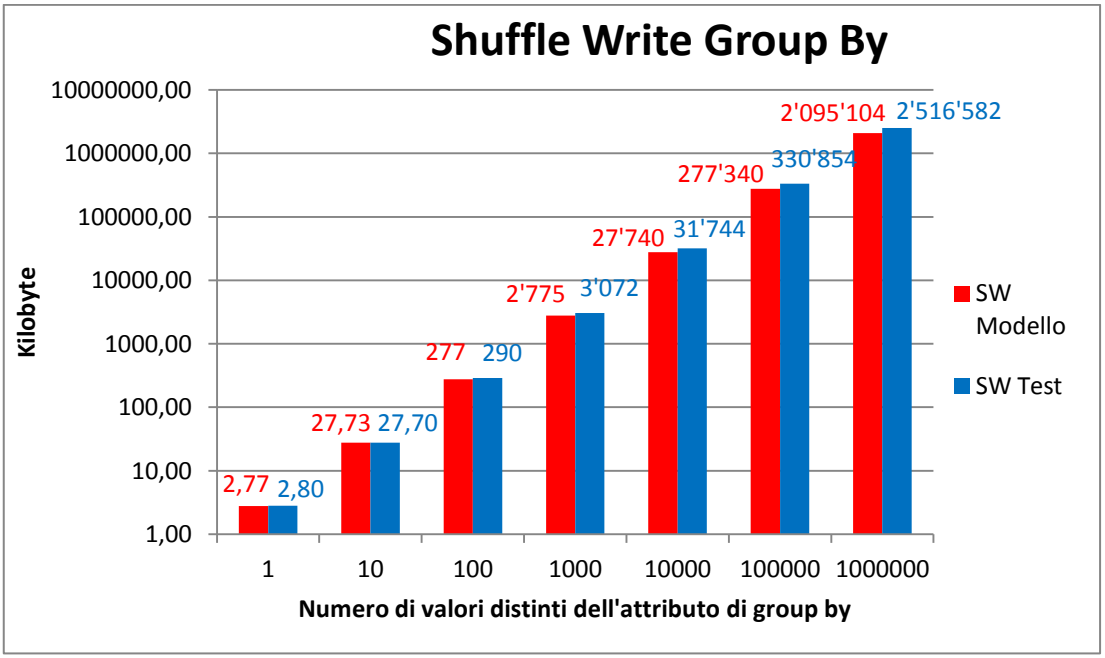

Figura 41 Confronto sulla quantità di dati scritti in shuffle write nel caso di group by

In base al numero di gruppi creati la quantità di dati può variare di molto passando da pochi Kilobyte a diversi Gigabyte per questo è stata utilizzata una scala logaritmica. In ogni caso si può osservare come i dati del modello e quelli osservati scalino allo stesso modo all'aumentare del numero di gruppi considerati.

4.2.3 Dati letti in shuffle read

Dopo che uno stage ha scritto i dati in shuffle write lo stage successivo li andrà a leggere e tale quantità sarà definita come shuffle read. Ovviamente la quantità di dati in shuffle write coincide con la quantità in shuffle read ma si deve comunque considerare che una parte viene letta in locale e una parte viene invece letta dai nodi remoti (se si utilizza più di un executor). Per confrontare tale quantità quindi si è considerata la stessa query utilizzata per la valutazione dello shuffle write nel caso del join ma eseguendola con diversi executor per osservare se all'aumentare degli executor anche la quantità di dati letti da nodi remoti aumenti. Di seguito i risultati dei test:

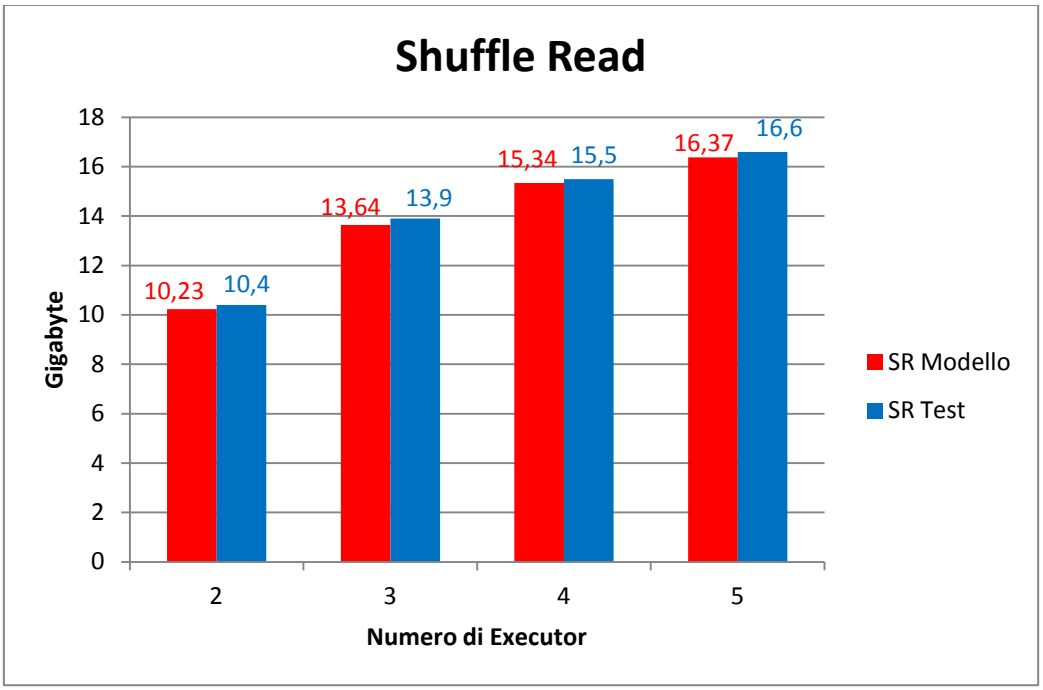

Figura 42 Confronto sulla quantità di dati letti in shuffle read utilizzando più executor

Come si può osservare le quantità di dati sperimentali e calcolati sono molto vicine. Si evidenzia un piccolo scostamento verso il basso dei valori calcolati dal modello ma questo è dovuto al fatto che la quantità di dati in shuffle write calcolata risulta essere di 20,46 Gb mentre quella sperimentale di 20,7 Gb

4.3 Risultati sui tempi di elaborazione

Per valutare la correttezza del modello realizzato sono stati eseguiti diversi test. Si sono voluti basare questi test sulla comprensione dei tempi a livello di una query GPSJ per le ragioni specificate nel capitolo precedente. Tutti i test sono stati effettuati utilizzando un numero di executor che va da 1 a 5. Si è dovuto scegliere 5 come valore massimo perché risulta essere il limite per il quale Spark riesce ad allocare un solo executor su ogni nodo all'interno del nostro cluster rendendo l'analisi più chiara. Il numero di nodi nel cluster è 7 ma si deve considerare che su uno di questi nodi viene allocato il driver mentre su un altro nodo vengono fatti girare i servizi del cluster stesso, risulta quindi che i nodi disponibili per allocare gli executor siano 5. Altra dimensione su cui si sono sviluppati i test riguarda invece il numero di VCore assegnati ad ogni executor per capire come riesce a scalare il sistema condividendo determinate risorse come il disco la ram. Le letture e scritture su disco sono state valutate con un valore di 100Mb/s in tutti i casi (locali e remoti) considerando un valore di overloading basato su dati sperimentali. Si è anche considerato un valore di overloading

della rete pari a 1 dato che non risulta possibile avere valori reali da poter applicare e in più la connessione tra i nodi risulta essere punto a punto e talmente veloce che non dovrebbe incidere in maniera significativa sui tempi. Per quanto riguarda i parametri di Spark sono gli stessi che sono stati utilizzati per la valutazione dei volumi di dati così come l'assenza di compressione dei dati di shuffle.
Come detto la prima fase riguarda la lettura dei dati che si trovano in HDFS, si è quindi deciso di effettuare alcuni test sulla semplice valutazione dei tempi a livello di fetch dei dati per avere dimostrazione che le formule proposte modellino correttamente il comportamento del sistema in questa fase.

4.3.1 Lettura dei dati e shuffle write del risultato

La prima fase in cui vengono letti i dati è caratterizzata da due stage che per l'appunto prelevano i dati in input ed eseguono una shuffle write per preparare i dati per lo stage successivo in cui avverrà il join. In oltre questi due stage non avendo nessuno scambio di dati tra loro risultano indipendenti e quindi vengono eseguite dal sistema in parallelo, per questo si è deciso di valutare in maniera puntuale questa prima fase su di una singola tabella per poi riportare tali considerazioni al caso della GPSJ. Si è considerata quindi una semplice query con la quale si effettua il fetch di 3 dei 4 attributi della fact table andando però a prelevare tutte le tuple presenti:

```
SELECT Chiave0,Chiavedt,Misura0
FROM Ft
WHERE Chiave0<1
```

Dato che il tempo di scrittura di tutte le tuple potrebbe risultare piuttosto alto dato l'elevato numero di dati, si è posto un filtro che impedisca di scrivere tuple su tale tabella di modo che il tempo finale osservato sia il più possibile vicino al valore reale del tempo necessario per effettuare la lettura della tabella. Le tuple devono comunque essere tutte lette per applicare il filtro però nessuna di esse verrà scritta in tabella di modo che i tempi che otterremo riguarderanno solamente il fetch dei dati. Nella formulazione dei risultati del modello sono ovviamente state considerate tutte le caratteristiche della tabella e la sua disposizione sul cluster (blocchi in locale e remoti).
Di seguito i risultati dei test:

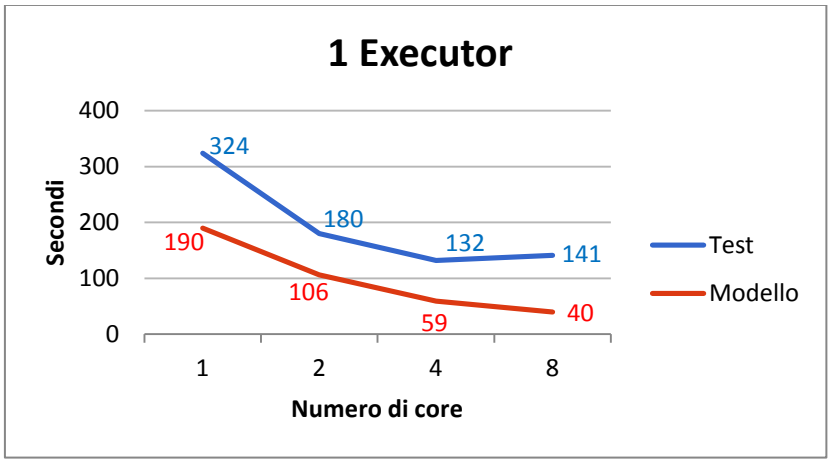

Figura 43 Esecuzione della query di lettura dei dati utilizzando 1 executor

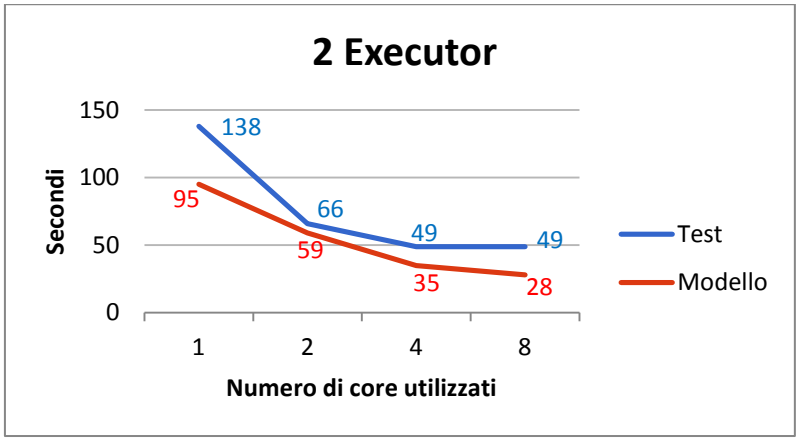

Figura 44 Esecuzione della query di lettura dei dati utilizzando executor

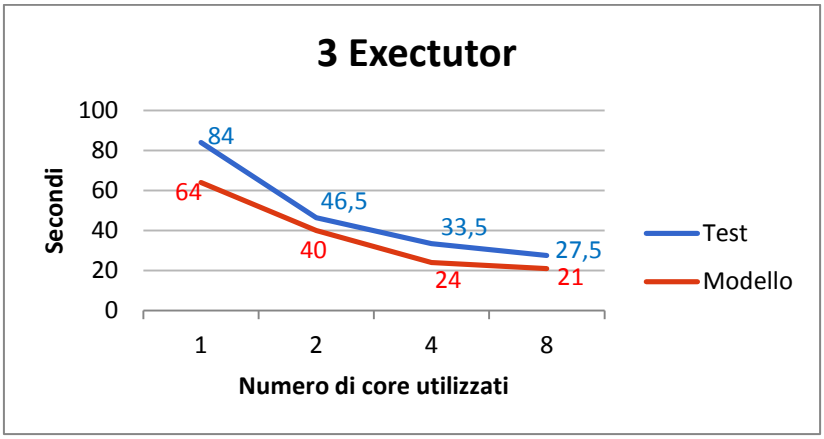

Figura 45 Esecuzione della query di lettura dei dati utilizzando 3 executor

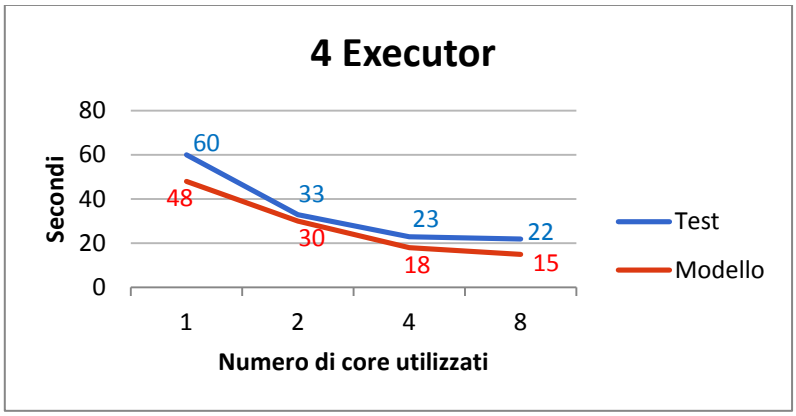

Figura 46 Esecuzione della query di lettura dei dati utilizzando 4 executor

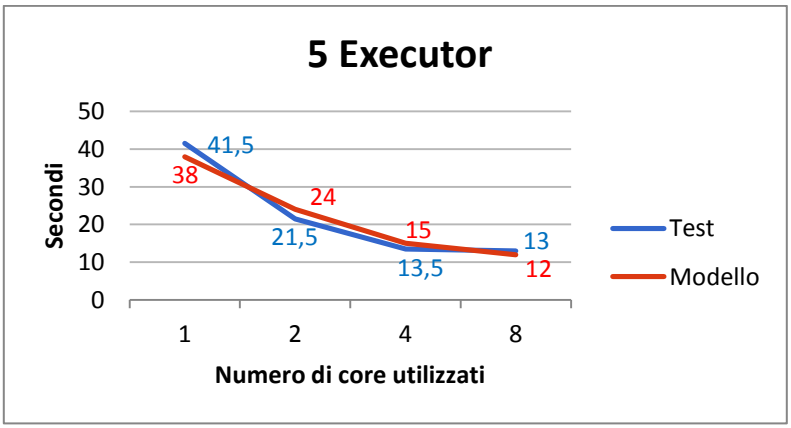

Figura 47 Esecuzione della query di lettura dei dati utilizzando 5 executor

Prima di tutto si può osservare dai grafici come all'aumentare del numero di core assegnati ad ogni executor il miglioramento tendi a degradare. Questo comportamento è anche dato dall'overloading che caratterizza il disco di ogni nodo su cui risiede un executor che naturalmente aumenta all'aumentare dei core (e quindi delle richieste) che gli vengono recapitate. Altro elemento che si può notare è lo scostamento tra il modello e i dati sperimentali che si evidenzia nel caso di utilizzo di un solo executor. Tale scostamento risulta probabilmente dovuto al fatto che per semplicità si è considerato la velocità con cui vengono ottenuti i dati dei blocchi remoti dipenda solo dalla velocità del disco remoto (impostata nei calcoli a 100 Mb/s) mentre in realtà risulta essere minore e questo probabilmente perché non vengono considerate tutte le operazioni di preparazione dei dati per il loro trasferimento sulla rete per la loro ricezione e tutte le richieste che devono essere effettuate tra i due nodi per far arrivare tali dati a destinazione.

Come detto nel primo stage si ha anche una componente di shuffle write, si è quindi realizzato un secondo test che vada a considerare anche tale tempo. Per tale test è stata eseguita la seguente query:

SELECT MAX(Chiave0) as maxc,MIN(Chiavedt),misura0
FROM Ft
WHERE Chiave0<100000000
GROUP BY misura0
HAVING maxc<1

Eseguendo tale query costringiamo Spark ad eseguire due stage dato che la presenza di un group by impone lo shuffe dei dati e di conseguenza nel primo stage di lettura dei dati avremo anche una componente che riguarda la scrittura dei dati in shuffle write per lo stage successivo. Il filtro è stato impostato per considerare solo un decimo delle tuple presenti nella tabella altrimenti, dato che non viene utilizzata la compressione nella scrittura dei dati in shuffle write, la quantità di dati sarebbe eccessiva e rischierebbe di causare dei fail nell'esecuzione dei task. E'stato aggiunto anche un ulteriore filtro a livello di gruppo per far si che le tuple realmente scritte siano zero, per lo stesso motivo specificato nella fase di lettura dei dati. Il group by è stato eseguito sull'attributo Misura0 di modo che il numero di gruppi fosse sufficientemente elevato (intorno ai 100 milioni) per avere una quantità di dati scritti in shuffle write rilevante. Di seguito i risultati dei test che si riferiscono al solo primo stage quindi alla lettura dei dati più la loro scrittura in shuffle write:

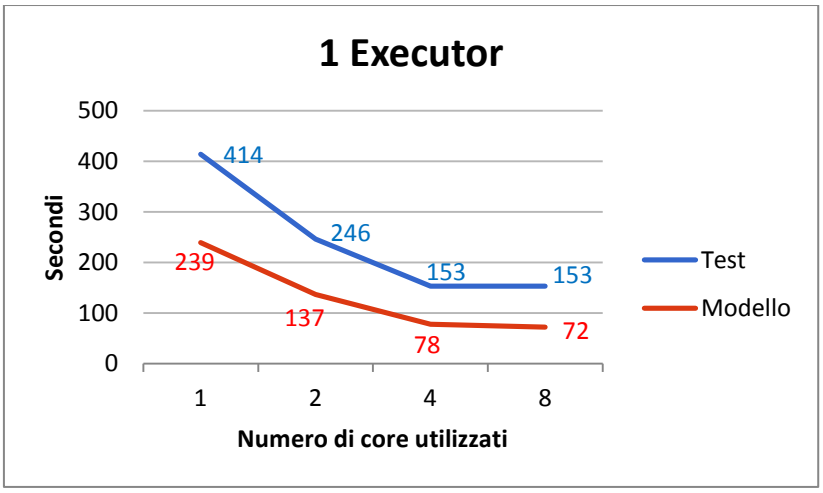

Figura 48 Tempi di esecuzione del primo stage della query con group by utilizzando 1 executor

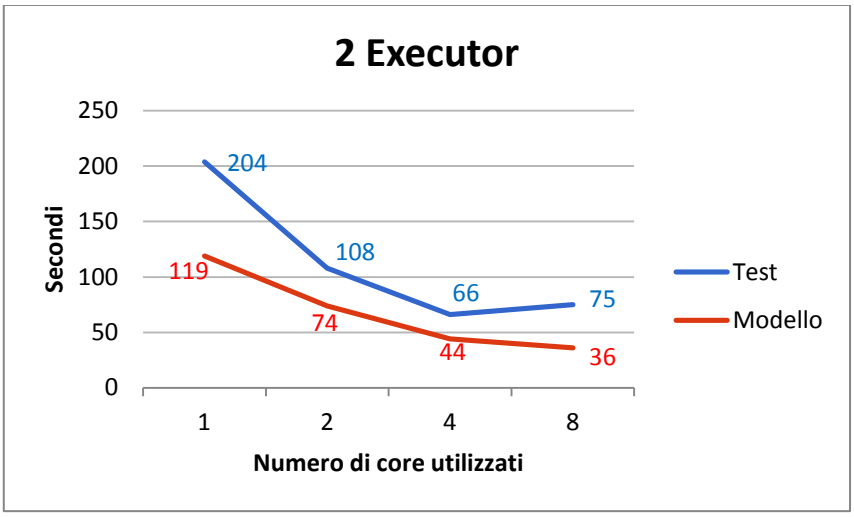

Figura 49 Tempi di esecuzione del primo stage della query con group by utilizzando 2 executor

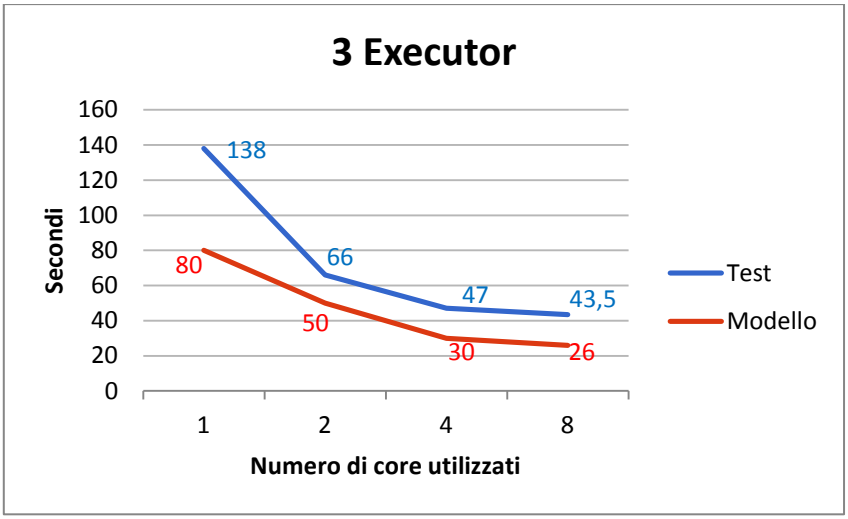

Figura 50 Tempi di esecuzione del primo stage della query con group by utilizzando 3 executor

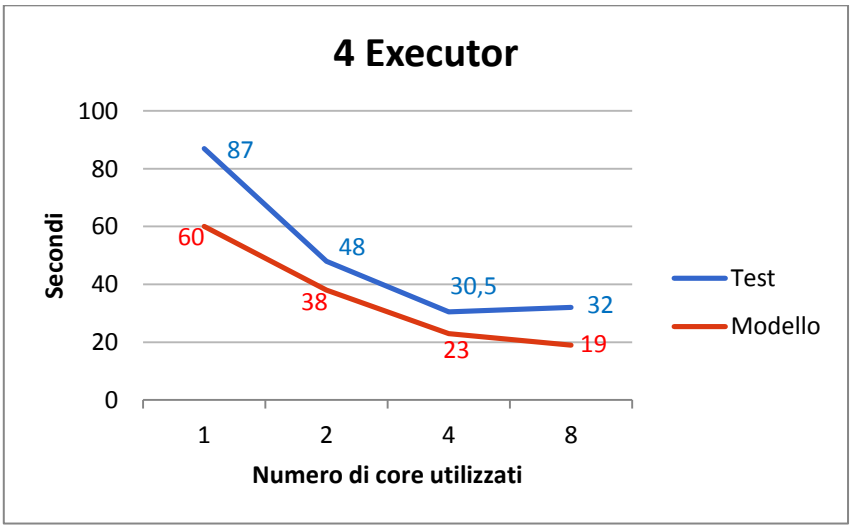

Figura 51 Tempi di esecuzione del primo stage della query con group by utilizzando 4 executor

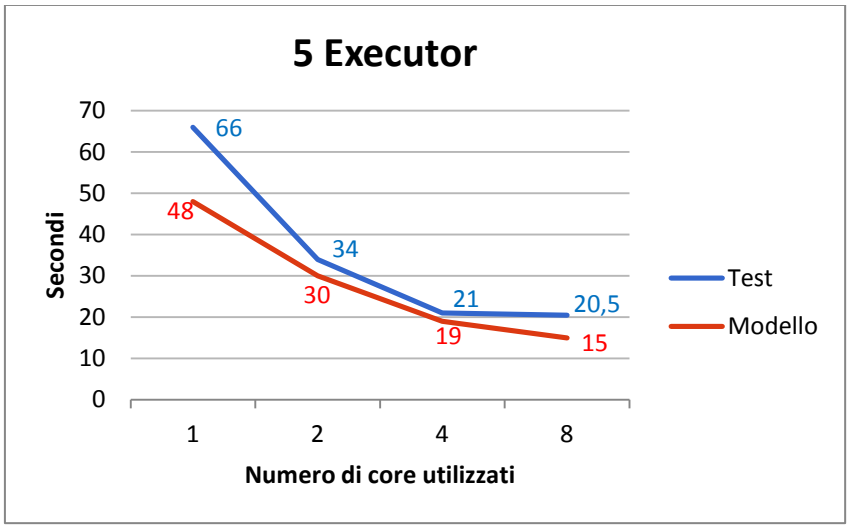

Figura 52 Tempi di esecuzione del primo stage della query con group by utilizzando 5 executor

Prima di tutto si può osservare come i tempi siano ovviamente aumentati rispetto ai primi test in cui veniva effettuato un semplice scan a causa del tempo necessario per scrivere i dati in shuffle write. In ogni caso l'andamento del modello e dei test risulta molto simile al di fuori di una leggera sottostima.

4.3.2 Query GPSJ

Come ultimo test procediamo alla valutazione di una query GPSJ fatta nel seguente modo:

```
SELECT d.Attributo5,MAX(f.Chiave0),MIN(f.Misura0)
 FROM Ft as f, Dt as d
WHERE f.Chiave0<50000000 AND d.Chiavedt<20000000 AND f.Chiavedt=d.Chiavedt
GROUP BY d.Atttributo5
```

Anche in questo caso utilizziamo un filtro piuttosto restrittivo per evitare di avere una quantità di dati eccessiva nelle fasi di scrittura e lettura che possano andare a causare dei fallimenti nei task. Viene effettuato un group by sull'Attributo5 della dimension table per avere un numero di gruppi elevato (intorno ai 100 mila). Come specificato nel capitolo precedente per l'esecuzione del job relativo ad una query GPSJ sono necessari 4 stage: i primi due dove si leggono i dati e li si preparano per essere inviati in shuffle per eseguire la query, un terzo per leggere i dati in shuffle eseguire la query, il primo group by parziale sui dati e scrivere in shuffle write i risultati ed infine un ultimo stage per l'esecuzione finale del group by e la scrittura del risultato come tabella Hive. Nelle formule utilizzate per stimare questi tempi è stato considerato un overloading nel terzo stage pari al numero di core. Questo si è potuto verificare anche sperimentalmente osservando come i tempi di uno stage che

esegue solamente una shuffle read e una shuffle write dipenda principalmente dal numero di executor utilizzati senza scalare in maniera rilevante all'aumentare dei core utilizzati. Questo risulta essere probabilmente causato dall'inteso lavoro che deve essere fatto sul disco locale dove vengono salvati e letti i dati di shuffle. Per quanto riguarda invece il calcolo della dimensione delle tuple sono stati utilizzati gli stessi dati usati nella fase di valutazione delle quantità di dati

Di seguito i risultati dei test confrontati con il modello proposto:

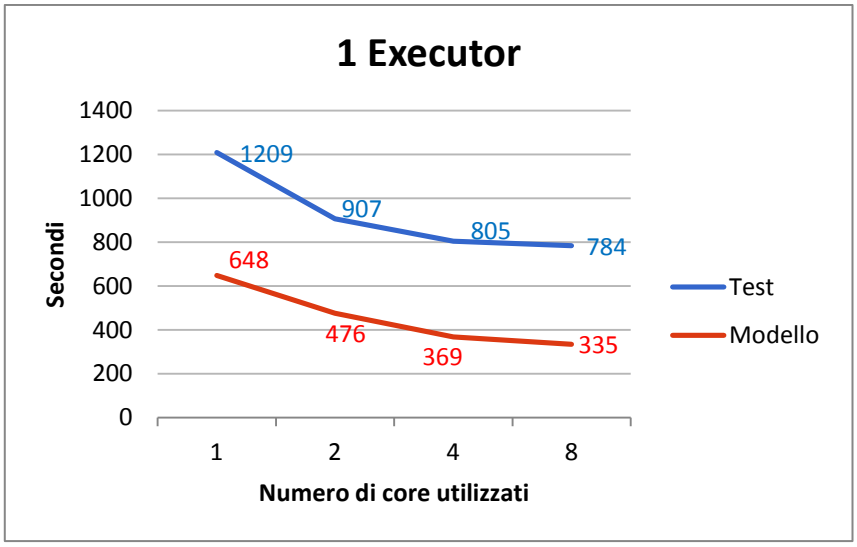

Figura 53 Tempi di esecuzione e stime di una query GPSJ utilizzando 1 executor

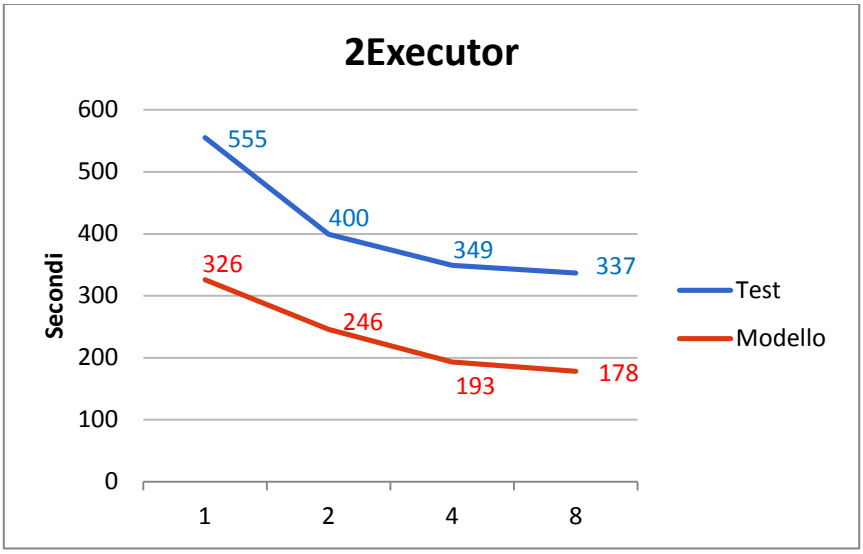

Figura 54 Tempi di esecuzione e stime di una query GPSJ utilizzando 2 executor

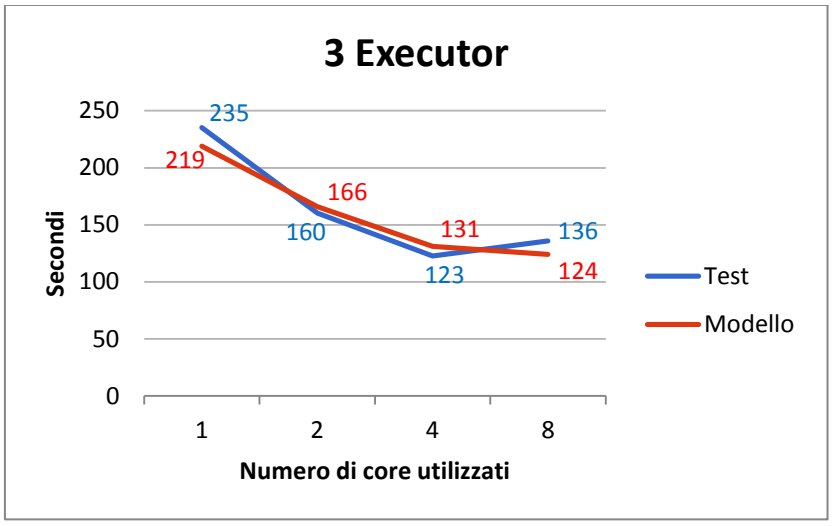

Figura 55 Tempi di esecuzione e stime di una query GPSJ utilizzando 3 executor

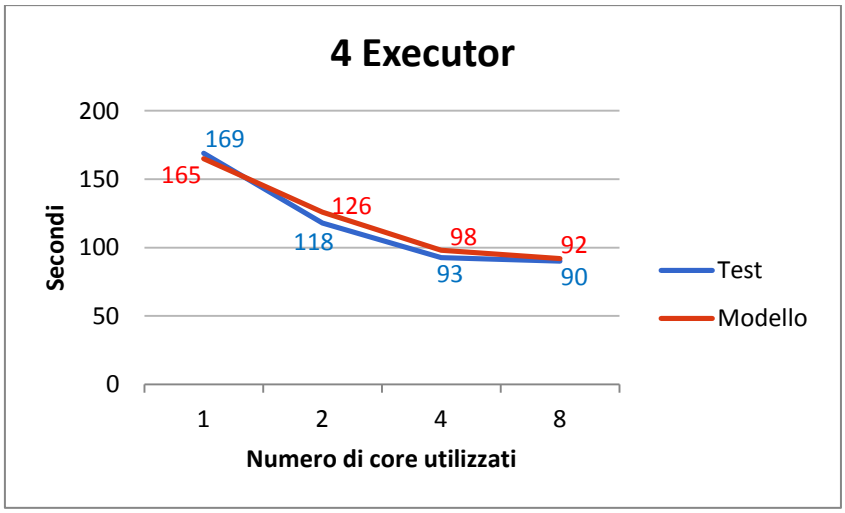

Figura 56 Tempi di esecuzione e stime di una query GPSJ utilizzando 4 executor

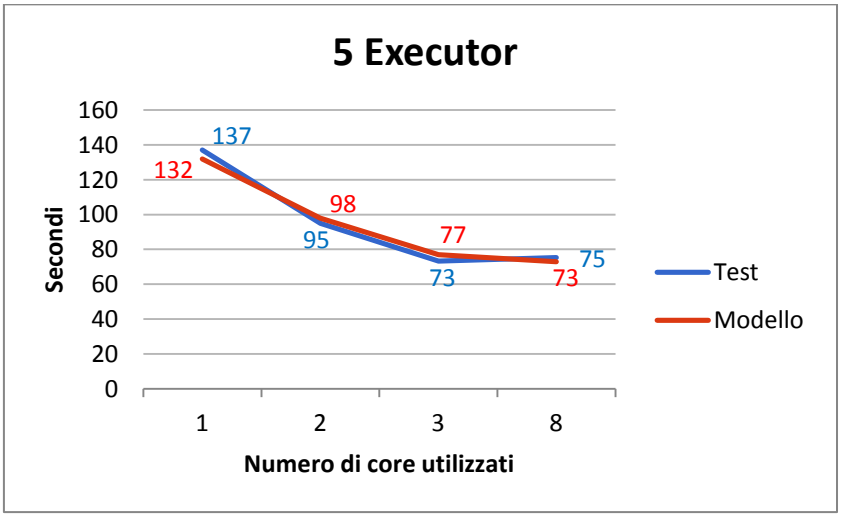

Figura 57 Tempi di esecuzione e stime di una query GPSJ utilizzando 5 executor

Dai test eseguiti si evidenzia come la tendenza del terzo stage a non scalare col numero di executor porti a riallineare la curva che caratterizzava lo scalare dei tempi di esecuzione in base al numero di core. Il modello risulta avere un andamento molto simile a quello dei test e per i casi in cui gli executor sono diversi (più di due) le stime si avvicinano molto hai tempi misurati. L'unico elemento in contro tendenza risulta avvenire nel caso in cui i core siano 8 dove i test evidenziano addirittura un rallentamento mentre il modello prevede un valore stazionario. Questo può essere dovuto al fatto che di default in Spark non viene utilizzata la consolidation e quindi è possibile che con diversi core il numero di richieste di I/O sui file richiesti dai task possa rallentare ulteriormente l'executor. Nei primi due casi l'andamento osservato risulta essere lo stesso tra il modello e i test ma molto scostato (c'è una sottostima del modello). Tale sottostima riguarda principalmente il terzo stage nel quale viene eseguito il join e dove la quantità di dati in shuffle read e shuffle write risulta essere più rilevante. Purtroppo a causa delle poche informazioni che è stato possibile ottenere dall'interfaccia Spark UI e dall'analisi dei log dell'applicazione non è stato possibile chiarire il motivo di tale sottostima.

Dai dati ottenuti si può fare una considerazione su come ottimizzare i tempi in base alle risorse disponibili. Risulta infatti evidente sia dal modello che dai test che dovendo scegliere tra aumentare il numero di core o il numero di executor da assegnare all'applicazione conviene scegliere la seconda opzione:

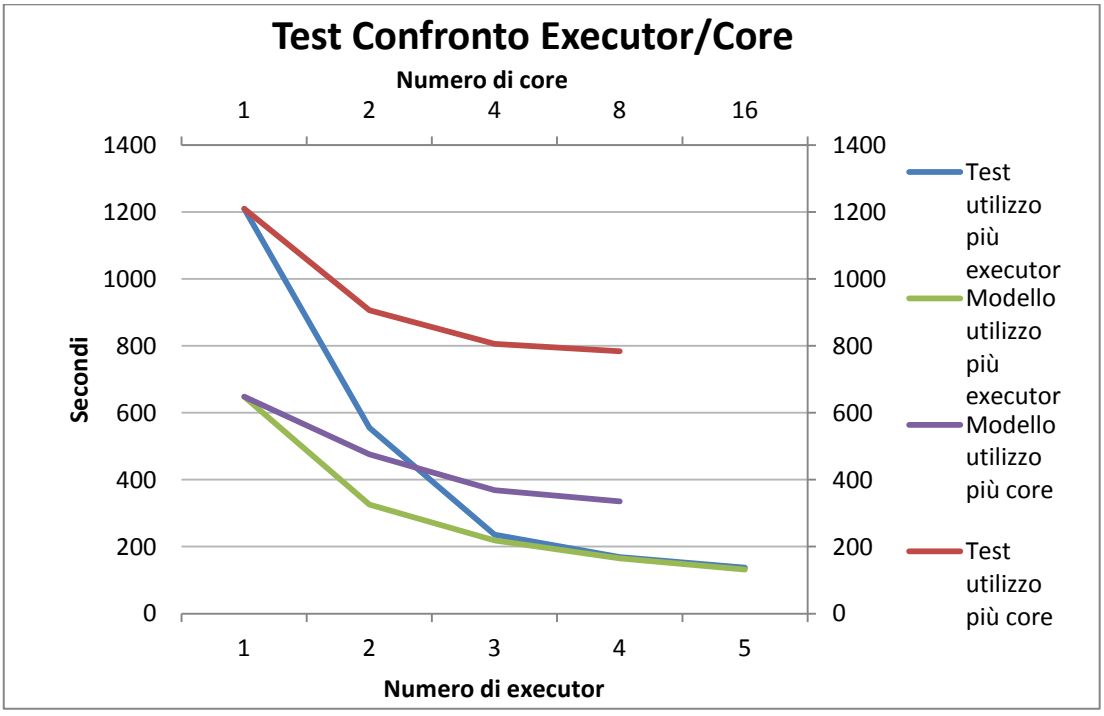

Figura 58 Grafico di confronto tra l'utilizzo di più core su di un executor o più executor con gli stessi core

Nel grafico vengono confrontati i valori stimati e sperimentali dell'esecuzione della query GPSJ utilizzando un executor e aumentando il numero di core (linea rossa e linea viola) con i valori stimati e sperimentali utilizzando sempre un solo core ma con diversi executor (linea blu e verde). Si può osservare come utilizzando più executor i tempi scalino molto più velocemente grazie all'utilizzo di più risorse messe a disposizione da più nodi evitando quindi tutti i problemi relativi alla loro condivisione che si hanno utilizzando più core sulla stessa macchina.

5 Conclusioni

L'evoluzione e la forte spinta che ha caratterizzato i Big Data negli ultimi anni ha mosso l'interesse di numerose aziende che hanno trovato nella loro analisi un potente strumento business che, se correttamente utilizzato, può comportare un vantaggio sia a livello di rapporto col cliente sia nel confronto di aziende concorrenti sul mercato. Quindi sia grandi aziende sia i settori di ricerca si sono quindi focalizzati su tale ambito portando alla realizzazione di diversi sistemi che permettano la l'elaborazione dei Big Data sfruttando dei cluster. Tra questi i più utilizzati dalla comunità in questo periodo si può trovare Hadoop che permette di gestire grandi moli di dati su cluster di commodity hardware ma non in una maniera così veloce come Spark che permette anch'esso di elaborare dataset a livello di Big Data ma con una maggiore efficienza. Entrambi i sistemi sfruttano i nodi di un cluster e quindi tutte le risorse che porta con se ogni nodo per memorizzare ed analizzare i dati. Nel caso particolare di Spark i dati vengono elaborati direttamente in memoria centrale permettendo quindi di avere una velocità di esecuzione delle query fino a 100 volte maggiore rispetto ad Hadoop. Ovviamente la gestione di queste grandi moli di dati non più in un unico punto centralizzato ma su diversi nodi comporta una serie di accorgimenti come la necessità di inviare i dati da nodo a nodo o di essere fault tollerant che vanno a complicare il sistema stesso. Ci si è concentrati quindi su Spark per realizzare una funzione di costo che modelli nel modo più chiaro possibile il suo comportamento. Si è quindi studiato e compreso tale framework andando a realizzare un'insieme di formule che ne costituiscono un modello. Una comprensione profonda di Spark e di come l'esecuzione delle interrogazioni sia effettivamente eseguita permette non solo di poter programmare in maniera più efficiente le applicazioni da eseguire sul cluster ma può dare anche valide informazioni a livello di progettazione di un sistema per l'interrogazione di dataset a livello di big data.

Il modello è stato confrontato con test effettuati sul cluster del gruppo di ricerca per valutarne la correttezza ottenendo valori sperimentali da confrontare con quelli ottenuti matematicamente sia per quanto riguarda la quantità di dati letti scritti e trasferiti sia per quanto riguarda i tempi effettivi di esecuzione. I risultati ottenuti per quanto riguarda la quantità di dati si avvicinano molto ai dati sperimentali in tutti i casi di studio presi ad esempio. Per quanto riguarda i tempi si è osservata una buona valutazione nel caso siano utilizzati diversi executor mentre si è evidenziato un leggero scostamento nel caso di utilizzo di solo uno o due executor. Il secondo caso però risulta avere un'importanza minore dato che nell'utilizzo classico di Spark su di un cluster viene effettuato sfruttando più executor.

Questa affermazione è stata anche confermata dall'ultimo confronto fatto (sia sperimentalmente che attraverso il modello) che evidenzia come si abbia un miglioramento dei tempi maggiore nel caso in cui siano utilizzati più executor (e quindi più nodi) piuttosto che più core. Questo risulta dato dal fatto che sfruttando più nodi vengono anche sfruttate più risorse come il disco, la connessione sulla rete, la RAM ecc ecc andando a diminuire i problemi relativi alla condivisione di solo alcune di queste risorse da più unità di esecuzione contemporaneamente, lasciandone altre in attesa.

Tra i possibili sviluppi futuri che possono seguire si può valutare una migliore comprensione del comportamento. Durante i test sono stati utilizzati strumenti puntuali per la rilevazione dei dati come la Spark UI e l'osservazione dei dati di log ma sempre attraverso un analisi umana. Si può invece considerare di sfruttare i dati scritti su log per realizzare uno strumento di lettura e visualizzazione automatica che permetta non solo di avere dati più precisi e in quantità maggiori ma anche di eseguire un grande numero di applicazioni di seguito lasciando poi in un secondo momento la fase di analisi dei dati ottenuti. I dati di log sono organizzati in formato JSON quindi risulterebbe piuttosto semplice la creazione di uno strumento che ne faccia il parsing.

La funzione di costo realizzata si focalizza principalmente sui concetti fondamentali del sistema come la lettura dei dati, il loro trasferimento sulla rete. Il numero di variabili in gioco risulta però essere elevato data la complessità del sistema. Ad esempio non è stato considerato il caso del broadcast join dato che le due tabelle prese ad esempio erano entrambe di dimensioni elevate andando a rendere l'utilizzo di tale join non conveniente a causa della necessità di andare a centralizzare la seconda tabella sul driver prima di realizzare il broadcast vero e proprio. Nel caso però si esegua uno studio sull'utilizzo di tabelle molto grandi in join con tabelle di dimensioni ridotte potrebbe risultare interessante la valutazione dei tempi aggiungendo tale possibilità al modello di costo. Inoltre i parametri che Spark permette di settare sono numerosi e la modifica di tali parametri può spesso causare una modifica anche nel comportamento e quindi nei tempi considerevole. Per semplicità i parametri utilizzati sono rimasti fissi durante lo studio e i test ma potrebbe risultare interessante andare a eseguire uno studio su tali parametri e come la modifica di ognuno di essi si possa riflettere sui tempi di esecuzione di determinate query.

Bibliografia

[1] D. Edd, «What is big Data?,» https://www.oreilly.com/ideas/what-is-big-data.

[2] «ITU releases 2015,» http://www.itu.int/net/pressoffice/press_releases/2015/17.aspx#.VkNZgLcvfIV.

[3] http://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/).

[4] «How Much Data Is Generated Every Minute On Social Media?,» http://wersm.com/how-much-data-is-generated-every-minute-on-social-media/.

[5] P. Sadalage, «NoSql Databases: An Overview,» https://www.thoughtworks.com/insights/blog/nosql-databases-overview.

[6] S. P. Bappalige, «An introduction to Apache Hadoop for big data,» http://opensource.com/life/14/8/intro-apache-hadoop-big-data.

[7] T. D. M. Z. Michael Armbrust, «Scaling Spark in the Real World: Performance and Usability».

[8] A. K. P. W. &. M. Z. Holden Karau, Learning Spark,Lightning-Fast Data Analisys.

[9] «Tuning Spark applications,» https://www.princeton.edu/researchcomputing/computational-hardware/hadoop/spark-memory.

[10] S. Penchikala, «Big Data Processing with Apache Spark,» http://www.infoq.com/articles/apache-spark-introduction.

[11] M. C. T. D. A. D. J. M. Matei Zaharia, «Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing».

[12] R. Ho, «Big Data Processing in Spark,» http://horicky.blogspot.it/2015/02/big-data-processing-in-spark.html.

[13] M. C. M. J. F. S. S. I. S. Matei Zaharia, «Spark: Cluster Computing with Working Sets».

[14] M. M. Johan Eder, Advanced Information Systems Engineering.

[15] S. Ryza, «How-to: Tune Your Apache Spark Jobs,» http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/.

[16] «Job Scheduling,» https://spark.apache.org/docs/1.0.2/job-scheduling.html.

[17] https://spark.apache.org/docs/1.2.0/api/java/org/apache/spark/scheduler/Task.html.

[18] M. Armbrust, «Catalyst: A Query Optimization Framework for Spark and Shark,» https://spark-summit.org/2013/talk/armbrust-catalyst-a-query-optimization-framework-for-spark-and-shark.

[19] R. S. X. C. L. H. Michael Armbrust, «Spark SQL: Relational Data Processing in Spark».

[20] «Network Topology and Hadoop,» http://blog.csdn.net/heyutao007/article/details/5597372.

[21] B. Bự, «Understand the shuffle component in spark-core,» http://www.trongkhoanguyen.com/2015/04/understand-shuffle-component-in-spark.html.

[22] A. O. Aaron Davidson, «Optimizing Shuffle Performance in Spark».

[23] K. Ousterhout, «Shuffle Internals,» https://cwiki.apache.org/confluence/display/SPARK/Shuffle+Internals.

Ringraziamenti

Vorrei ringraziare prima di tutti i miei genitori Marco e Patrizia per avermi dato la possibilità di intraprendere questo lungo percorso e per avermi sempre sostenuto in tutte le mie scelte ed essermi sempre stati accanto senza farmi mancare mai nulla soprattutto nell'affetto. Un ringraziamento speciale anche alla mia compagna Valentina che è stata al mio fianco aiutandomi in ogni occasione con amore e pazienza soprattutto nei momenti più difficili. Un ringraziamento anche a mia sorella Elena per essere stata un esempio da seguire. Un grazie anche i Good All Boys, da chi è al mio fianco fin dall'asilo fino a chi ha condiviso la mia strada negli ultimi anni. Un ringraziamento anche a Manuele amico da sempre e sempre presente nei momenti importanti. Ringrazio anche mia nonna Maria e i miei zii Paolo e Patricia che mi sono stati accanto e che mi hanno sempre sostenuto e voluto bene. Ringrazio il professor Golfarelli e il dottor Baldacci che mi hanno seguito nel percorso di tesi ad ogni passo con grande professionalità e pazienza. Desidero ringraziare anche Stefano e tutta la NewTeam per la grande disponibilità che hanno dimostrato nei miei confronti in questi due anni. Un grazie anche a tutti gli amici che durante il percorso sono stati al mio fianco, chi per poco tempo e chi per tutti questi anni. Infine un bacio ai miei due nipotini Anna e Mattia che con la loro spensieratezza hanno reso le giornate di studio più liete.