ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

UN SISTEMA PER L'ACQUISIZIONE AUTOMATICA DEI METADATI PER SPARKSQL

Elaborato in
LABORATORIO DI BASI DI DATI

Relatore
Prof. MATTEO GOLFARELLI

Co-relatore
Dott. ENRICO GALLINUCCI

Presentata da
SHAPOUR NEMATI

Seconda Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

Big Data

Spark SQL

Cost Model

Visualization

A mia madre, eterna studentessa,
che è sempre un CFU avanti a me

Indice

Introduzione

Nell'ultimo decennio il ruolo dei Big Data è diventato centrale in moltissimi campi, ed a fare i conti con essi non sono più solo le grandi multinazionali, ma anche aziende di medie dimensioni, il mondo accademico, e le startup. I primi articoli su sistemi a scopo generale per l'elaborazione dei Big Data sono state pubblicate da Google. In particolare Google File System [6] del 2003 e Map Reduce [4] del 2004 hanno avuto un impatto profondo sulla comunità Open Source ed ispirato la realizzazione di vari progetti con lo scopo di poter gestire i Big Data. Il più importante è Apache Hadoop, oggi standard di riferimento per le piattaforme Big Data, e framework sul quale sono sviluppati moltissimi altri componenti. Con il tempo sono nati altri progetti che, basandosi su Hadoop, ne hanno espanso le funzionalità, adeguandolo alle nuove necessità, primi fra questi il gestore di risorse YARN ed il motore di esecuzione Spark, che si occupano rispettivamente della gestione del bilanciamento del carico di lavoro e dell'utilizzo efficiente dell'hardware a disposizione, evitando operazioni non necessarie di salvataggio su disco altrimenti richieste dall'implementazione Hadoop del paradigma Map Reduce.

In questo contesto, un approccio stile RDBMS classico risulta non essere adeguato alle tecnologie ed alle moli dei Big Data; tuttavia, l'astrazione relazionale rimane uno degli strumenti prediletti dagli analisti per l'interrogazione dei dati. Per questo motivo è stato sviluppato, fra i tanti, il progetto SparkSQL, che consente di utilizzare ad alto livello query in linguaggio SQL; le interrogazione vengono mappate in primitive Spark, garantendo efficienze e mettendo a disposizione dell'utente linguaggi di alto livello con i quali ha già familiarità. SQL è un linguaggio dichiarativo, quindi è necessario che il sistema scelga fra i diversi possibili piani di esecuzione quello più efficiente. Nel caso di SparkSQL l'ottimizzatore Catalyst è prevalentemente basato su regole, ed utilizza in maniera marginale le statistiche. Nei sistemi RDBMS, più maturi delle piattaforme Big Data grazie all'età e la formalità dei primi, il modello di ottimizzazione basato su statistiche è un elemento fondamentale, dato il comportamento potenzialmente molto diverso fra un dataset e l'altro. Una possibile risoluzione di questo problema è presentato nell'articolo [7], che propone un modello di costo per SparkSQL, il quale basandosi su statistica e

metadati computa il tempo necessario all'esecuzione di una query. Tale modello necessita di varie informazioni derivanti dai diversi componenti dell'ambiente del cluster sul quale le query vengono eseguite, e può essere complesso per un utente non esperto recuperare tutti i dati necessari al modello di costo. È stata sviluppata l'applicazione web SparkTune, che effettua i calcoli relativi al modello di costo, ma richiede l'inserimento manuale di tutti i parametri.

In questa tesi è stato sviluppato un sistema per l'acquisizione automatica di metadati per il modello di costo, assieme ad una semplice interfaccia web integrata sul sito SparkTune, dove è già possibile – immettendo manualmente i dati – calcolare il costo di una query secondo il modello di costo. L'interfaccia web sviluppata guida l'utente nell'inserimento dei dati, salvandoli all'interno di un database consultabile dalle altre pagine dell'applicazione, rendendoli disponibili per il calcolo mediante il modello di costo. La complessità di questo lavoro risiede anche nella natura eterogenea in tipologie e fonti dei parametri: alcuni richiedono di verificare la configurazione hardware del cluster, altre le performance, altri ancora di recuperare metadati relativi alle tabelle interrogate tramite query.

Di seguito una breve introduzione ai capitoli

- Capitolo 1: introduzione alle tecnologie fondamentali per l'analisi dei Big Data, in particolari quelle abilitanti a SparkSQL;
- Capitolo 2: descrizione del modello di costo, che introduce i concetti generali per il calcolo dei tempi di esecuzione attraverso blocchi base relativi a Spark, e gli algoritmi di join specifici per SparkSQL;
- Capitolo 3: descrizione del progetto sviluppato, la sua architettura, i moduli che lo compongono, l'interazione con l'utente, ed il deploy.

Capitolo 1

Tecnologie

Il sistema studiato e sviluppato per questa tesi ha una componente tecnologica prevalente, in quanto il modello di costo che ha portato alla sua realizzazione è specifico per SparkSQL, componente eseguito in un ambiente dove altri framework e servizi completano le sue funzionalità. Di seguito saranno introdotte le tecnologie utilizzate, mettendo in evidenza quali problemi esse risolvano, il loro funzionamento interno, e l'interazione fra di essi, per poter comprendere il modello di costo.

1.1 Apache Hadoop

Apache Hadoop è un framework basato sui modelli proposti da Google nelle pubblicazioni su Google File System[6] e MapReduce[4], per immagazzinare e processare Big Data su un cluster di computer.
Un cluster è un insieme di computer detti nodi ciascuno con un sistema operativo, visti esternamente come un'unica entità computante. I nodi sono suddivisi in rack, in base all'appartenenza alla stessa rete locale. La popolarità di tali sistemi è dovuta alla possibilità di riutilizzare hardware comunemente impiegato per uso singolo, riducendo i costi relativi all'acquisto di hardware nuovo e fornendo un sistema per più utilizzi. Ciò è possibile in quanto il middleware per l'utilizzo del cluster si fa carico di orchestrare l'esecuzione, con particolare attenzione al tema del bilanciamento del carico di lavoro. L'impiego di hardware di riciclo è diventato nel tempo uno standard per il cluster computing, ed i sistemi che consentono di utilizzare cluster di computer tengono in considerazione questo fattore, soprattutto in relazione alla probabilità più alta di errori rispetto ad hardware specializzato o nuovo.

1.1.1 Architettura di alto livello

Hadoop è un sistema complesso, composto da diverse componenti, che consentono ad un client di operare sul cluster con le stesse astrazioni che si utilizzerebbero su una macchina singola.
Lo stato dell'arte attualmente vede Hadoop come il framework sul quale si appoggiano quasi tutte le applicazioni Big Data. Hadoop è Open Source, e questo ha contribuito al suo ampio utilizzo, consentendo di integrare altri framework in maniera efficiente. L'ecosistema cresciuto attorno ad Hadoop conta più di 100 progetti [8], fra sistemi di analitica, algoritmi, persistenza, interfacce SQL e NoSQL, logistica e parallelizzazione. Solo pochi di questi progetti sono trattati in questa tesi, e per poter apprezzare la complessità dell'ecosistema (seppur senza citare tutti i progetti) è presentata l'immagine 1.2 Hadoop Distributed File System (HDFS) ne è il file system ad alta scalabilità ed affidabilità, pronto per il parallelismo. Yet Another Resource Negotiator (YARN) può essere identificato come il sistema operativo di Hadoop [5], ed è il software di gestione del cluster che controlla le risorse allocate alle applicazioni. I motori di esecuzione sono dei framework che utilizzano i servizi forniti da YARN ed HDFS per stabilire una strategia esecutiva ottimizzata per una o più famiglie di operazioni. Verranno trattati i due motori MapReduce ed Apache Spark. Le applicazioni vengono scritte per un determinato motore utilizzando le relative API, implementando solo la logica della computazione mentre il motore si occupa della distribuzione sul cluster. Hive consente ad Hadoop di essere utilizzato come infrastruttura per Data Warehouse, fornendo un'interfaccia SQL ai dati immagazzinati in HDFS e tenendo traccia dei metadati. La figura 1.1 mostra l'interazione dei vari componenti.

1.1.2 HDFS: un file system distribuito

HDFS è un file system non Posix basato sul modello del Google File System[6], particolarmente efficiente nell'ambito dell'analisi dei dati grazie ad una serie di assunzioni basate sul contesto in cui il sistema lavora, che hanno permesso di semplificare il modello dove questo era pesante, ed aggiungere meccanismi di prevenzione o risoluzione di problematiche frequenti, sulla base delle seguenti assunzioni:

- Il sistema è composto da molte macchine poco costose e di riciclo, con hardware eterogeneo. Pertanto è possibile che ci siano dei malfunzionamenti, quindi è necessario rendere il sistema resistente ed in grado di recuperare;

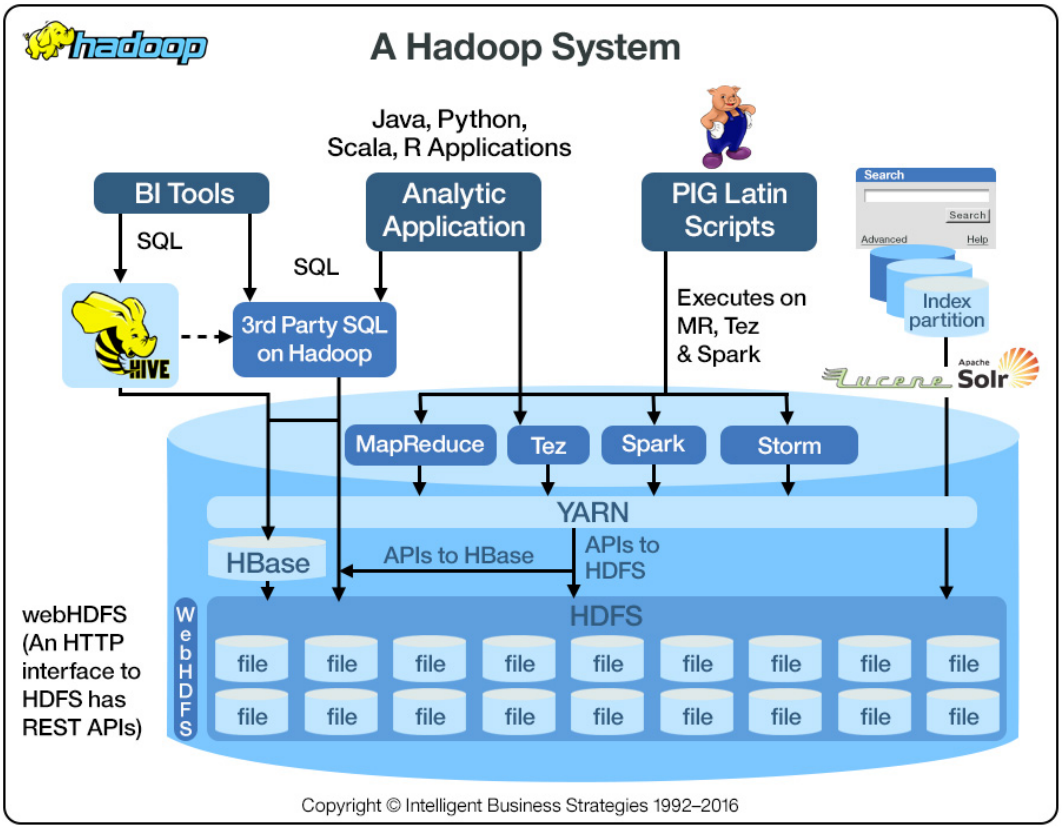

Figura 1.1: I principali componenti di Hadoop e le loro interazioni[5]

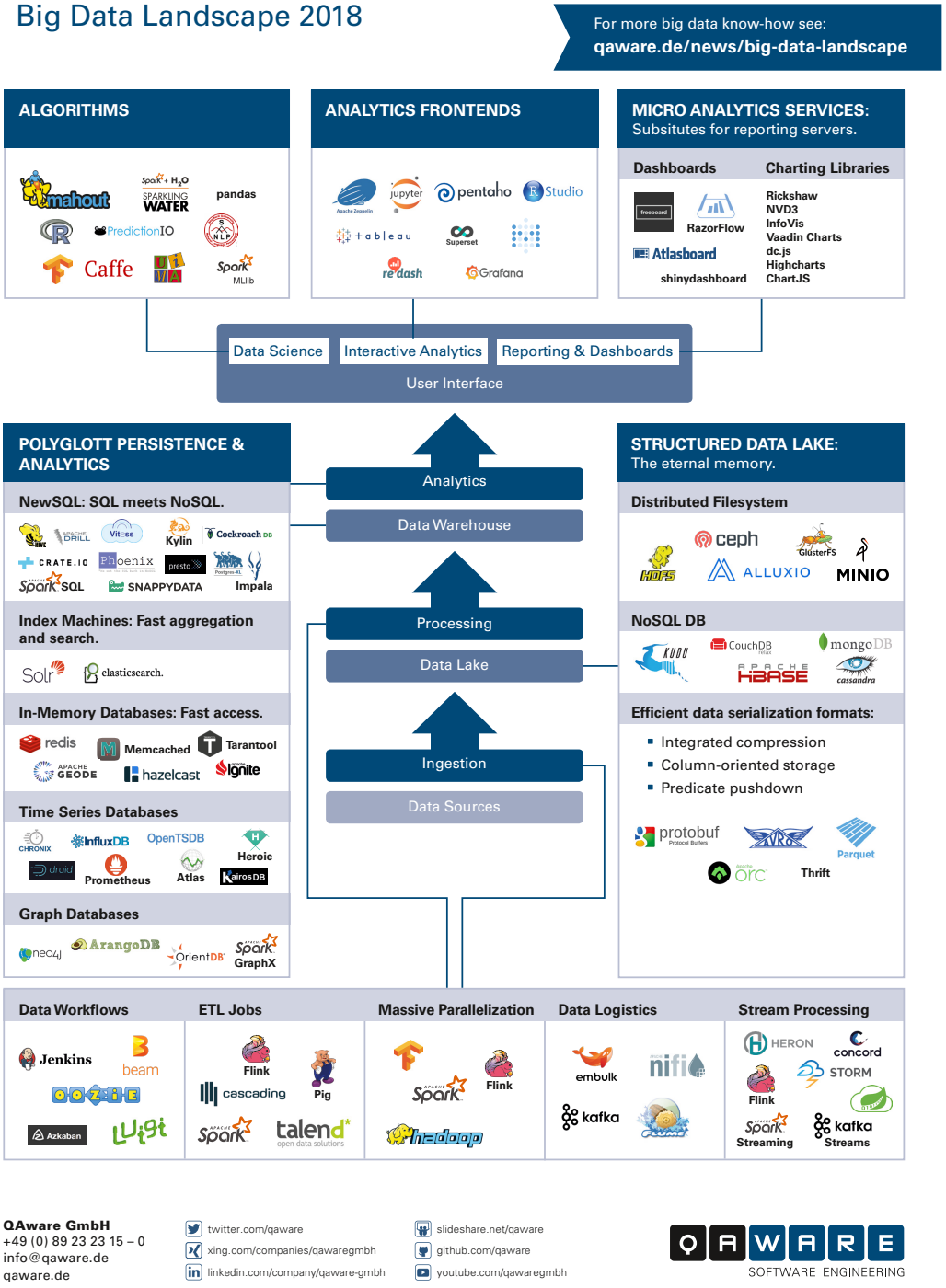

Figura 1.2: Alcuni progetti dell'ecosistema Hadoop [10]

- I file sono prevalentemente di grandi dimensioni, e bisogna ottimizzare questi anche se a discapito dei più piccoli;
- Una volta scritti, i file sono raramente modificati, e quando ciò avviene, solitamente si tratta di un'operazione di aggiunta in coda;
- Una larghezza di banda alta e sostenuta è più importante del tempo di latenza nella maggior parte delle applicazioni.

HDFS si compone di più macchine, che possono servire il ruolo di NameNode (master in Google File System) oppure quello di DataNode (chunkserver in Google File System). Una rappresentazione dei componenti e delle interazioni esistenti fra questi è visibile nell'immagine 1.3.
Il NameNode mantiene i metadati relativi alla struttura del file system ed alla ripartizione in blocchi dei file, memorizzando su quali macchine sono salvati. Inoltre, espone delle API che forniscono le informazioni necessarie a realizzare le operazioni di creazione, cancellazione, apertura, chiusura, lettura e scrittura dei file, permettendo ad un client esterno di utilizzare il file system. Bisogna precisare che il NameNode non effettua un accesso ai file, ma fornisce le istruzioni per potervi accedere, rendendo possibile un alto livello di parallelismo senza che il NameNode ne diventi il collo di bottiglia.
Il DataNode sono le macchine su cui vengono gestite le operazioni di Input/Output che hanno effetto sul file system, memorizzano i dati in blocchi di file, ciascuno di questi replicato su più macchine per garantire resistenza ai guasti: di default il fattore di replicazione è pari a 3 copie, distribuite su macchine diverse. Esistono diversi formati di file compatibili con Hadoop, strutturati e non, ciascuno avente costi diversi per operazioni diverse. Il DataNode comunica periodicamente con il NameNode con una serie di messaggi denominati HeartBeat tramite i quali vengono notificati i cambiamenti avvenuti e lo stato attuale. Grazie a questi messaggi il NameNode è in grado di determinare un eventuale malfunzionamento, tenere traccia della distribuzione del carico di lavoro, e gestire la ridondanza.

1.1.3 Il paradigma di programmazione MapReduce

MapReduce è un modello di programmazione ed un motore di esecuzione per processare e generare dataset di grandi dimensioni, separando la computazione in due fasi per poterne automatizzare una parallelizzazione ottimizzata, togliendo questo onere dal programmatore e rendendo le applicazioni così scritte automaticamente scalabili su un cluster.
La funzione Map prende in input dei dati sotto forma di coppie chiave/valore, e produce un nuovo set intermedio di coppie, che vengono raggruppate per chiave

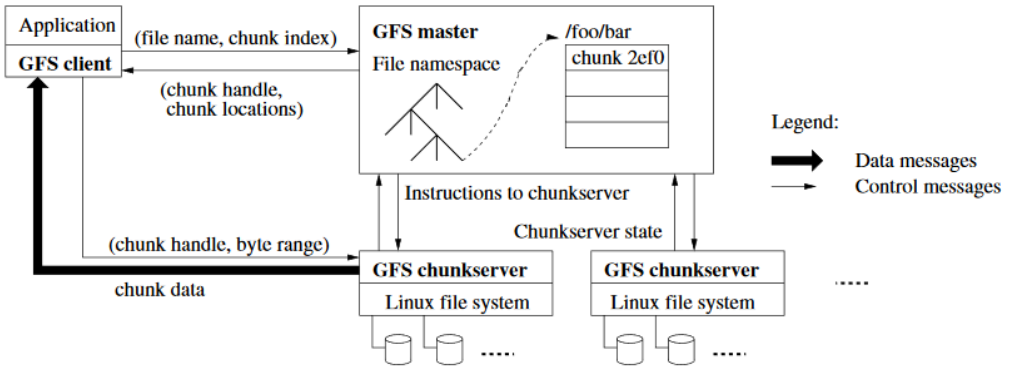

Figura 1.3: Architettura di GFS [6]

e passate come input alla funzione Reduce, la quale svolge una computazione il cui risultato è una lista contente l'output desiderato. In questo paradigma è possibile codificare moltissime delle operazioni che normalmente si effettuano su dataset di grandi dimensioni, anche se a volte è necessario concatenare più funzioni per garantire buone prestazioni in parallelo, in quanto è possibile che alcune funzioni Reduce abbiano inizio prima che tutte le Map terminino. La figura 1.4 mostra il funzionamento della MapReduce in un ambito distribuito, evidenziando il parallelismo e la contemporaneità dell'esecuzione delle due funzioni.

Il motore MapReduce consiste di un'implementazione da parte dell'utente delle funzioni di Map e Reduce, ed un'implementazione del motore che si occupa di distribuire la computazione. Quest'ultima può variare notevolmente data la semplicità del modello di base e la diversità di configurazioni del sistema distribuito che la deve eseguire. Nel dettaglio Hadoop implementa direttamente [11] il motore di esecuzione descritto nella pubblicazione di Google [4], schematizzato in 7 passaggi:

1. L'input viene diviso in M parti dette split, ed una copia del programma viene eseguito su varie macchine del cluster.

2. Una copia del programma è detta Master, che seleziona dei nodi worker a cui assegnare i task di Map e Reduce.

3. I nodi worker leggono lo split assegnato, applicano la funzione Map definita dall'utente ed immagazzinano in un buffer in memoria il risultato intermedio.

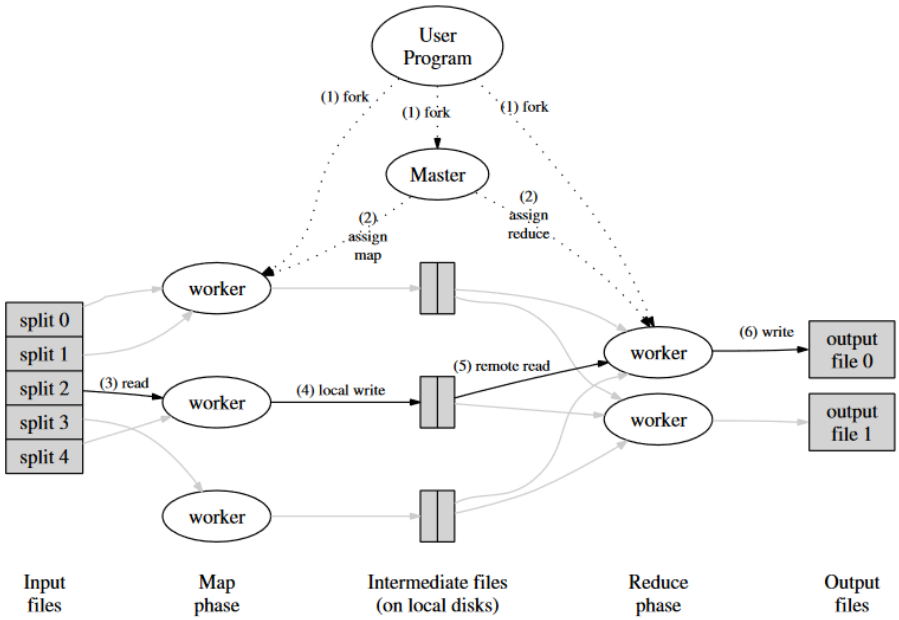

Figura 1.4: Architettura di MapReduce [4]

4. Periodicamente il buffer è scritto sul disco locale, e l'indirizzo di questi blocchi è inviato al Master

5. Quando i worker vengono notificati dal Master, leggono i blocchi da remoto, ed ordinano per chiave.

6. Per ogni chiave distinta il worker passa i valori alla funzione Reduce, ed aggiunge il risultato in coda al file di output.

7. Al termine di tutti i task il master notifica il programma utente, comunicando gli indirizzi dei file di output.

Al termine di queste operazioni non vi è una unificazione automatica degli output, in quanto questi potrebbero essere l'input di un'ulteriore MapReduce, e sarebbe quindi più efficiente che fosse già distribuito anziché centralizzato. Questo concetto è riutilizzato sia per le fasi intermedie dell'esecuzione, sia all'inizio, in quanto Hadoop si avvale delle caratteristiche di HDFS per ottimizzare MapReduce, in particolare con il concetto di Data Locality, che consiste nell'assegnazione delle funzioni di Map alle macchine che contengono le relative porzioni dell'input, in modo da ridurre al minimo l'utilizzo della rete. Esistono altre ottimizzazioni, legate più al cluster e ad un utilizzo bilanciato delle sue

risorse, indipendenti da MapReduce e quindi delegati ad un altro componente, YARN, trattato nella prossima sottosezione.

1.1.4 Resource Manager

In Hadoop 1 MapReduce svolgeva due ruoli correlati ma distinti: gestione delle risorse ed esecuzione delle applicazioni. Questo approccio ha portato ad ottimizzare in favore di MapReduce, a discapito delle applicazioni che non seguono il paradigma, in particolare quelle che richiedono bassa latenza, motivo che assieme alla separazione delle responsabilità ha portato allo sviluppo di YARN: Yet Another Resource Negotiator, prima come parte di MapReduce, poi come progetto indipendente a servizio di tutti i motori di esecuzione.
YARN è un framework per lo scheduling dei processi e la gestione delle risorse che si posiziona allo stesso livello di MapReduce, con il quale lavora in parallelo, delegandogli il monitoraggio, la gestione degli errori e la computazione. YARN è un servizio centralizzato con visione globale alle risorse del cluster, sulle macchine del quale alloca container per eseguire le applicazioni richieste. I due componenti di YARN sono il ResourceManager, globale, che si occupa di allocare sul cluster risorse per le applicazioni ed i container, e gli ApplicationManager, uno per applicazione, responsabili della gestione dei container, il monitoraggio delle risorse della macchina e la comunicazione con il ResourceManager. La richiesta di esecuzione di un task viene fatta al ResourceManager, che la valida e la passa allo scheduler, mentre l'Application Master gestisce gli aspetti relativi al ciclo di vita dell'applicazione ovvero la gestione delle risorse, il flusso di esecuzione, gli errori, e l'ottimizzazione rispetto al task specifico, inviando richieste al ResourceManager. Una rappresentazione grafica di questa architettura è mostrata in figura 1.5.

Per quanto sia possibile scrivere applicazioni direttamente in YARN è prassi far comunicare il motore di esecuzione con YARN e scrivere le applicazioni sul motore di esecuzione adatto.

1.2 Apache Spark

Apache Spark è un motore di esecuzione a scopo generico, compatibile con Hadoop, basato su necessità moderne e costruito attorno allo stato attuale della tecnologia.

1.2.1 Perché Spark?

Hadoop e MapReduce sono stati resi disponibili al pubblico attorno al 2012, con il rilascio di Hadoop 1, ma si basano su una pubblicazione di Google

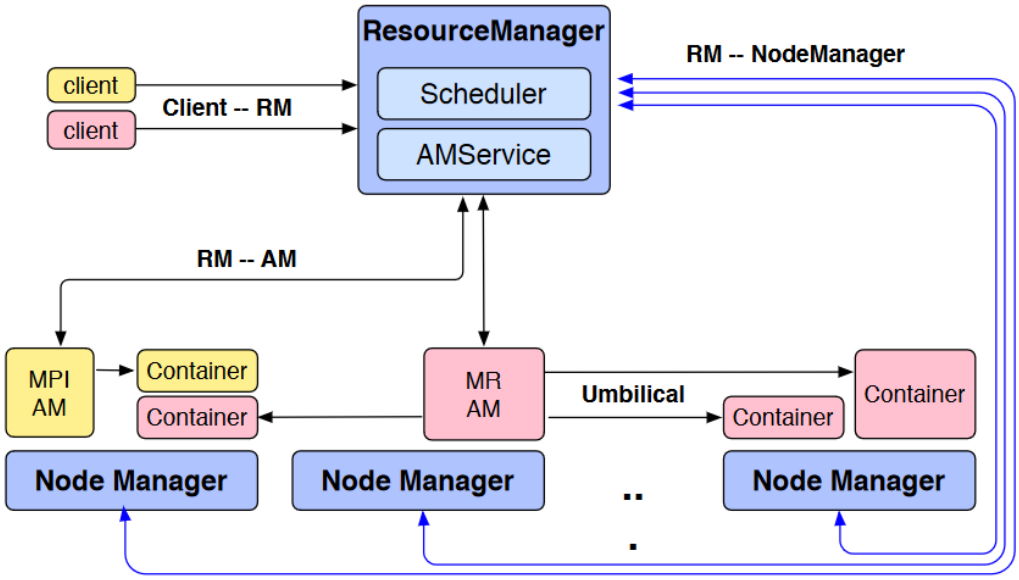

Figura 1.5: Architettura di YARN, esempio con due client [12]

risalente al 2004, il cui lavoro è iniziato nel 2000. Con il tempo sono cambiati hardware, software, mole di dati, ed esigenze degli utenti.

- Hardware
  - Principale fonte di dati da Hard Disk a RAM
  - Processori da core singolo a multi core
- Software
  - Il paradigma di programmazione funzionale sta diventando più popolare
  - Crescente popolarità dei database NoSQL rispetto a quelli SQL
- Dati
  - Maggior numero di compagnie che trattano Big Data
  - Necessità di applicazioni con bassa latenza oltre che processi batch
  - Nuovi casi d'uso con Machine Learning, profilazione utenti ecc.

Spark, programmabile in Scala, Java e Python, sfrutta pipelining, immagazzinamento dei dati in memoria centrale, computazione lazy e compatibilità con Hadoop per fornire un ambiente di esecuzione adatto alle nuove esigenze ed ai cambiamenti tecnologici.

1.2.2 RDD

Spark ha alla base una struttura dati detta RDD, Resilient Distributed Dataset. Gli RDD sono collezioni di dati partizionate e distribuite sui nodi del cluster, con un meccanismo di resilienza che ne consente il recupero automatico quando si verificano degli errori. L'efficienza computazionale che deriva dall'utilizzo degli RDD è legata alla loro immutabilità e l'immagazzinamento in memoria centrale, capacità che consentono di effettuare il pipelining delle operazioni e l'applicazione della lazy evaluation. L'immutabilità semplifica il parallelismo a discapito di una maggiore occupazione in memoria, in quanto una trasformazione richiede di generare il nuovo dataset mentre quello precedente è ancora in memoria, ma non richiede alcuna lock in quanto l'RDD di partenza può essere al più letto in concorrenza da un altro processo, ma non modificato. Il pipelining e la lazy evaluation sono strettamente correlati in quanto la seconda permette di avere il piano di esecuzione completo prima di iniziare la computazione, piano che viene successivamente ottimizzato ed eseguito in pipeline, minimizzando le scritture su disco, l'uso della rete, il numero di iterazioni e fornendo un meccanismo molto efficiente per applicazioni che richiedono basse latenze. Esistono due tipo di operazioni sugli RDD: le trasformazioni che vengono salvate in un albero di esecuzione, e le azioni, che scatenano il processo di ottimizzazione e conseguentemente di esecuzione.

1.2.3 Suddivisione dell'unità di lavoro

In questa sottosezione saranno introdotti alcuni termini ed una notazione grafica, entrambi utilizzati da Spark per suddividere la computazione ed identificare diversi livelli di astrazione. Un'unità di lavoro di Spark, ovvero una serie di trasformazioni ed una azione, prende il nome di Job, che può essere rappresentato come Directed Acyclic Graph, ovvero un grafo aciclico direzionato, dove i nodi sono gli RDD e gli archi le operazioni. Questa rappresentazione enfatizza l'efficienza dell'immagazzinamento in memoria del dataset, in quanto rispetto ad una MapReduce con implementazione tradizionale ogni arco risparmia una scrittura ed una lettura, oltre ad eventuali costi di rete. Ogni DAG rappresenta un Job, che viene diviso in uno o più Stage, ovvero trasformazioni da un RDD ad uno nuovo, che corrispondono pertanto ad una transizione nel grafo. Gli Stage sono suddivisi in task, l'unità fisica di computazione che agisce su una partizione dell'RDD e che viene gestita dallo scheduler. Una rappresentazione della suddivisione è fornita in figura 1.6.

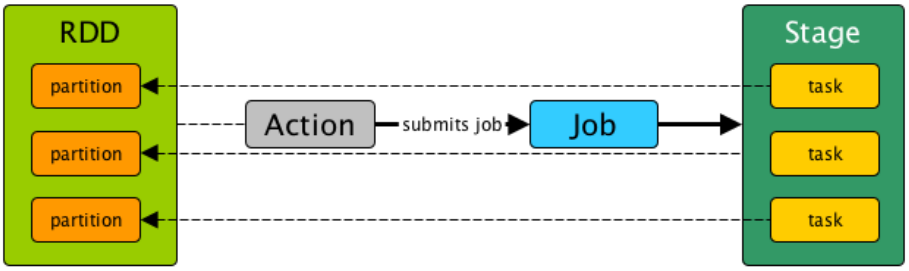

Figura 1.6: Suddivisione di un Job, come indicato in [9]

1.2.4 Spark SQL

Nelle applicazioni Big Data l'approccio relazionale risulta spesso limitante, sia per questioni di performance, a causa della necessità di integrare le operazioni dell'algebra relazionale con quelle procedurali, che per la gestione di dati semi-strutturati o non strutturati. Spark SQL nasce in questo contesto per poter far fronte a tali problematiche sfruttando le funzionalità di Spark, fornendo le astrazioni del mondo relazionale, ed un'integrazione con esso, all'interno di un contesto dove è possibile operare in maniera procedurale e trattare anche dati semi strutturati. Si tratta di un componente che sfrutta le funzionalità di Spark esponendo delle API per utilizzare il linguaggio SQL o i suoi costrutti sui dati. In Spark SQL è presente la struttura dati DataFrame, un'implementazione degli RDD con metadati aggiuntivi per essere equivalente ad una tabella di un Database relazionale.

Catalyst

Analogamente agli RDBMS, in Spark SQL l'utilizzo di un linguaggio dichiarativo richiede un ottimizzatore per scegliere uno fra i possibili piani di esecuzione. Catalyst, l'ottimizzatore di Spark SQL, è estensibile e si occupa di tradurre una interrogazione dichiarativa SQL in una serie di Job, applicando i tipici passi di ottimizzazione: analisi e validazione, ottimizzazione logica, ottimizzazione fisica, e generazione del codice. L'ottimizzazione fisica genera uno o più piani fisici e seleziona il migliore, secondo una strategia prevalentemente basata su regole e con un marginale utilizzo di metadati e quindi di regole basate sul costo. Il processo è esemplificato in figura 1.7.

Nella scelta del piano migliore la maggiore discriminante è l'algoritmo di Join. Spark fornisce due principali algoritmi di Join efficienti per l'utilizzo in un cluster.

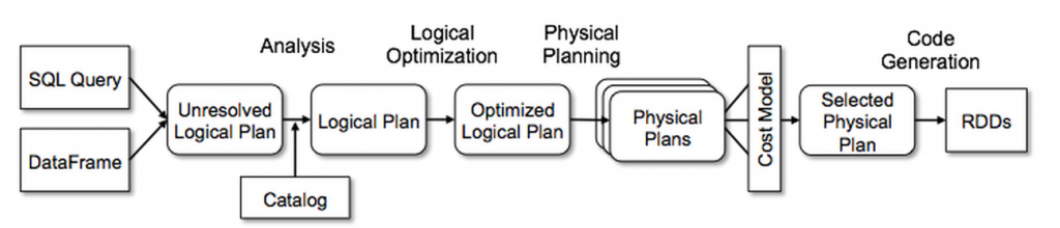

Figura 1.7: Fasi di ottimizzazione con Catalyst [1]

Broadcast Join

Può essere utilizzato solo quando una delle due tabelle non occupa più memoria di quella disponibile. In questo caso la tabella di dimensione inferiore è inviata tramite broadcast a tutti i nodi che fanno parte della computazione, così che localmente venga effettuata una join fra la partizione della tabella più grande e la tabella in memoria.

Shuffle Join

Prima entrambe le tabelle sono ordinate oppure viene calcolato l'hash dei loro attributi di Join, poi vengono suddivise in partizioni uguali in numero. Le partizioni sono salvate su disco, ed avviene un'operazione di shuffling per cui tutte le partizioni con lo stesso intervallo di valori di chiave delle due tabelle diverse sono inviate ad una macchina che ne effettua la Join.

1.2.5 Driver ed Executor

Similarmente a quanto succede nel motore di MapReduce implementato in Hadoop, anche Spark basa il suo modello di esecuzione su due processi: il Driver, in singola copia, che orchestra gli Executor, allocati su alcuni nodi del cluster, pronti a ricevere informazioni. Quando un'applicazione Spark è inviata per l'esecuzione, il driver viene allocato, successivamente il gestore del cluster (YARN, il gestore interno di Spark o altri) alloca le risorse sulle macchine. Una volta connesso Spark invia il codice dell'applicazione agli Executor, per poi assegnargli i task da eseguire. L'architettura è mostrata in figura 1.8.

1.3 Apache Hive

Un altro componente basato su Hadoop è Apache Hive, che consente di utilizzare il cluster Hadoop come data warehouse, ricoprendo il ruolo che è solitamente delegato ad un DBMS. Hive consente all'utente di eseguire query

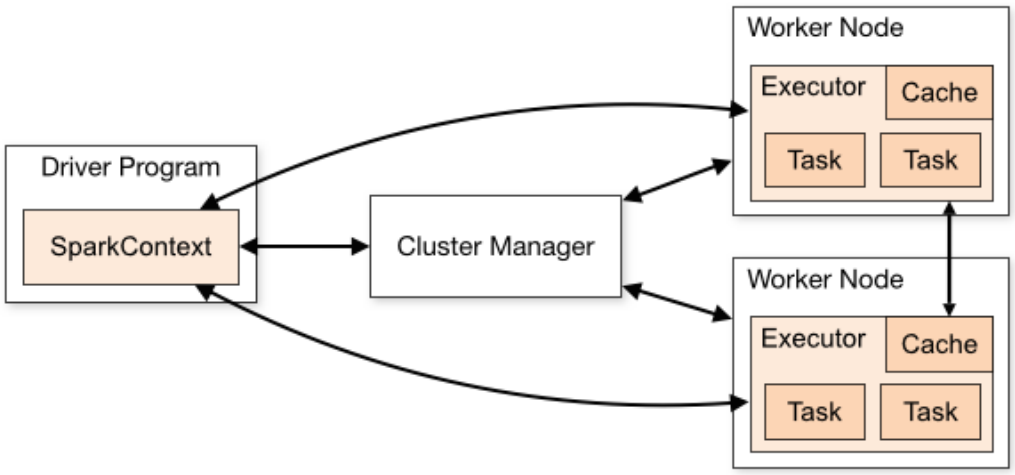

Figura 1.8: Comunicazione fra i componenti di Spark durante l'esecuzione di un'applicazione [2]

nel linguaggio Hive Query Language (HiveQL), molto simile ad SQL. Hive salva tutti i dati in maniera strutturata sfruttando la ridondanza fornita da HDFS, e mantiene i metadati nel MetaStore, approfondito nella sottosezione 1.3.1. A differenza di Spark SQL, dove le query sono implementate tramite primitive di Spark Core, in Hive si utilizzano primitive di MapReduce. Questo consente una alta interoperabilità fra i due componenti, dato che i job di MapReduce prodotti da Hive possono essere eseguiti dal motore di Spark, e SparkSQL può utilizzare i metadati forniti dall'Hive Metastore. Basandosi su Hadoop la componente distribuita di Hive si identifica con la MapReduce ed HDFS, risultando quindi in una JVM per macchina con il servizio di Hive attivo.

1.3.1 MetaStore

Hive salva i metadati in un database relazionale, che può essere configurato in tre modalità diverse: embedded, locale o remoto.
La modalità embedded è quella di default, e consiste in un database Derby (DBMS nativo per la JVM) che è eseguito nella stessa JVM del processo di Hive. Questo limita il numero di sessioni di Hive ad un massimo di una, in quanto più sessioni in JVM differenti non potrebbero comunicare e gestire la concorrenza.
Nella modalità locale il Metastore viene comunque eseguito nello stesso processo di Hive, ma è solo un'interfaccia che si connette tramite connessione standard JDBC all'effettiva istanza di database eseguita su una JVM separata.

La modalità remota prevede che gli stessi processi di Metastore siano esterni alla JVM dell'applicazione Hive, con un database centrale interrogabile anche esternamente ad Hive.

Capitolo 2

Modello di costo

Le tecnologie trattate nel capitolo precedente, per quanto largamente utilizzate, sono ancora in fase di sviluppo e non sono robuste ed evolute come gli RDBMS tradizionali. La criticità principale è relativa a Catalyst, il modulo che effettua l'ottimizzazione delle query in SparkSQL, basato prevalentemente su regole, e con un utilizzo marginale dei metadati e delle statistiche. Lo scopo principale del modello è contestualizzato alla scelta del piano di esecuzione ottimo, ma può essere utilizzato per la messa a punto dei parametri di sistema e l'analisi delle performance.

2.1 Parametri del cluster e dell'applicazione

Un cluster è formato da $\#N$ nodi distribuiti uniformemente su $\#R$ rack, ed ogni nodo ha $\#C$ core. Viene fatta l'assunzione di avere rack e nodi omogenei fra loro, tutti i rack con lo stesso numero di nodi $\#RN$ ed ogni macchina con lo stesso numero di core. I dati sono salvati su HDFS con fattore di ridondanza rf. Un elemento fondamentale per una modellazione corretta è il throughput del disco, calcolato attraverso due funzioni: $\delta_r(\#Proc)$ e $\delta_w(\#Proc)$, che restituiscono la velocità in MB/s per singolo processo con $\#Proc$ processi concorrenti. Il tempo modellato è quello necessario a rendere disponibile in memoria sotto forma di RDD un'informazione su disco, incorporando in maniera implicita i tempi di CPU necessari per decompressione e deserializzazione, oltre alla contenzione delle risorse della macchina da diversi processi.
Viene fatta l'assunzione che i nodi del cluster siano connessi attraverso una rete punto-punto con un limite di ampiezza di banda per ogni connessione. Le funzioni per modellare il throughput di ogni connessione sono $\rho_i(\#Proc)$ e $\rho_e(\#Proc)$, il primo per la velocità in MB/s per processo su una connessione fra due nodi appartenenti allo stesso rack, mentre la seconda è fra nodi appartenenti a rack diversi.

Durante la fase di avvio vengono assegnate ad ogni applicazione Spark varie risorse, fra cui il numero di esecutori $\#E$ e di core per esecutore $\#EC$. Viene fatta l'assunzione che il driver risieda su una macchina diversa da quelle sulle quali vi è un esecutore. Il numero di Shuffle Bucket $\#SB$ influisce sui tempi di esecuzione delle operazioni di shuffle, ed è quindi preso in considerazione. Il numero di partizioni degli RDD è solitamente maggiore del numero di core disponibili, quindi il Resource Manager smista i task in diverse ondate. Un'ondata è l'esecuzione parallela di un insieme di task dello stesso tipo, uno per core disponibile sugli esecutori dell'applicazione, ognuno avente come input una diversa partizione di un RDD. Ogni ondata è omogenea rispetto alla provenienza dei dati, quindi può essere un'ondata locale (L), rack (R), o cluster (C).

2.2 Elementi di base

Le risorse maggiormente influenti sul tempo di esecuzione di un Job Spark sono il disco e la rete, pertanto il sistema si basa su un insieme di elementi base di basso livello, evitando la complessità relativa ad un'analisi puntuale di ogni trasformazione ed azione Spark, che è inoltra soggetta a variare anche considerevolmente con le diverse versioni del software. Ogni elemento modella un'informazione sul cluster, sull'applicazione, o sull'esecuzione di un'operazione su una partizione di un RDD, prendendo in considerazione il parallelismo e la conseguente condivisione delle risorse. Gli elementi sono indipendenti da SQL, e relativi a Spark ed al cluster.

2.2.1 Read

La natura distribuita di Spark e l'appoggio su Hadoop delineano tre tipi di lettura diversa, in ordine di efficienza: sullo stesso nodo, su un nodo dello stesso rack, e su un nodo di un altro rack. Il pipelining delle operazioni consente di trasferire i dati sul nodo di interesse mentre questi vengono letti dal nodo sul quale sono immagazzinati, il che implica, trascurando i brevi tempi di inizializzazione, che il tempo di lettura di un RDD di dimensioni $Size$ sia il massimo fra il tempo di lettura da disco ed il tempo di trasmissione sulla rete. Nel caso della lettura locale il tempo di trasmissione è pari a 0, quindi quello di read è il tempo necessario al core per leggere la partizione di dimensione $Size$ dal disco, calcolato con la funzione $\delta_r(\#Proc)$. Se l'ondata è di tipo rack invece ogni core esecutore riceve una partizione di un RDD da un altro nodo sul rack che non ha un esecutore allocato, fino ad $\#RN - \#RE$ nodi sono disponibili. Si assume una distribuzione uniforme del carico di lavoro e la lettura completa

di una stessa partizione da un solo nodo alla volta, ottenendo una media di $\frac{\#RE \cdot \#EC}{\#RN - \#RE}$ richieste da servire per nodo che non ospita un esecutore. Il costo della lettura da disco risulta essere uguale alla dimensione dell'RDD diviso il throughput del disco in lettura avente come parametro il numero di richieste medie sopracitato.

$$ReadT_R = \frac{Size}{\delta_r(\lceil \frac{\#RE \cdot \#EC}{\#RN - \#RE} \rceil)}$$

Bisogna considerare ora il tempo di trasmissione, dove un esecutore ha $\#EC$ core che ricevono una partizione da uno degli $\#RN - \#RE$ nodi rimanenti. Assumendo una distribuzione uniforme delle partizioni degli RDD sui nodi dello stesso rack, il numero di core che condividono la stessa connessione è al massimo $\lceil \frac{\#EC}{(\#RN - \#RE)} \rceil$, quindi il tempo di trasmissione è dato dalla dimensione dell'RDD divisa per il throughput della rete interna al rack con numero di processi come sopracitato.

$$TransT_R = \frac{Size}{\rho_i(\lceil \frac{\#EC}{(\#RN - \#RE)} \rceil)}$$

Il caso dell'RDD salvato su un rack diverso da quello dove sono istanziati gli esecutori è simile al precedente, con la differenza che i nodi che non ospitano un esecutore possono potenzialmente ricevere una richiesta da tutti i core degli esecutori che appartengono a rack diversi, ovvero $(\#R-1) \cdot \#RE \cdot \#EC$. Queste richieste sono distribuite su $(\#R-1) \cdot (\#RN - \#RE)$ nodi al di fuori del rack che non ospitano un esecutore. Il throughput del disco è quindi calcolato su un numero di processi pari al rapporto fra il numero di core che possono computare ed il numero di nodi che non stanno ospitando un esecutore. Il tempo di lettura è il rapporto fra la dimensione ed il throughput.

$$ReadT_C = \frac{Size}{\delta_r(\lceil \frac{(\#R-1) \cdot \#RE \cdot \#EC}{(\#R-1) \cdot (\#RN - \#RE)} \rceil)} = \frac{Size}{\delta_r(\lceil \frac{\#RE \cdot \#EC}{(\#RN - \#RE)} \rceil)}$$

Dal punto di vista della rete ogni core può essere servito da un nodo che non ospita un esecutore, quindi mediamente la stessa connessione sarà impiegata da un numero di core uguale al rapporto fra i core per esecutore ed il numero totale di non nodi senza esecutori, ed il tempo di transizione è:

$$TransT_C = \frac{Size}{\rho_e(\frac{\#EC}{\lceil (\#R-1) \cdot (\#RN - \#RE) \rceil})}$$

Il principio di data locality è quindi molto utile per evitare possibili colli di bottiglia relativi alla trasmissione su rete delle partizioni da computare.

2.2.2 Write

Il tempo di scrittura per core è equivalente alla quantità di dati da scrivere fratto la velocità di scrittura con $\#EC$ core. Il formato su disco è compresso, quindi la dimensione $Size$ dell'RDD va moltiplicata per $sCmp$, risultando nella seguente formula:

$$Write(Size) = \frac{Size \cdot sCmp}{\delta_w(\#EC)}$$

2.2.3 Shuffle Read

Lo shuffling è il processo di ridistribuzione delle partizioni fra i diversi esecutori, per ottenere un alto livello di parallelismo. Nella Shuffle Read vengono utilizzati $\#SB$ task pari al numero di Shuffle Buckets, ognuno dei quali viene ripartito fra i diversi esecutori. La funzione ShuffleRead indica il tempo necessario alla lettura di un singolo bucket di dimensione $Size$.
Analogamente a quanto succede nell'operazione di Read, lettura e trasmissione su rete sono in pipeline, quindi il tempo totale equivale al massimo fra i due. In questa fase tutti e soli i nodi che ospitano un esecutore sono coinvolti, e non può essere utilizzato il principio di data locality in quanto i dati non sono replicati ed ogni bucket è distribuito su tutti gli esecutori. La porzione di bucket immagazzinata da ogni esecutore è equivalente alla dimensione della partizione fratto il numero di esecutori, misurata in MB. Ogni core richiede in parallelo agli altri esecutori la porzione di bucket, quindi ogni ognuno deve soddisfare $\#E \cdot \#EC$ richieste della dimensione sopra riportata. Considerata l'esecuzione parallela il tempo di lettura è:

$$ReadT = \frac{\#E \cdot \#EC}{\delta_r(\#E \cdot \#EC)}$$

Il tempo di rete è dipendente dalla posizione relativa dei nodi: se sono sullo stesso rack o meno. La probabilità che v esecutori siano sullo stesso rack è pari a:

$$P_{SR}(v) = \begin{cases} \frac{\binom{\#RN}{v}}{\binom{\#N}{v}} \cdot \#R, & \text{se } v \leq \#\text{RN} \\ 0, & \text{altrimenti} \end{cases}$$

Questo valore è 0 se $v \leq \#RN$, in quanto ci sarebbero più nodi rispetto a quelli presenti nel rack. Con semplici proprietà delle probabilità è possibile ricavare quindi la formula del tempo di trasmissione:

$$TransT = \frac{Size/\#E}{P_{SR}(\#E) \cdot \rho_i(\#EC) + (1 - P_{SR}(\#E)) \cdot \rho_e(\#EC)}$$

2.2.4 Broadcast

Il broadcast di un RDD consiste nell'acquisizione di tutte le sue partizioni (di dimensione $Size$) al driver dell'applicazione, che ridistribuisce poi l'RDD agli esecutori per parallelizzare le successive operazioni. Il tempo di esecuzione è dato dalla somma dei tempi delle due fasi appena descritte.

$$Broadcast(Size) = CollectT + DistributeT$$

Le partizioni raccolte in parallelo sono $\#E \cdot \#EC$, una per ogni core di esecuzione, implicando che ogni connessione fra nodo e driver è condivisa da $\#EC$ processi. Come nella Shuffle Read possiamo calcolare un limite superiore al tempo di trasmissione considerando quello fra nodi di rack diversi.

$$CollectT = \frac{Size}{P_{SR}(E+1) \cdot \rho_i(\#EC) + (1 - P_{SR}(\#E+1)) \cdot \rho_e(\#EC)}$$

Il tempo di distribuzione delle partizioni dell'ondata corrente agli executor può essere calcolata come:

$$DistributeT = \frac{Size \cdot \#E \cdot \#EC}{P_{SR}(E+1) \cdot \rho_i(1) + (1 - P_{SR}(\#E+1)) \cdot \rho_e(1)}$$

2.3 Modellazione Query GPSJ

Le query SQL modellate in [7] sono di tipo GPSJ: "Generalized Projection, Selection, Join". Questa famiglia di interrogazioni può essere composta dalle sole operazioni di join, selezione dei predicati, ed aggregazione. Un albero di esecuzione per query GPSJ è composto da 5 tipi di task: Table Scan, Table Scan e Broadcast, Shuffle Join, Broadcast Join, e Group By. In questa sezione sono descritti tali elementi ed i rispettivi costi, che sommati forniscono il costo totale dell'albero in esame.

2.3.1 Statistiche e Stima della selettività

Per poter calcolare il costo dei task è necessario utilizzare i metadati relativi al database in esame, introdotti in questa sottosezione. Una tabella t include un set di attributi $t.Attr$, ha cardinalità $t.Card$ e la sua occupazione in formato non compresso è $t.Size$. HDFS dispone di formati compressi aventi una riduzione media pari ad $fcomp$, compreso fra 0 ed 1. La dimensione media delle partizioni di una tabella immagazzinata in HDFS è $t.PSize$, parametro di HDFS solitamente impostato a 128 MB che può variare nella pratica per tabelle piccole. Ogni attributo $a \in t.Attr$ è caratterizzato da un numero di

valori distinti $a.Card$ ed una lunghezza media in byte $a.Len$.
Prassi consolidate negli RDBMS hanno portato a formule standard per la selettività di un predicato $Sel(t, pred)$, la cardinalità di una join $JCard(t1, t2, pred)$, e la sua dimensione $JSize(t1, t2, pred)$, ma le formule sono molteplici a causa dei diversi predicati ammissibili, e non essendo utile alla discussione dei task successivi un ulteriore livello di dettaglio, queste non sono riportate. Altre formule per la stima di valori nel sistema relazionale sono: Fattore di riduzione $Proj(t, cols)$ sugli attributi $cols$

$$Proj(t, cols) = \frac{\sum_{a \in cols} a.len}{\sum_{a \in t} a.len}$$

Fattore di riduzione a seguito di una Group By, ottenuta sfruttando la formula di Cardenas. Viene assunta una distribuzione uniforme.

$$Group(\#tuples, \#groups) = \frac{\Theta(\#tuples, \#groups)}{\#tuples}$$

$Sel()$, $Proj()$, e $Group()$ valgono 1 se i parametri non sono definiti.

2.3.2 Scan

Il task Scan accede alla tabella t immagazzinata in HDFS. Viene definita la funzione $SC(t, pred, cols, groups, pipe)$, che restituisce il tempo necessario ad eseguire il task. Le operazioni di base effettuate sono:

1. Lettura in memoria delle partizioni di RDD che contengono t;
2. Filtraggio delle tuple in base al predicato $pred$;
3. Rimozione delle colonne inutilizzate;
4. Raggruppamento delle tuple su uno o più campi;
5. Scrittura su disco delle tuple rimanenti.

La prima operazione richiede che vengano lette da HDFS e distribuite agli esecutori le partizioni relative alla tabella t. Questa è l'unica operazione obbligatoria, le successive sono opzionali. Catalyst utilizza il push down delle operazioni, pertanto sia il filtraggio delle tuple che la rimozione delle colonne sono eseguite non appena possibile, eventualmente anche fino al punto da non leggere affatto tuple o colonne non desiderate nel caso di formati di file come Parquet, che grazie ai metadati salvati consentono di accedere solo alle porzioni di interesse. Il push down del raggruppamento può essere utile nella

riduzione della quantità di dati gestiti in seguito, influendo positivamente sui tempi di trasmissione e computazione delle operazioni successive. La scrittura avviene quando il risultato è finale, e viene evitata se una broadcast join è in pipeline (argomento *pipe*=T).
Il numero di partizioni che compongono t è pari alla sua dimensione effettiva su disco fratto quella delle sue partizioni:

$$\#TableP = \frac{t.Size \cdot fcmp}{t.Psize}$$

La dimensione di ogni partizione in memoria è uguale a quella delle partizioni della tabella per i fattori di selettività e proiezione:

$$RSize = t.PSize \cdot Sel(t, pred) \cdot Proj(t, cols)$$

Il numero di ondate da eseguire è uguale all'intero superiore delle partizioni totali dell'RDD fratto quelle che possono essere gestiti in un'ondata, ovvero il numero di core totali:

$$\#Waves = \lceil \frac{\#TableP}{\#E \cdot \#C} \rceil$$

La probabilità che l'ondata non sia locale è pari ai casi in cui gli esecutori sono allocati su nodi che non contengono nessuna delle rf repliche $\binom{\#N-rf}{\#E}$ fratto tutte le possibili allocazioni degli esecutori sui nodi del cluster $\binom{\#N}{\#E}$. La probabilità che l'ondata sia locale è il complementare di quella che non lo sia:

$$P_L = 1 - \frac{\binom{\#N-rf}{\#E}}{\binom{\#N}{\#E}}$$

La probabilità che l'ondata sia esterna al rack richiede una formula complessa, qui affrontata in 3 fasi.
La prima è la probabilità che esattamente x rack abbiano una delle rf repliche di p. Si calcola, tramite il principio di inclusione-esclusione, la probabilità che esattamente x rack abbiano almeno una delle rf repliche, poi moltiplicata per le possibili scelte degli x rack fissati.

$$P_{Part}(x) = \binom{\#R}{x} \cdot \sum_{j=0}^{x} (-1)^j \cdot \binom{x}{j} \frac{\binom{\#RN \cdot (x-j)}{rf}}{\binom{\#N}{rf}}$$

Simmetricamente alla formula appena scritta, quella per la probabilità che esattamente y rack abbiano almeno un esecutore allocato è:

$$P_{Exe}(y) = \binom{\#R}{y} \cdot \sum_{j=0}^{y} (-1)^j \cdot \binom{y}{j} \frac{\binom{\#RN \cdot (y-j)}{\#E}}{\binom{\#N}{\#E}}$$

Date queste due probabilità è possibile definire quella dell'ondata esterna al rack considerando tutti i casi in cui gli y rack in cui sono allocati gli esecutori siano scelti fra i rimanenti $\#R - x$ rack che non hanno partizioni di p. La probabilità che cioè avvenga è $\frac{\binom{\#R-x}{y}}{\binom{\#R}{y}}$, che deve essere valutata per tutti i valori di x ed y, e pesata per le probabilità $P_{Part}(x)$ e $P_{exe}(y)$. La formula risultante è:

$$P_C = \sum_{x=1}^{min(\#R,rf)} \sum_{y=1}^{min(\#R,\#E)} P_{Part}(x) \cdot P_{exe}(y) \cdot \frac{\binom{\#R-x}{y}}{\binom{\#R}{y}}$$

La probabilità che l'ondata sia a livello di rack è il caso in cui non è né locale né esterna al rack:

$$P_R = 1 - P_L - P_C$$

Successivamente alla lettura con gli eventuali filtri è possibile che vi siano operazioni di group by e scrittura su disco. Nel caso in cui non venga effettuata l'operazione di raggruppamento la scrittura viene eseguita in pipeline alla lettura ed alle trasformazioni, altrimenti avviene alla fine dell'operazione di group by. Il tempo totale è quindi dato dal massimo fra tempo di lettura e scrittura nel primo caso, oppure dalla loro somma nel secondo caso.
I byte in scrittura sono pari alla dimensione della tabella per i fattori di selezione, proiezione, e raggruppamento

$$WSize = t.Psize \cdot Sel(t, Size) \cdot Proj(t, cols) \cdot Group(t.Card \cdot Sel(t, pred), \prod_{a \in groups} a.Card) \quad (2.1)$$

Le due possibili formule di SC, con presenza o meno di group by, sono le seguenti:

$$SC(t, pred, cols, groups, pipe) = \lceil \frac{\#TableP}{\#E \cdot \#EC} \cdot \rceil \sum_{X \in \{L,R,C\}} P_X \cdot MAX(Read(RSize, X), Write(WSize)) \quad (2.2)$$

$$SC(t, pred, cols, groups, pipe) = \lceil \frac{\#TableP}{\#E \cdot \#EC} \cdot \rceil \sum_{X \in \{L,R,C\}} P_X \cdot MAX(Read(RSize, X) + Write(WSize)) \quad (2.3)$$

2.3.3 Scan e Broadcast

Il task Scan e Broadcast accede alla tabella t immagazzinata in HDFS e la invia al driver dell'applicazione, che effettua poi il broadcast dell'intera tabella a tutti gli esecutori. La funzione $SB(t, pred, cols)$ restituisce il tempo necessario per eseguire il task.
La fase iniziale di acquisizione della tabella è analoga alla Scan, ma la dimensione delle partizioni è influenzata solo da filtri e proiezioni:

$$BrSize = t.PSize \cdot Sel(t, pred) \cdot Prok(t, cols)$$

L'acquisizione dei dati ed il loro broadcast avviene in pipeline, risultando nella seguente formula:

$$SB(t, pred, cols) = \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \cdot \sum_{X \in \{L,R,C\}} P_X \cdot MAX(Read(RSize, X), Broadcast(BrSize)) \quad (2.4)$$

2.3.4 Shuffle Join

Il task Shuffle Join effettua una join di due tabelle $t1$ e $t2$ le cui partizioni sono state precedentemente hashate in $\#SB$ bucket, ed il cui costo è il risultato della funzione $SJ(t1, t2, pred, cols, groups, pipe)$. In un'ondata le operazioni svolte sono:

1. Shuffle read dei bucket di $t1$ e $t2$,
2. Join locale a carico degli esecutori, che uniscono le tuple tramite il predicato $pred$,
3. Proiezione, viene restituita la tabella con le sole colonne $cols$,
4. Aggregazione delle tuple,
5. Scrittura su disco del risultato.

L'operazione di shuffle read deve essere completata prima dell'inizio della Join, mentre le operazioni di proiezione, aggregazione e scrittura sono opzionali. La quantità di dati letti ad ogni ondata è pari a

$$RSize = \frac{t1.Size + t2.Size}{\#SB}$$

La quantità di dati scritti su disco da ogni core ad ondata è pari a:

$$Wsize = \frac{JSize(t1, t2, pred) \cdot Proj(t1 \bowtie t2, cols)}{\#SB} \cdot Group(JCard(t1, t2, pred), \prod_{a \in groups} a.Card) \quad (2.5)$$

Il costo totale della shuffle join equivale alla somma del tempo di ogni ondata:

$$SJ(t1, t2, pred, cols, groups) = \lceil \frac{\#SB}{\#E \cdot \#EC} \rceil \cdot (Sread(rSize) + Write(WSize)) \quad (2.6)$$

2.3.5 Broadcast Join

La broadcast join esegue una join fra due tabelle $t1$ e $t2$ quando una di queste è di dimensioni sufficientemente ridotte per essere mantenuta completamente nella memoria centrale di ogni esecutore. La funzione che calcola il tempo necessario $BJ(t1, t2, pred, cols, groups, pipe)$ tiene in considerazione solamente la scrittura su disco, in quanto una Broadcast Join avviene solamente in pipeline ad un'altra operazione fra scan, shuffle join o un'altra broadcast join, che implicano la presenza in memoria dei dati necessari. Il numero di ondate dipende dalle partizioni di $t2$, e la quantità di ogni ondata per core scritta su disco è:

$$WSize = \frac{JSize(t1, t2, pred) \cdot Proj(t1 \bowtie t2, cols)}{t2.Part} \cdot Group(JCard(t1, t2, pred), \prod_{a \in groups} a.Card) \quad (2.7)$$

Il tempo complessivo dell'operazione è la somma dei tempi relativi alle diverse ondate:

$$BJ(t1, t2, pred, cols, groups) = \lceil \frac{t2.Part}{\#E \cdot \#EC} \rceil \cdot Write(WSize)$$

2.3.6 Group By

La funzione $GB(t, pred, cols, groups)$ restituisce il tempo necessario per eseguire l'operazione di group by, tramite i seguenti passi:

1. Shuffle read dei bucket da t, una porzione di bucket per ogni esecutore,

2. Aggregazione delle tuple aventi stessi valori degli attributi *groups* e le operazioni di aggregazioni sulle rimanenti colonne *cols*,

3. Scrittura su disco delle tuple rimanenti.

L'aggregazione non può avvenire fino alla completa lettura dei bucket di t, quindi il tempo per eseguire un'ondata è uguale alla somma dei tempi per la lettura e scrittura (quest'ultima avviene in pipeline con l'aggregazione). I dati che ogni core deve leggere sono:

$$RSize = \frac{t.Size}{\#SB}$$

La quantità di dati scritti su disco da ogni core in ogni ondata è:

$$WSize = RSize \cdot hSel \cdot Proj(t, cols) \cdot Group(t.Card, \prod_{a \in groups} a.Card)$$

dove $hSel$ è un fattore di selettività, pari a 0.33 di default ed uguale ad 1 se il predicato pred non è definito. Questo numero è una stima standard utilizzata in sistemi commerciali. Il tempo complessivo per la Group By è la somma dei tempi delle ondate:

$$GB(t, pred, cols, groups) = \lceil \frac{\#SB}{\#EC \cdot \#EC} \rceil \cdot (SRead(RSize) + Write(WSize))$$

Capitolo 3

Progetto

Questo capitolo tratta il progetto sviluppato, prima introducendo l'applicazione Web dove esso è integrato, successivamente descrivendone i vari moduli e la loro integrazione attraverso l'interfaccia grafica unificata.

3.1 Sparktune

Sparktune è un'applicazione web realizzata dal gruppo di ricerca Business Intelligence Group[1], che consente di calcolare il tempo di esecuzione di una Query in SparkSQL secondo il modello descritto nel capitolo precedente. Allo stato iniziale l'applicazione richiedeva all'utente di inserire manualmente tutti i parametri necessari al calcolo del costo, riportati nella tabella 3.1, che saranno spiegati nelle sezioni successive. I parametri di configurazione di Spark ($\#RE$, $\#E$, ed $\#EC$) sono scelti dall'utente, ed in quanto tali devono essere comunque forniti da esso e non saranno recuperati automaticamente dal sistema. I problemi di delegare all'utente la raccolta dei parametri sono:

- Difficoltà di recupero di tutte le informazioni, specialmente quelle relative al throughput ed alle statistiche del database;
- Rischio che le informazioni vengano inserite in modo errato;
- Bassa riusabilità da parte di un utente poco esperto.

Delegare all'utente questo lavoro implica la possibilità di procedure non standard per l'acquisizione di certi dati, che soprattutto nel caso del throughput del disco potrebbero generare valori non corretti, ad esempio campionando tramite strumenti esterni, senza tenere conto della serializzazione e compressione effettuate da Spark. Altri problemi minori sono relativi alla user experience,

[1] http://big.csr.unibo.it/

ed all'abilitazione di utenti non esperti ad usufruire del servizio, nell'ottica di un ampio utilizzo dell'applicazione web. Il progetto sviluppato per questa tesi consiste in una funzionalità aggiuntiva per automatizzare l'acquisizione delle informazioni richieste dal modello di costo riguardanti il cluster.

Architettura

Allo stato iniziale del lavoro di tesi, il progetto consta di un classico stack XAMP: Web server Apache con backend in PHP collegato ad un database MySQL. La piattaforma big data di riferimento è un cluster di 11 macchine da 8 core ciascuna, sulle quali è installata la CDH (Cloudera Distribution of Hadoop) 5.10, una distribuzione di Hadoop fornita da Cloudera[2].

Nelle sezioni successive vengono descritte le modalità di acquisizione dei parametri descritti nel capitolo precedente.

3.2 Topologia del cluster

Le informazioni sulla topologia riguardano la composizione strutturale del cluster: i rack che lo compongono, i nodi che vi appartengono, e le loro caratteristiche hardware. Hadoop mette a disposizione alcune Web API all'indirizzo IP del NameNode, per ottenere informazioni su topologia ed attività in esecuzione. Per accedere a queste informazioni è sufficiente essere in grado di raggiungere a livello di rete il NameNode, ed effettuare la GET all'indirizzo desiderato. Per ottenere tutte le informazioni di questo modulo è stato sufficiente utilizzare l'API con indirizzo *http://nameNodeIP:8088/ws/v1/cluster/nodes*, che restituisce un JSON contenente un array di oggetti relativi ai nodi del cluster, ognuno dei quali contiene le seguenti informazioni: IP o hostname, rack di appartenenza, numero di core. L'array JSON è manipolato tramite codice php per estrapolare i seguenti dati:

- $\#R$ numero dei valori distinti relativi al rack di appartenenza,
- $\#N$ numero di nodi,
- $\#RN$ rapporto fra $\#N$ ed $\#R$,
- $\#C$ calcolando la media del numero di core per nodo.

[2]https://www.cloudera.com/

Parametro	Descrizione	Tipo
$\#R$	Numero di rack del cluster	Topologia
$\#RN$	Numero di nodi in ogni rack	
$\#N$	Numero di nodi del cluster ($\#R \cdot \#RN$)	
$\#C$	Numero di core per nodo	
$\delta_r(\#Proc)$	Velocità di lettura da disco in MB/s in funzione del numero di processi concorrenti	Performance
$\delta_w(\#Proc)$	Velocità di scrittura su disco in MB/s in funzione del numero di processi concorrenti	
$\rho_i(\#Proc)$	Velocità di rete in MB/s per nodi dello stesso rack in funzione del numero di processi concorrenti	
$\rho_e(\#Proc)$	Velocità di rete in MB/s per nodi di rack diversi in funzione del numero di processi concorrenti	
$\#SB$	Numero di bucket per lo shuffling	
$sComp$	Fattore di compressione per la trasmissione su rete	
$fComp$	Fattore di riduzione per il salvataggio su disco	
$hSel$	Fattore di selettività per le clausole having nella query	
$\#RE$	Numero di esecutori allocati per l'applicazione Spark in ogni esecutore	Configurazione Spark
$\#E$	Numero di esecutori totali allocati per l'applicazione Spark ($\#RE \cdot \#R$)	
$\#EC$	Numero di core per ogni esecutore allocato all'applicazione Spark	
$t.Attr$	Set di attributi nella tabella t	Metadati
$t.Size$	Dimensione in MB della tabella t salvata in formato non compresso	
$t.PSize$	Dimensione media in MB delle partizioni di RDD per la tabella t	
$t.Card$	Numero di tuple nella tabella t	
$t.Part$	Numero di partizioni di cui è composta la tabella t	
$a.Card$	Numero di valori distinti per l'attributo a	
$a.Len$	Lunghezza media in byte dell'attributo a	

Tabella 3.1: Tabella complessiva dei parametri utilizzati dal modello di costo.

Nel modello di costo viene assunta una distribuzione uniforme dei nodi fra i rack, e di processori fra i nodi, anche se nella realtà è possibile che ciò non accada. Per questioni di coerenza con il modello di costo in Sparktune si calcola una media di entrambi i valori, e si tiene conto dell'intero superiore.

3.3 Performance del cluster

Le performance del cluster riguardano le velocità di disco e di rete, calcolate a livello di singolo nodo e di singola connessione, sia esternamente che internamente al cluster. Per la raccolta di queste informazioni si è scelto di adottare un approccio basato sui dati: invece di effettuare calcoli teorici a partire dalle caratteristiche tecniche di disco, processore, e sistema operativo, si è deciso di eseguire dei test direttamente sulle macchine del cluster per raccogliere dei risultati reali. Attraverso la realizzazione di un test in SparkSQL sono stati raccolti dati applicabili alle diverse configurazioni applicative. Tale test può essere eseguito su un cluster qualsiasi, non richiede permessi eccessivamente invadenti, evita di lasciare tracce sul cluster di esecuzione, ed è indipendente dagli aggiornamenti di SparkSQL.

3.3.1 Disk Throughput

L'obiettivo è quello di calcolare i valori delle funzioni $\delta_r(\#Proc)$ e $\delta_w(\#Proc)$. In particolare, vista la possibilità di avere più processi paralleli che accedono contemporaneamente al disco, si vuole verificare come cambiano le performance all'aumentare nel numero di processi concorrenti. Per ottenere un risultato rilevante è necessario utilizzare le funzioni di Spark. La complessità del task è data dalla caratteristica di Spark di ottimizzare il carico di lavoro massimizzando il pipeling di operazioni che possono essere eseguite sullo stesso nodo e senza dover fare shuffling. Sebbene questa sia una delle caratteristiche più importanti di Spark, ciò rende difficile discernere i tempi di esecuzione relativi a letture e scritture su disco da quelli dell'utilizzo della CPU. Non sarebbe corretto l'altro estremo, effettuando una lettura o scrittura da disco con le sole primitive di sistema operativo, in quanto è necessario incorporare il tempo di decompressione e quello di deserializzazione (o quelli di compressione e serializzazione nel caso della scrittura).

Il tempo calcolato da $\delta_r(\#Proc)$ è dato dal tempo che intercorre dalla richiesta di lettura a quando l'RDD è in memoria, pronto per l'utilizzo, mentre quello di $\delta_w(\#Proc)$ è il tempo dalla richiesta di scrittura di un RDD già presente in memoria a quello in cui i file che compongono le partizioni dell'RDD sono tutte scritte su disco, pronte per l'accesso. Si procede a presen-

tare l'approccio finale utilizzato, lasciando alla conclusione di questa sezione la discussione degli esperimenti che non hanno avuto un risultato adeguato.

La soluzione adottata: script Scala

Per il test delle performance ci si è avvalsi dell'utilizzo di uno script Scala per Spark, configurando l'ambiente di esecuzione per essere locale. Spark mette a disposizione sia la Spark Shell, interprete interattivo, che la Spark Submit, per applicazioni già compilate e comprensive di eventuali dipendenze. Le opzioni rilevanti all'esecuzione dei test trattati sono:

- *master* indica il nodo Master, può avere valore local;
- *deploy mode* specifica se il processo driver è eseguito sul nodo dal quale è richiesta l'esecuzione dell'applicazione (valore *client*), oppure su un nodo qualsiasi (valore *cluster*);
- *executor-cores* numero di core allocati per nodo esecutore;
- *executor-memory* memoria dedicata all'applicazione per nodo esecutore, utile per evitare swap su disco.

Impostando il master locale e la deploy mode a client si ottiene l'esecuzione del Job di Spark su una sola macchina, condizione adeguata per eseguire il test di benchmark. Lo script viene eseguito più volte, una per numero di core, che varia da 1 a $\#C$, per calcolare i diversi valori delle funzioni. La memoria degli esecutori è impostata ad un valore tale per cui le strutture utilizzate per il benchmark siano completamente contenute in memoria, ed è ottenuto tramite un processo di tuning basato sulle statistiche di esecuzione.

La soluzione al problema consiste in uno script che utilizza SerializerManager, una classe di Spark che si occupa di serializzare e salvare su disco un file, oppure di leggerlo e deserializzarlo. Tale classe non è solitamente utilizzata da applicazioni utente di alto livello, che delegano questo genere di dettagli al framework. Si è deciso di utilizzare tale classe in seguito all'ispezione del codice di Spark, qui discussa.

Funzionamento interno dell'attività di Persist

La classe RDD implementa la funzione *persist(newLevel)*, che si limita ad impostare il valore di una variabile, senza effettuare alcuna azione coerentemente al principio di laziness. Il flag viene preso in considerazione durante l'iterazione fra le partizioni dell'RDD, delegando il lavoro BlockManager, la

cui istanza è ottenuta tramite il singleton SparkEnv, che mantiene i riferimenti ai Manager dell'ambiente. La partizione dell'RDD può essere assente o presente, locale oppure remota, ed in memoria o su disco. La scrittura su disco avviene quando l'RDD è locale ma assente su disco, mentre la lettura avviene quando l'RDD è presente in locale e su disco. Nel caso della lettura viene creato l'oggetto di classe BlockData che incapsula le informazioni riguardanti l'accesso su disco della partizione, successivamente questa viene passata al SerializerManager, che fornisce un iteratore come risultato. Per la scrittura il processo è simile, ma è necessario operare il boxing del file in un *BufferedOutputStream* per poterlo passare al SerializerManager. Una versione semplificata del diagramma delle classi utilizzate è presentata di seguito.

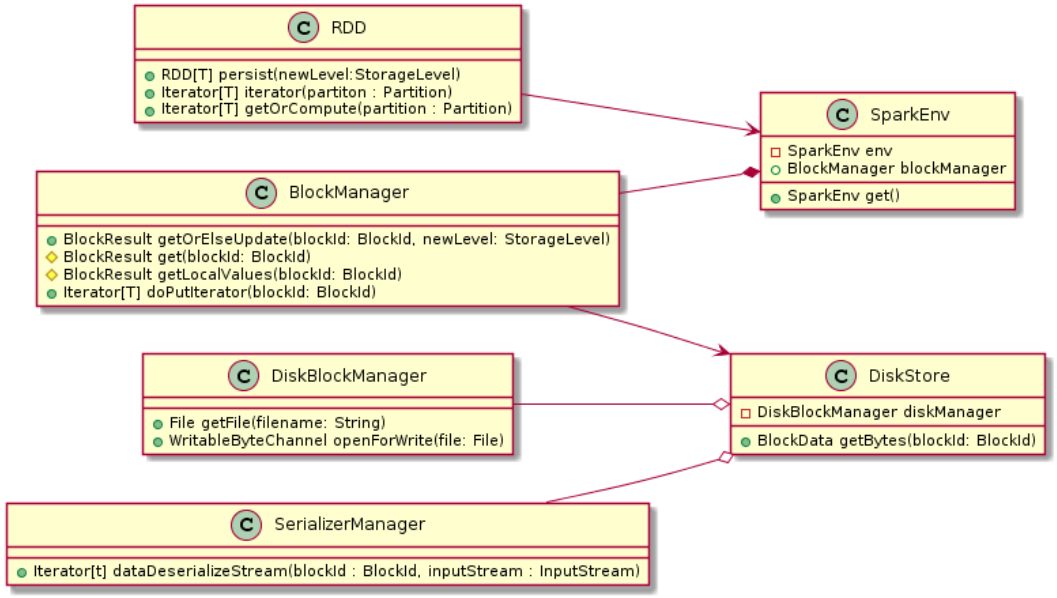

Figura 3.1: Schema delle classi principali utilizzate da Spark per la persistenza degli RDD

A partire dai passi eseguiti da Spark si è costruito un modello semplificato per rimuovere tutti i possibili overhead relativi alla laziness, già presa in considerazione dal modello di costo, mantenendo le operazioni da eseguire. In particolare la locazione sul file system è gestita tramite le API di Java, e le operazioni di serializzazione, compressione e scrittura sono delegate al SerializationManager

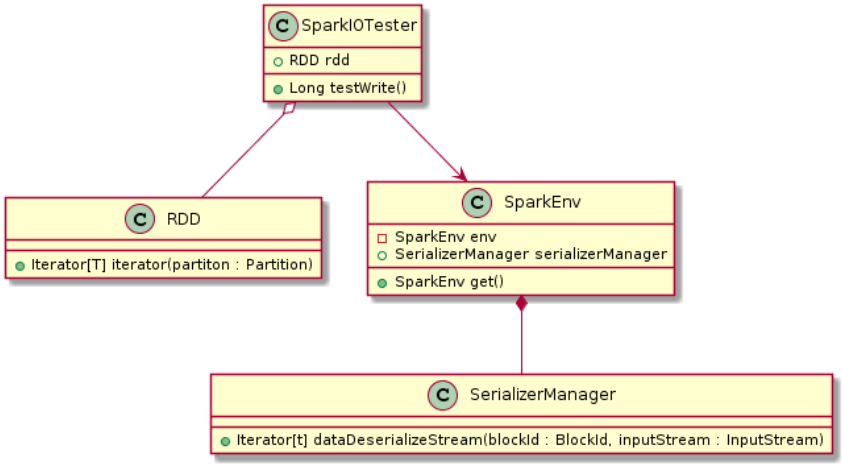

Figura 3.2: Schema delle classi utilizzate nello script per il test dell'IO

Eseguendo lo script sul cluster si ottiene un risultato del tipo rappresentato in 3.3

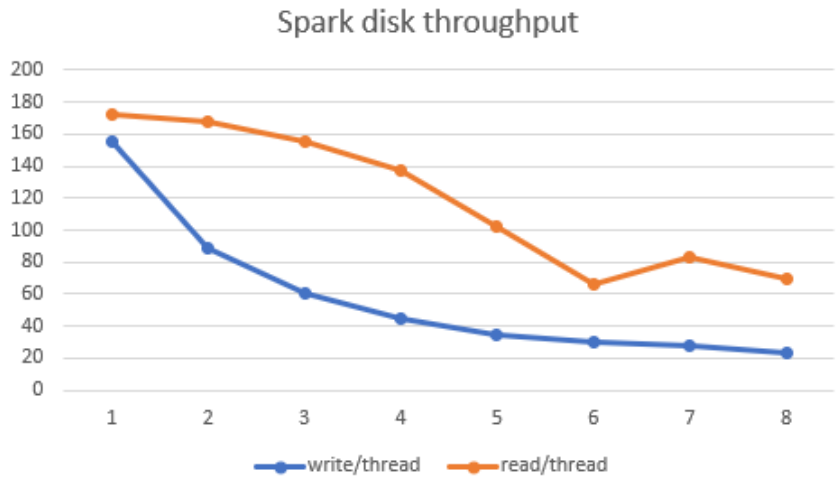

Figura 3.3: δ_r e δ_w in base al numero di processi

La curva della scrittura può essere rappresentata anche come $\delta_w(n) = \frac{\delta_w(1)}{n}$, in quanto il collo di bottiglia è il disco ed incrementando il numero di processi concorrenti la velocità non aumenta. Il caso della lettura presenta invece un notevole miglioramento dei tempi per i primi thread aggiuntivi, implicando che la velocità di lettura è superiore a quella di decompressione e deserializzazione di un singolo thread, per cui il disco non rimane in attesa di essere letto mentre un processore svolge le operazioni sull'RDD letto, ma è assegnato ad un ulteriore processo parallelo al precedente. Oltre un determinato numero

di thread il comportamento diventa non predicibile, con spike probabilmente causati dalle strategie di scheduling del sistema operativo della macchina.

I tentativi precedenti: cosa non ha funzionato e perchè

Prima di raggiungere questo risultato sono stati eseguiti vari altri test, due dei quali sono rilevanti al fine di comprendere il funzionamento del sistema Spark e le conclusioni che hanno portato al risultato finale.

Un primo approccio prevedeva la generazione in memoria un RDD arbitrario, per poi salvarlo su disco, chiudere l'applicazione per evitare processi di cacheing, ed eseguire un secondo script per la lettura del/dei file. In fase di analisi delle API di Spark si è constatato che le modalità di generazione dell'RDD sono molto varie, così come i formati di file, e le strategie di serializzazione e compressione. In SparkSQL non esiste uno standard relativo a quali opzioni usare, in quanto queste dipendono dalle impostazioni scelte dall'utente, pertanto testare tutte le possibili combinazioni risulterebbe eccessivamente oneroso. In alternativa si potrebbe richiedere all'utente di indicare tali informazioni, perdendo i vantaggi della automatizzazione. In ultimo un approccio simile è molto sensibile ai cambiamenti implementativi di Spark, e con le versioni successive del software il funzionamento interno di SparkSQL potrebbe utilizzare funzionalità diverse ed il throughput calcolato in maniera rigida non lo rispecchierebbe.

Considerata la natura data driven del test su disco si è deciso di utilizzare un database di benchmark – tpch [3] – in modo da testare con dati sufficientemente eterogenei e delle dimensioni adeguate. Scelta una tabella questa deve essere caricata in memoria, successivamente scritta su disco e poi riletta. L'ultima lettura avviene in quanto la prima non è significativa, dato che la tabella è inizialmente distribuita su diversi nodi del cluster, e tempistiche di rete sono incorporate in tale operazione. Ad applicazione avviata si effettua la connessione al database tpch_10gb, viene poi eseguita la query sql *select * from lineitem limit 15000000*. Nonostante l'esecuzione del job Spark sia locale l'esecutore è ancora connesso alla rete, e può utilizzare le funzionalità messe a disposizione da Hadoop, in particolare la query che richiede la lettura della tabella lineitem implica l'utilizzo dei vari nodi sui quali è distribuita, utilizzando il disco di altri nodi e la rete. L'operazione è volta quindi unicamente ad ottenere nella memoria centrale del nodo esecutore una copia della porzione di tabella di interesse, ma a causa della lazy evaluation ciò avviene solo a fronte dell'operazione successiva di count. L'utilizzo di un'azione è necessaria per innescare la lettura effettiva, e fra le varie operazioni messe a disposizione da Spark, count - da eseguire in pipeline - è quella che presenta il minor overhead. La stessa operazione seguente di persist è considerata lazy, e bisogna effettuare

una azione per ottenere il risultato. Questa operazione risulta problematica in quanto le azioni prevedono che l'RDD venga prima scritto su disco e poi vengono effettuate le altre operazioni in pipeline alla lettura, non fornendo il tempo di scrittura richiesto, ma la sua somma con il tempo di lettura.

Per analizzare l'esecuzione dello script e verificare la correttezza delle affermazioni precedenti, ci si è avvalsi di una rappresentazione del throughput durante l'esecuzione fornita da Cloudera, che mette a disposizione diversi grafici per il monitoraggio del cluster. La prima operazione di count avviene alle 7:52; Nel primo grafico 3.4 si nota un picco notevole nell'utilizzo del disco sui diversi nodi del cluster, mentre nel nodo locale, come indicato in 3.5 vi è un massimo correlato alle letture precedenti, e di breve durata. Dal punto di vista della rete sul cluster (3.6) si raggiunge un picco di trasmissione e ricezione di uguale valore, in quanto il nodo esecutore riceve tutti i dati, come mostrato dal primo picco in ricezione in 3.7. Alle 7:55 viene eseguita la seconda count, che attiva la persistenza su disco, e per quanto sul disco dell'esecutore vi sia un picco considerevole di scrittura, la presenza di I/O di rete non trascurabile indica che parti dell'RDD sono state salvate anche su dischi remoti.

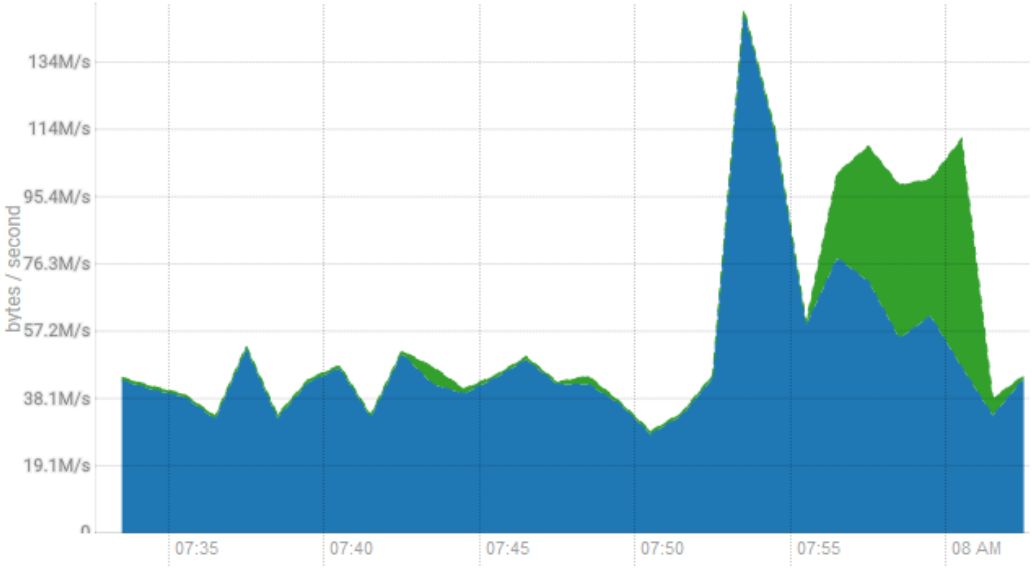

Figura 3.4: Utilizzo del disco a livello di cluster. In blu velocità lettura, in verde di scrittura.

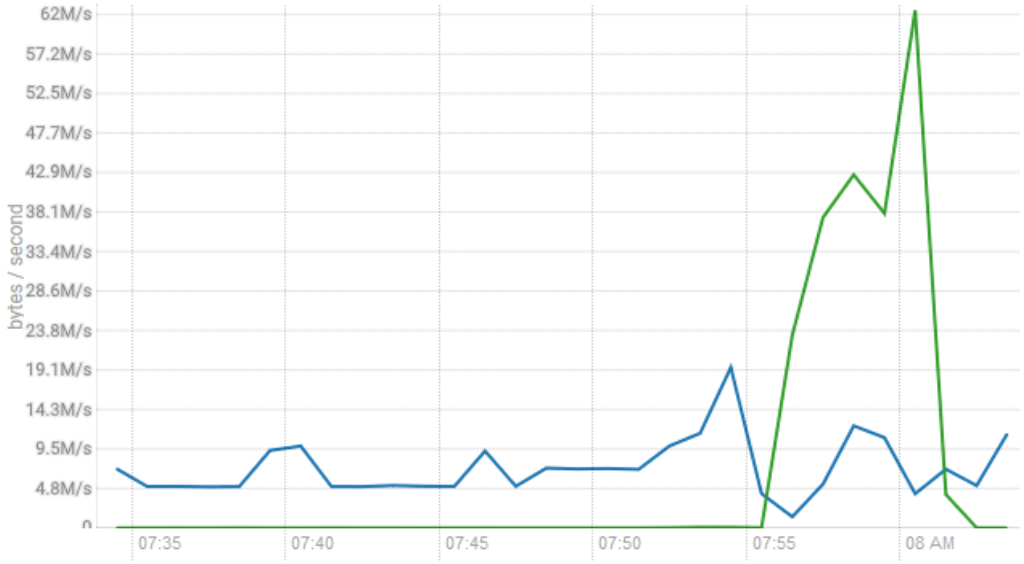

Figura 3.5: Utilizzo del disco sul nodo esecutore. In blu velocità lettura, in verde di scrittura

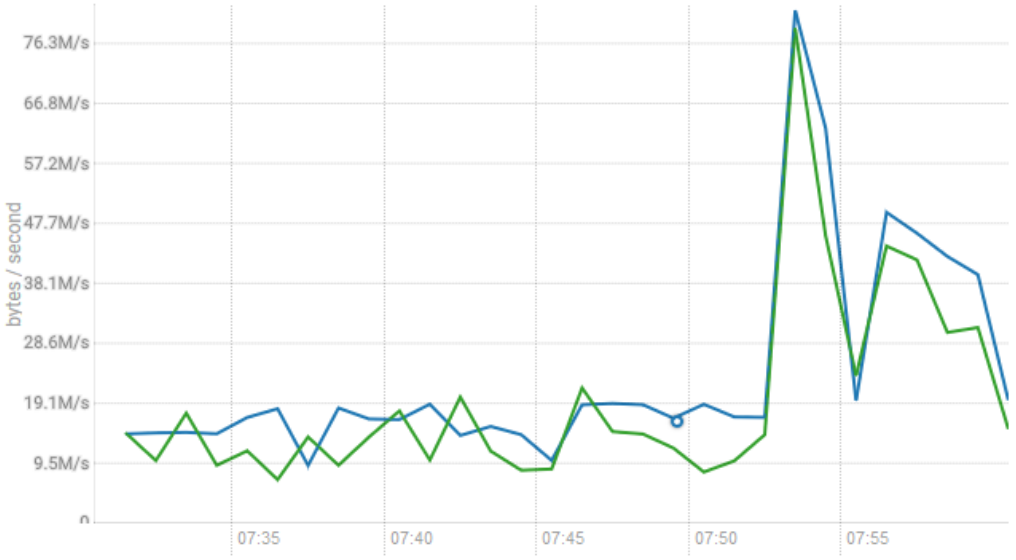

Figura 3.6: Utilizzo della rete a livello di cluster. In blu velocità lettura, in verde di scrittura

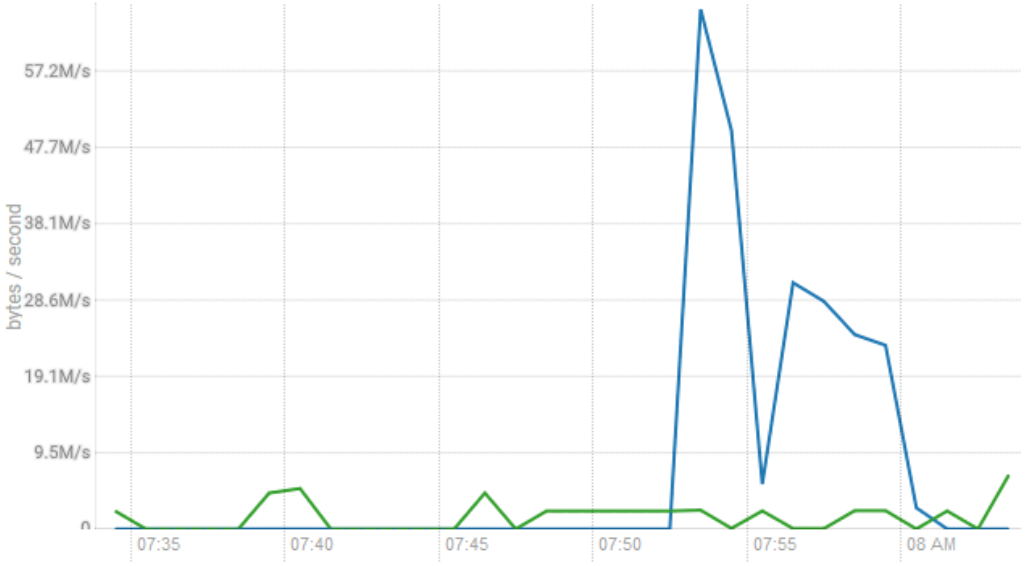

Figura 3.7: Utilizzo della rete sul nodo esecutore. In blu velocità lettura, in verde di scrittura

Data la mancanza di ulteriori funzionalità di alto livello e l'inadeguatezza di quelle messe a disposizione da Spark per un'operazione di basso livello come il benchmark del disco in maniera precisa, si è optato per studiare approfonditamente l'implementazione interna di Spark, data la sua natura open source, per individuare le operazioni eseguite a fronte della chiamata della funzione persist, allo scopo di riprodurre solo la porzione di scrittura su disco incorporando tempi di serializzazione e compressione, senza overhead per action, funzioni indesiderate in pipeline, o altre ottimizzazioni che potrebbero non essere state prese in considerazione in quanto meccanismi interni non documentati ad alto livello, arrivando alla soluzione presentata sopra.

3.3.2 Network throughput

Il processo di benchmark della rete avviene tramite l'utility di sistema operativo netcat, che consente di istanziare semplici server e client per il trasferimento di dati. L'input per il test di rete sono le credenziali di tre nodi diversi del cluster, di cui due sullo stesso rack ed uno su un rack esterno, per poter testare la velocità fra rack. Viene stabilita una connessione SSH su tutte e tre le macchine: sulle due appartenenti allo stesso rack vengono lanciate rispettivamente un server ed un client netcat, mentre su quella esterna solo un client. Vengono eseguiti due trasferimenti di 1 GB di dati, prima internamente

al rack, e poi esternamente. Il risultato viene scritto su un file di testo parsato per estrarre l'informazione finale sulla velocità, che corrisponde al throughput massimo su una connessione del tipo relativo, da dividere per il numero di processi concorrenti $\#Proc$ per ottenere il dato ricercato.

3.4 Dati MetaStore

L'obiettivo di questo modulo è recuperare i metadati sul database per il quale si vuole calcolare il costo delle query, in particolare quelli relativi alla dimensione delle tabelle e delle partizioni in cui sono suddivise. I Metadati sono salvati su un Hive MetaStore configurato in modalità remota. Il MetaStore è un database – PostgreSQL nel caso studiato – dove i metadati relativi ai database Hive del cluster sono memorizzati secondo uno schema relazionale, del quale si propone una versione semplificata di seguito:

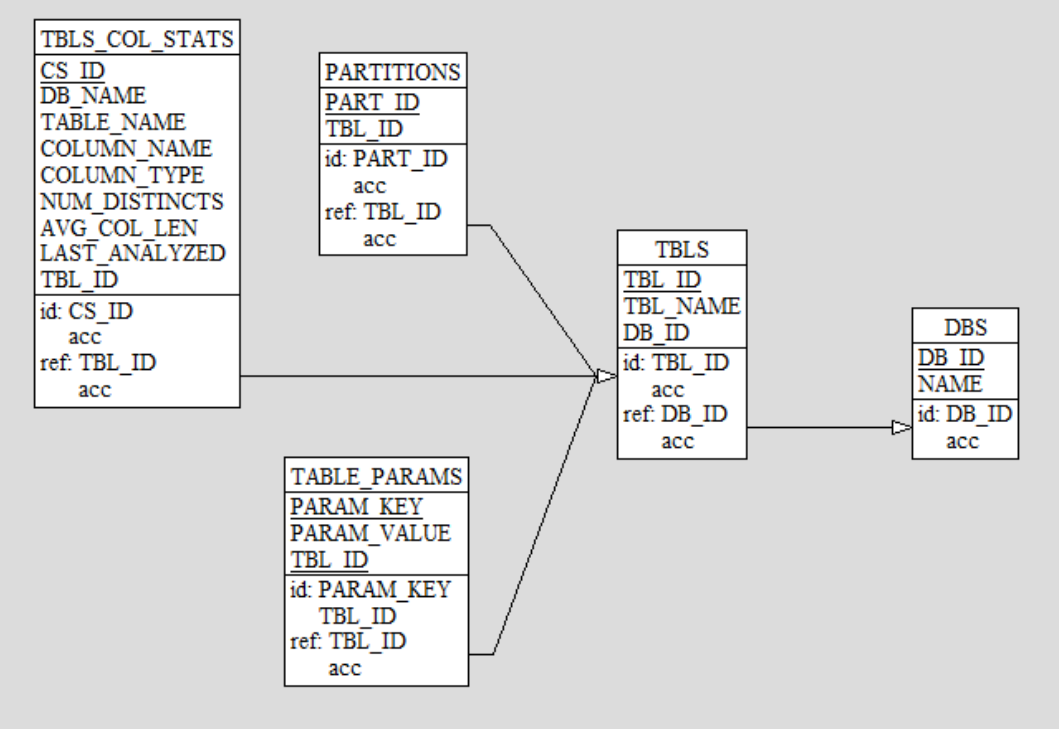

Figura 3.8: Schema ER semplificato del database MetaStore relativo ad un cluster.

La prima tabella di interesse è DBS, le cui tuple corrispondono ai diversi database immagazzinati su Hive. Selezionato il Database, è possibile risalire a tutte le tabelle che vi appartengono, tramite la tabella del MetaStore $TBLS$.

Il parametro $t.Attr$ si ottiene interrogando la tabella TAB_COL_STATS, identificando le tuple di interesse tramite chiave esterna, ed ottenendo in output nomi e tipi degli attributi. Successivamente si interroga la tabella $TABLE\_PARAMS$, che espone uno schema a tre colonne: una per identificare la tabella di appartenenza, e due che fungono da chiave e valore, in questo modo è possibile salvare molti parametri diversi, eventualmente anche aggiungendo informazioni senza modificare lo schema. In particolare i valori di interesse sono quelli aventi chiave

- $totalSize$ che corrisponde direttamente a $t.Size$
- $numRows$ che corrisponde direttamente a $t.Card$
- $numFiles$ che corrisponde direttamente a $t.Part$

Per ottenere il numero di partizioni $t.Part$ è necessario interrogare la tabella $PARTITIONS$, raggruppando per ID di tabella ed applicando l'operazione di aggregazione count. Conoscendo $t.Size$ e $t.Part$ è possibile calcolare $t.PSize$ come il rapporto fra i due valori. I parametri rimanenti sono relativi agli attributi, entrambi presenti come colonne della tabella TAB_COL_STATS, in particolare i valori distinti $a.Card$ sono espressi nella colonna $NUM\_DISTINCTS$, e la lunghezza media $a.Len$ è AVG_COL_LEN.

La problematica principale relativa a questo componente consiste nell'eventualità che il MetaStore non sia popolato o aggiornato. In entrambi i casi la tabella di riferimento è TAB_COL_STATS, la cui colonna $LAST\_ANALYZED$ fornisce il dato sul quale decidere se le informazioni sono sufficientemente aggiornate oppure se queste necessitino di essere ricalcolate. Nel caso specifico del database utilizzato per il cluster sul quale è stato eseguito il test, è sufficiente utilizzare i comandi *analyze table tableName compute statistics;* ed *analyze table tableName compute statistics for columns;* da shell Hive.

3.5 Interfaccia Utente

I diversi elementi sono integrati nell'applicazione Web Sparktune, tramite un'interfaccia dedicata. Sono presenti tre moduli principali:

- Topologia
- Performance
- Metadati Database

La pagina propone tre schermate, una per modulo, navigabili secondo il diagramma delle sequenze 3.9

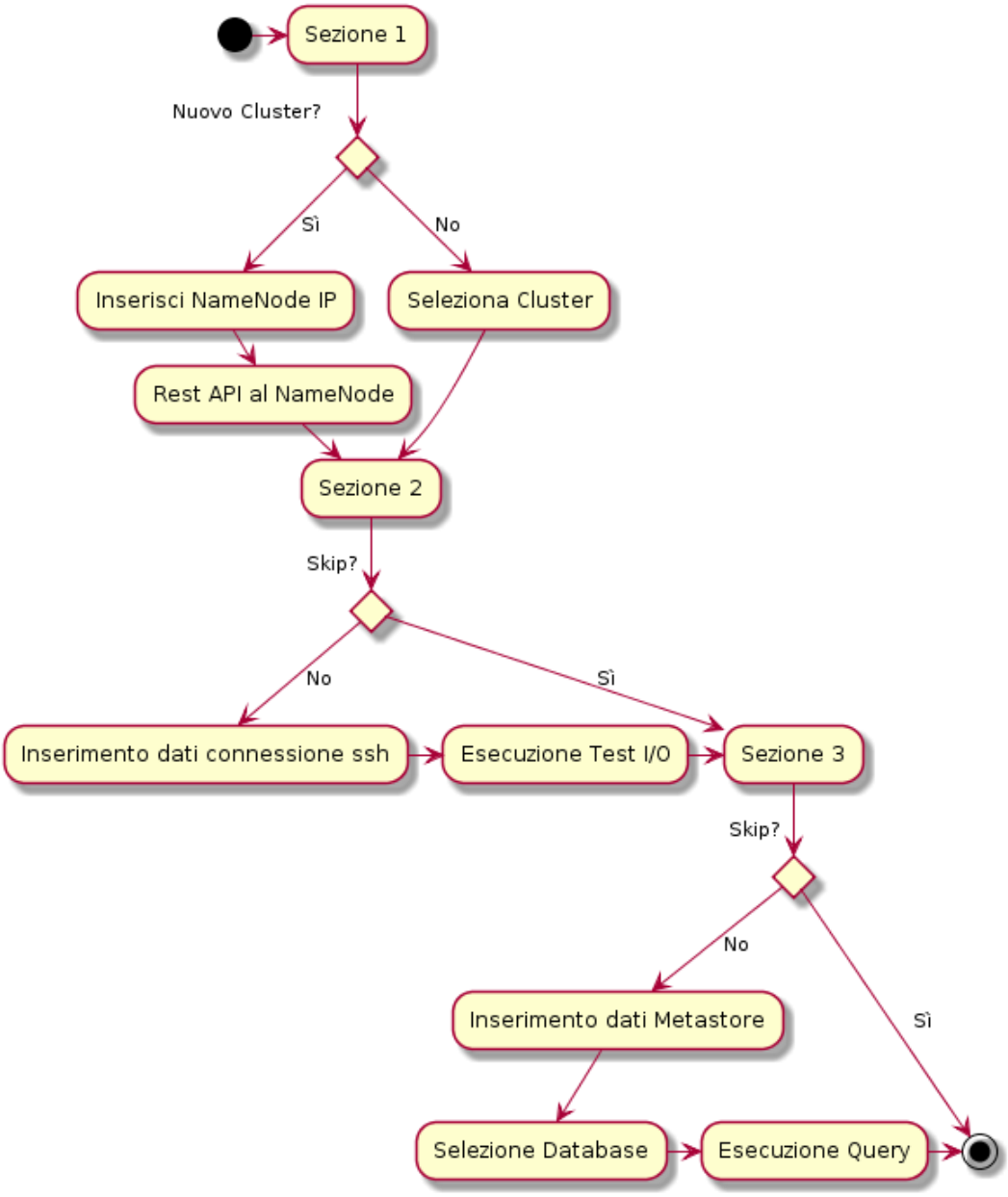

Figura 3.9: Sequence Diagram dell'interfaccia utente

Ogni modulo è eseguito separatamente ed indipendentemente dagli altri, con il vincolo di selezionare il cluster di riferimento nella prima schermata. Ogni modulo prevede l'acquisizione di input dall'utente, l'esecuzione di codi-

ce php lato server, e termina con l'inserimento nel Database sparkcost delle informazioni ottenute.

La prima schermata 3.10 richiede se il cluster è già presente o meno, ed in base all'opzione selezionata fornisce una lista dei cluster già presenti oppure richiede l'immissione dell'IP del NameNode del nuovo cluster. Le informazioni sulla topologia sono ottenute tramite la funzione php *fileGetContent*, che fornisce il risultato in formato JSON, e consente di gestire un possibile errore di connessione. L'input della funzione è l'URL della risorsa, ovvero *http://nameNodeIP:8088/ws/v1/cluster/nodes*, risulta quindi sufficiente l'IP del NameNode.

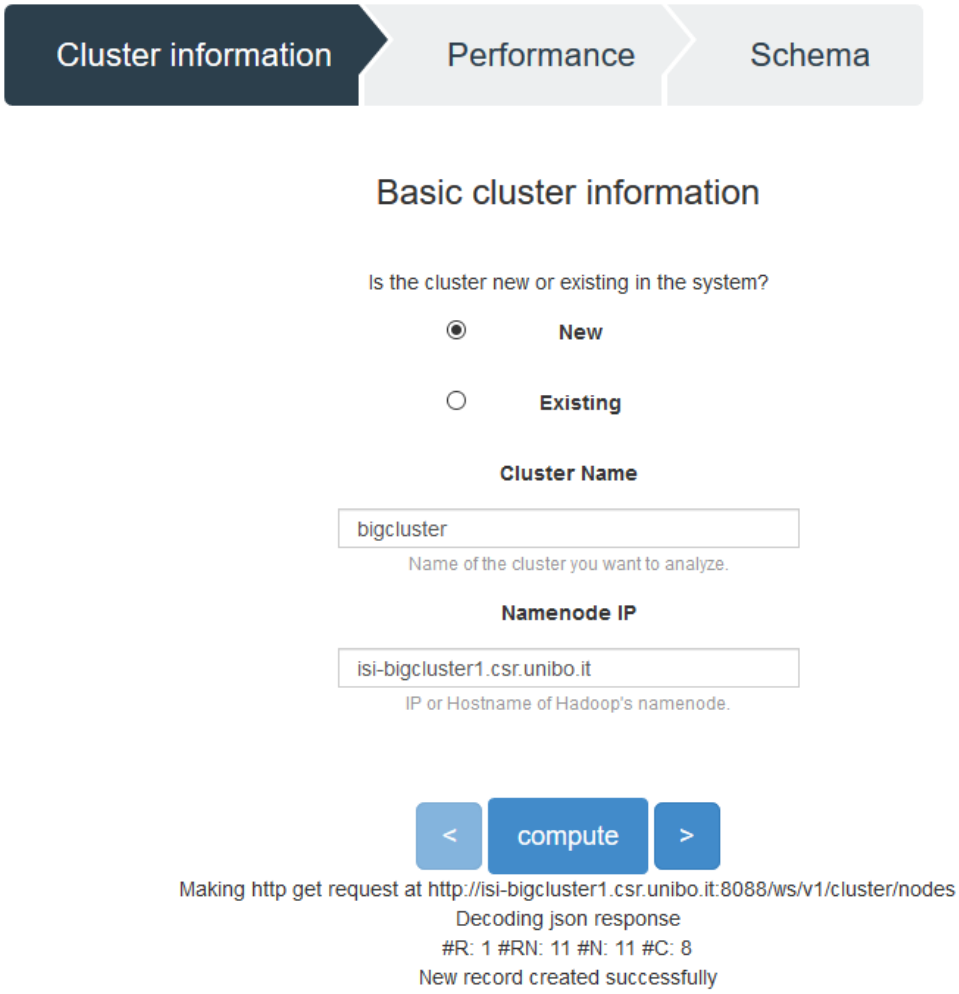

Figura 3.10: Prima schermata dell'interfaccia utente

La seconda schermata 3.11 ottiene dalla precedente una lista di indirizzi dei nodi del cluster, e richiede di selezionarne uno per il disco e la rete, uno

per rete intra rack, ed uno per rete extra rack, e fornire i dati di accesso per effettuare una connessione ssh. Dal punto di vista server la connessione è instaurata utilizzando la libreria phpseclib[3], compatibile con versioni di php superiori alla 5. Nel caso di problemi di autenticazione il sistema richiede il reinserimento dei dati corretti.

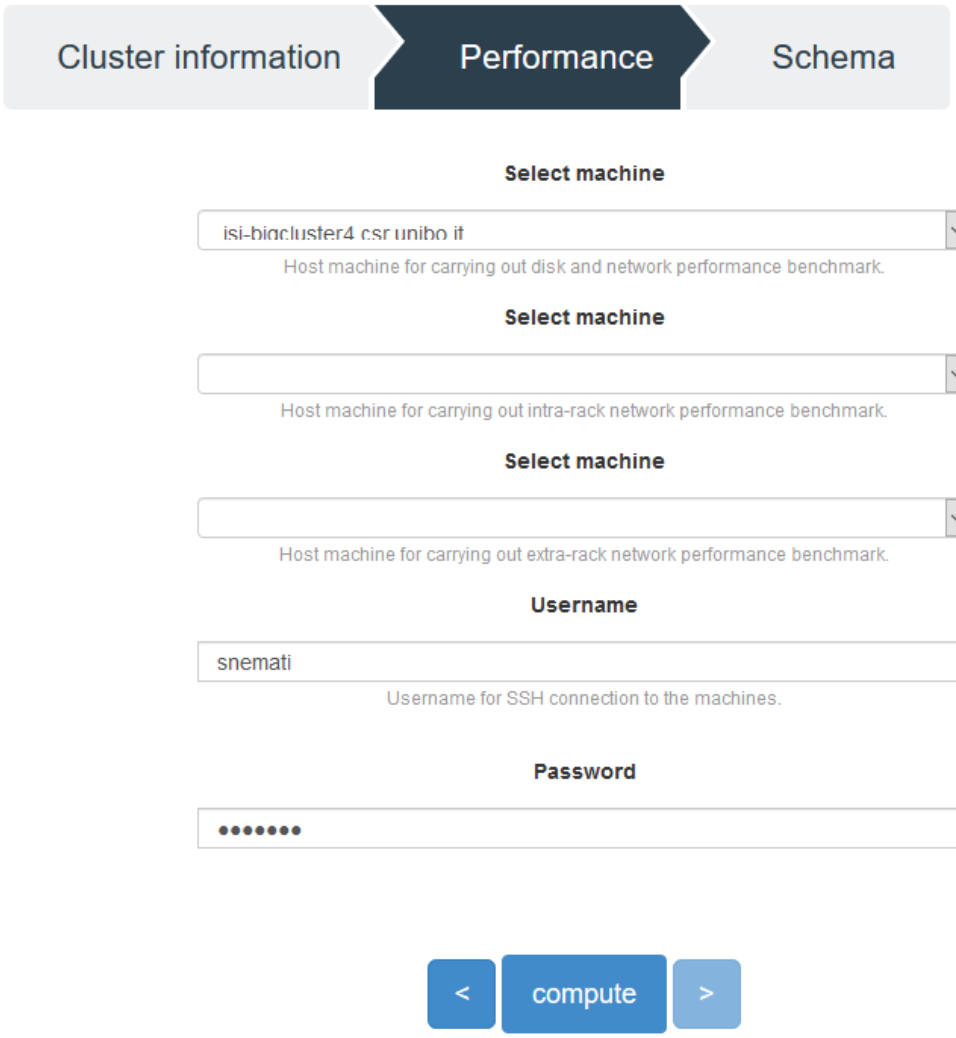

Figura 3.11: Seconda schermata dell'interfaccia utente

La terza schermata 3.12 è utilizzata per raccogliere i dati dal Metastore, che richiedono due fasi per essere raccolti: prima viene inizializzata una connessione al MetaStore, e vengono recuperati i nomi e gli ID di tutti i database presenti. Nella seconda fase viene selezionato un Database, per il quale ven-

[3]https://github.com/phpseclib/phpseclib

gono eseguite le query necessarie a reperire i metadati necessari per il modello di costo. L'input utente consiste nell'indicare IP, porta, e credenziali per l'accesso al MetaStore per la prima fase, e semplicemente indicare il Database di interesse nella seconda. Esistono due possibili tipi di errori: l'impossibilità di connessione e l'assenza dei metadati sul MetaStore.

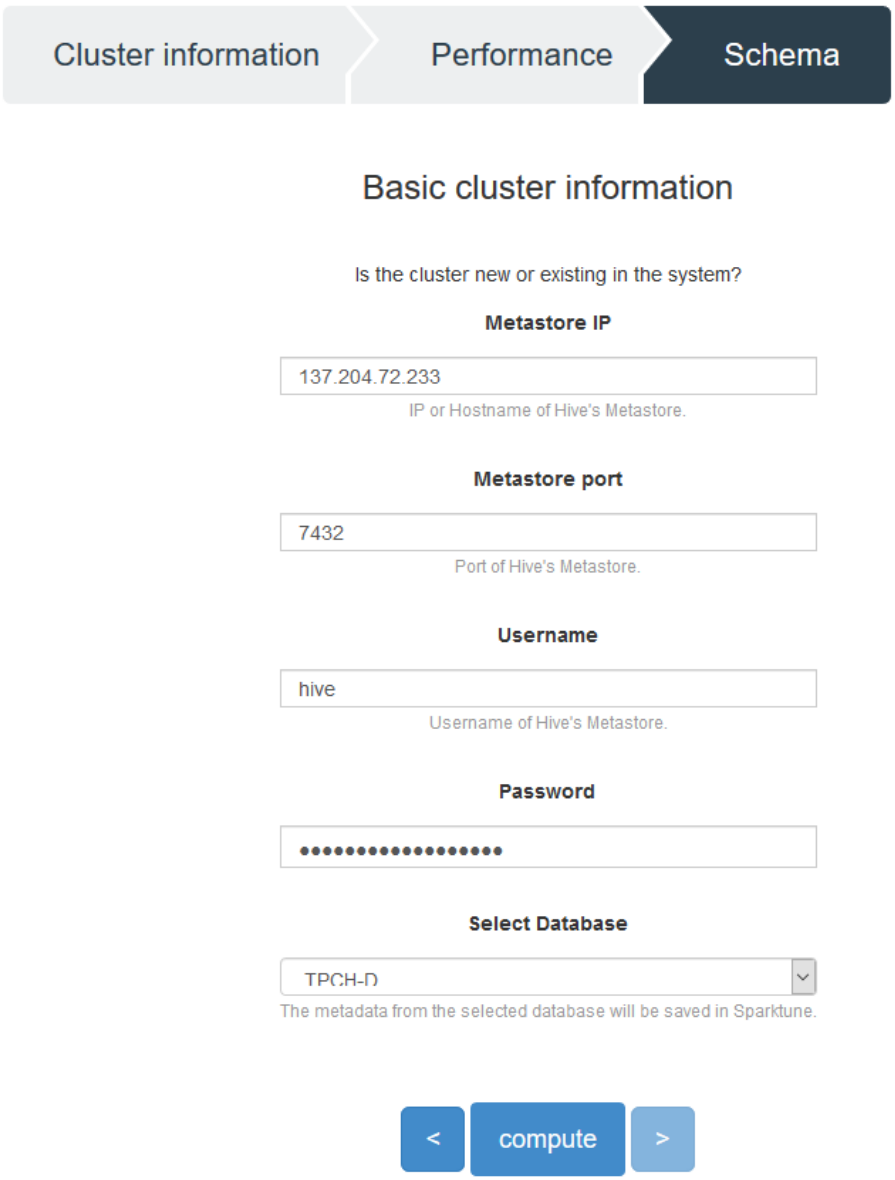

Figura 3.12: Terza schermata dell'interfaccia utente

Tutti i moduli sono identificati mediante una cascata di chiavi esterne, alla cui radice è presente quella del cluster del quale descrivono le informazioni. Le informazioni ottenute mediante il processo di automatizzazione sono direttamente disponibili nelle pagine di calcolo del costo, in quanto vengono salvate direttamente nel database utilizzato da queste. L'integrazione della funzionalità di raccolta automatica consiste semplicemente nel salvare le informazioni sul database del sito.

3.6 Deployment

Il deploy dell'applicazione può avvenire in maniere diverse, e con l'aggiunta del modulo per l'acquisizione automatica dei metadati la presenza in rete del cluster diventa un'informazione influente. In fase di progettazione sono state valutate due opzioni:

- L'applicazione Sparktune è un sito web pubblicamente accessibile su rete internet, un'unica istanza globale è acceduta da chi avesse necessità di utilizzare i servizi offerti;

- L'applicazione Sparktune è distribuita tramite immagine, del sito web vengono installate e conseguentemente hostate copie sulla rete privata di chi sia interessato ad utilizzare il servizio.

La prima presenta per l'utente il vantaggio di non necessitare di un'installazione di Sparktune, ma consente utilizzarlo direttamente in ottica SAAS, traendo i benefici relativi ad eventuali aggiornamenti in tempo reale, e non impiegando risorse per l'installazione, il funzionamento, e la manutenzione del servizio. L'hosting da parte del gruppo di ricerca presenterebbe la possibilità di raccogliere molti dati relativi ai costi teorici, eventualmente integrabili in un secondo momento con i dati effettivi di esecuzioni, per poter fare considerazioni sul miglioramento del modello di costo ed ottenere statistiche sulle caratteristiche dei cluster che usufruiscono del servizio. La stessa opzione presenta però criticità dal punto di vista della sicurezza, in particolar modo per la necessità di eseguire benchmark di rete e disco sui nodi del cluster, operazioni che richiedono di avere accesso diretto alla macchina.

La seconda opzione presenta per l'utente lo svantaggio di dover installare e manutenere il sito, riducendo le problematiche relative alla sicurezza. Si è optato per questa scelta, sia per non dover appesantire il lavoro con questioni di sicurezza, sia per la mancanza delle risorse fisiche per un approccio SAAS. Una rappresentazione del deployment è presentata in figura 3.13, mentre nella figura 3.14 è esemplificata l'opzione alternativa.

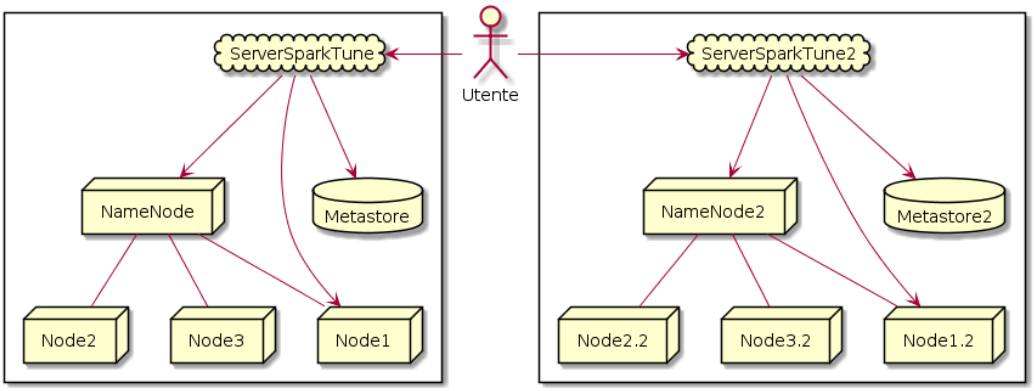

Figura 3.13: Deployment dell'applicazione con installazioni di SparkTune interne ai confini aziendali

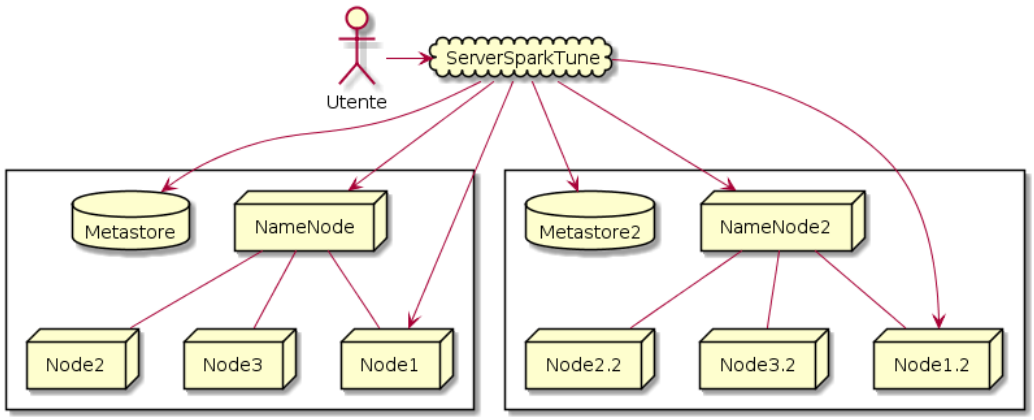

Figura 3.14: Deployment dell'applicazione. Il server SparkTune è al di fuori dei confini aziendali

Conclusioni

In questa tesi si è sviluppato un sistema che integra le varie componenti dell'ambiente di un cluster che utilizza SparkSQL, allo scopo di recuperarne i dati utili al calcolo del costo delle query secondo il modello di costo [7]. Il lavoro è stato suddiviso in tre componenti: topologia, acquisita mediante interrogazioni di tipo REST API, performance, raccolte tramite una serie di test sulle macchine del cluster, ed i metadati, ottenuti tramite query SQL al MetaStore di Hive. È stata successivamente sviluppata un'interfaccia web integrata nel sito SparkTune, per automatizzare la raccolta dei dati a fronte dell'immissione di alcune credenziali in una semplice interfaccia. I dati così catturati sono stati resi disponibili alle altre componenti del sito, il cui compito è calcolare il tempo di esecuzioni di Query secondo il modello di costo.

Il lavoro di tesi ha reso possibile una fruizione più ampia ed immediata di SparkTune, garantendo un processo guidato e semplice da utilizzare. Ciò implica la possibilità di eseguire test del modello di costo su larga scala in tempi brevi, comparando poi i risultati teorici con quelli pratici, per individuare eventuali casi critici non modellati correttamente.

Rispetto a quanto fatto finora, esistono ancora diversi margini di miglioramento del progetto. Un primo aspetto riguarda il calcolo dei throughput, che potrebbe essere migliorato estendendolo ad un insieme più elevato di macchine; in questo modo si potrebbero ottenere dei valori più veritieri (in quanto mediati su un insieme di macchine), a discapito di una maggior pervasività dell'applicazione all'interno del cluster. Altre modifiche potenziali sono collegate direttamente a miglioramenti del modello di costo, come l'utilizzo di istogrammi per calcolare in maniera più precisa la selettività delle query, oppure l'adattamento del modello di costo ad altri motori Big Data come Impala. Un altro aspetto importante è relativo al deploy dell'applicazione, che attualmente necessita di vari passi di installazione. L'utilizzo di tecnlogie a container renderebbe più agevole l'aggiornamento, ed aprirebbero la possibilità di includere Sparktune all'interno di altre distribuzioni più grandi, oltre all'opzione di essere eseguita in ambiente Kubernetes, una piattaforma open source per automatizzare deployment, scalabilità, e manutenzione di applicazioni in container.

Ringraziamenti

Ringrazio i miei genitori, per essersi sempre interessati ai miei studi, per l'entusiasmo che mi hanno trasmesso, per i sacrifici che hanno fatto pur di dirmi *sì*. Ringrazio mio padre, perché in quei dieci minuti a settimana in cui siamo al telefono mi fa sentire lì con lui, e mi sprona a dare sempre il meglio. Ringrazio mia madre perché qualche volta sbaglio, e lei lo capisce. Ringrazio mio fratello per la possibilità di confrontarmi quando ho bisogno. Ringrazio Olga e Francesco, perché anche con loro mi sento a casa. Ringrazio Chiara, perché dà un senso ai mesi che passano, e rende i miei giorni imprevedibili.

Ringrazio anche tutti gli zii e cugini, che sono sempre pronti ad ospitarmi nei posti più disparati del mondo, e che sono vicini ogni volta che possono.

Ringrazio i professori che mi hanno accompagnato in questo percorso: Pietro Di Lena, Paolo Albano, Luciano Margara, Davide Maltoni, Nicoletta Cantarini, Benny Peter Jørgensen, Adam Czerwinski, Bjørk Boye Busch, Vittorio Ghini, Mirko Viroli, Dario Maio, Luca Pasquini, Serena Morigi, Fabrizio Caselli, Alessandro Ricci, Paola Salomoni, Silvia Mirri, Franco Callegati, Claudia Cevenini, Aristide Mingozzi, Raffaele Cappelli, e Matteo Golfarelli, con cui ho svolto il periodo di tirocinio e la tesi di laurea. Un ringraziamento speciale va a Matteo Francia, che mi ha seguito nel percorso di tirocinio, e ad Enrico Gallinucci, co-relatore di questa tesi, che ha sempre trovato il tempo da dedicarmi. Li ringrazio per avermi trasmesso non solo la conoscenza relativa alla materia insegnata, ma una grande passione per ciò di cui loro si occupano, per i progetti che mi hanno dato la possibilità di mettermi alla prova, ed avermi incuriosito ad approfondire gli insegnamenti.

Ringrazio i *casinari* perché oltre ad aiutarmi con un'importantissima dose di svago, sono sempre pronti a discorsi seri, ad un confronto sul futuro, e la nostra posizione nella società, nonostante il nome. Ringrazio i *Pippél*, i miei coinquilini – di via Paiuncolo e via Boscone –, ed i coinquilini di Ruben, per aver riempito la mia permanenza a Cesena di condivisione, risate, ed opportunità.

Ringrazio l'associazione S.P.R.I.Te., perché mi ha fatto vivere l'esperienza più bella della mia vita. S.P.R.I.Te. è ciò di cui ad oggi vado più fiero nella mia vita, la vera opportunità di mettermi in gioco, la possibilità di vivere la vita studentesca appieno, con le iniziative – come dico sempre – dalle più

didattiche a quelle ludiche. Ma S.P.R.I.Te., oltre che ad eventi, burocrazia, bandi, rendicontazioni, e riunioni, è soprattutto la mia seconda famiglia. In spazietto ci ho messo le radici, con voi ci ho passato le giornate, le serate, mi sono confrontato, mi avete aiutato, abbiamo riso, litigato, gioito, pianto assieme.

Ringrazio le segretarie dell'università, perché ogni giorno hanno reso il mio entrare all'università più allegro. Grazie alla Floriana, perché ogni tanto faceva la severa, ma in fondo era per aiutarci. Grazie alla Lella, che è sempre stata dolce e dalla nostra parte. Grazie a Mary ed Elisa, perché anche se siamo da poco nel nuovo campus siamo subito diventati complici.

Grazie ad Enrica, perché con la mia fissa di andare all'estero ha avuto a che fare prima con un Erasmus, poi con il tentativo di andare in Overseas, e di nuovo con l'imminente secondo Erasmus. Grazie per tutto l'aiuto e tutta la pazienza con le domande sulla laurea, sull'Erasmus, sui crediti.

Grazie a te che stai leggendo ora queste righe, perché ci tieni davvero, o semplicemente perché sei saltato alla frase finale.

Bibliografia

[1] Catalyst optimizer. https://databricks.com/glossary/catalyst-optimizer. Accessed: 2018-11-10.

[2] Cluster mode overview. https://spark.apache.org/docs/latest/cluster-overview.html. Accessed: 2018-11-10.

[3] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. An overview of decision support benchmarks: Tpc-ds, tpc-h and ssb. In Alvaro Rocha, Ana Maria Correia, Sandra Costanzo, and Luis Paulo Reis, editors, *New Contributions in Information Systems and Technologies*, pages 619–628, Cham, 2015. Springer International Publishing.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[5] Mike Ferguson. What is hadoop. https://www.ibmbigdatahub.com/blog/what-hadoop. Accessed: 2018-11-10.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[7] M. Golfarelli and L. Baldacci. A cost model for spark sql. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2018.

[8] Github: javiroman. Hadoop ecosystem. https://hadoopecosystemtable.github.io/. Accessed: 2018-11-14.

[9] Jace Klaskowski. Mastering apache spark. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-DAGScheduler-Stage.html. Accessed: 2018-11-10.

[10] Qaware. Big data landscape. https://github.com/qaware/big-data-landscape. Accesed: 2018-11-14.

[11] G. Turkington and G. Modena. *Learning Hadoop 2.* Community experience distilled. Packt Publishing, 2015.

[12] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.