Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione di un Sistema di Analisi delle Performance di Spark SQL

Relatore:
Prof.
Matteo Golfarelli

Presentata da:
Longobardi Luca

Sessione III
Anno Accademico 2016/2017

Introduzione

I Big Data hanno imposto un cambiamento di paradigma nel modo in cui i dati vengono analizzati e processati. Il volume e la grande varietà dei dati hanno spinto inevitabilmente allo sviluppo di nuove soluzioni. In questo contesto Apache Hadoop è il framework che, negli ultimi anni, ha guadagnato grande popolarità sia tra le industrie sia tra i centri di ricerca grazie alle sue capacità di *scaling* e di *fault tolerance* su hardware di commodity. Tra gli strumenti che meglio si integrano ad Hadoop vi è Spark, una piattaforma di cluster computing che estende il popolare paradigma MapReduce per supportare in maniera efficiente diversi tipi di computazioni, come lo stream processing. Spark include anche un sottosistema basato sul linguaggio SQL, che traduce le query SQL standard in termini di comandi Spark, eseguiti poi in parallelo sul cluster. Le performance di Spark SQL permettono di effettuare computazioni on-line su Big Data, e molti vendors OLAP, come Tableau e Micro Strategy, forniscono una connessione con il proprio sistema proprietario.

Nonostante il sistema Hadoop e Spark siano già largamente adottati, essi risultano ancora grezzi e mancano di strumenti adatti a supporto di analisi di dati complesse. In particolare, il modulo Spark SQL non può essere considerato maturo come i tradizionali RDBMs, considerando che il suo componente di ottimizzazione, Catalyst, è tutt'ora rule-based. In questo contesto è stato sviluppato un modello di costo [1] per Spark SQL, che copre l'intera classe delle query GPSJ, e che permette di calcolarne il tempo di esecuzione basandosi sulle statistiche relative alla base di dati e sulla configurazione di

un cluster. Basandoci dunque sull'implementazione del modello sviluppata in [2], abbiamo costruito un'applicazione per il tuning delle analisi di dati e della configurazione di un cluster.

L'obiettivo della tesi è dunque lo sviluppo di un tool che metta a disposizione un insieme di funzionalità, facilmente accessibili ma al contempo estremamente potenti, che permettano all'utente di visualizzare e ottimizzare le performance di un determinato cluster e delle interrogazioni a cui è sottoposto. Le funzionalità sono costituite dall'analisi del workload, nel quale si esamina nel dettaglio il costo e i fattori che modellano un determinato insieme di interrogazioni, dall'analisi delle performance, nella quale si analizzano le performance di un determinato workload al variare delle risorse assegnate su un cluster, dall'analisi del costo, dove l'obiettivo è calcolare il costo di un ipotetico cluster cloud mantenendo come riferimenti la configurazione del cluster fisico e un determinato workload, e infine dall'analisi del cluster, dove l'obiettivo è fornire all'utente una rappresentazione delle performance di un workload al variare delle caratteristiche del cluster.

L'elaborato è strutturato in 5 capitoli. Il primo illustra nel dettaglio l'ecosistema Hadoop e tutti i suoi componenti rilevanti per il progetto, tra cui Spark, Spark SQL e il suo ottimizzatore, Hive e il formato di memorizzazione Parquet. Il secondo capitolo illustra il modello di costo [1], nel quale vengono trattate tematiche fondamentali come la grammatica adottata, le funzioni per il calcolo dei costi ed i parametri principali che li influenzano. Il terzo capitolo illustra nel dettaglio le varie funzionalità sviluppate nel progetto di tesi, soffermandosi in particolare sugli obiettivi e sulle possibilità offerte da questi potenti strumenti. Il quarto capitolo si sofferma sull'interfaccia utente sviluppata a supporto delle funzionalità del tool. Infine il quinto capitolo mostra alcuni risultati sperimentali, a supporto del modello di costo, ottenuti utilizzando il tool sviluppato.

Indice

Capitolo 1

Il sistema Spark SQL

Spark SQL è un modulo di Spark che permette di processare dati strutturati. Al contrario delle API di base di Spark, le interfacce fornite da Spark SQL forniscono all'engine molte più informazioni riguardo la struttura dei dati e la computazione da svolgere. In questo capitolo sono esaminati nel dettaglio sia Spark che il suo modulo SQL, insieme al contesto in cui sono collocati: HADOOP.

1.1 HADOOP

Apache Hadoop è un framework open-source utilizzato per la memorizzazione e la gestione di grandi dataset; la gestione viene effettuata utilizzando non solo MapReduce ma, dalla versione 2, anche altri paradigmi. Benchè non sia più efficiente delle classiche architetture centralizzate, Hadoop rende il cluster computing più semplice da amministrare e programmare, permettendo di scalare orizzontalmente a prezzi modici su hardware di commodity. Hadoop è costituito dai seguenti moduli:

- Hadoop Common: contiene librerie e utility necessarie agli altri moduli;
- Hadoop Distributed File System (HDFS): è un file system distribuito che memorizza i dati su macchine di commodity, fornendo un'elevata bandwidth aggregata;

- Hadoop YARN: è la piattaforma responsabile di amministrare le risorse computazionali nel cluster e usarle per schedulare le applicazioni utente;

- Hadoop MapReduce: è un'implementazione del paradigma MapReduce per il processing di dati su larga scala.

1.1.1 HDFS

L' Hadoop Distributed File System è un file system distribuito pensato per lavorare con hardware di commodity. Presenta diverse similarità con altri file system, ma anche significative differenze:

- hardware failure: il fallimento dell'hardware è la norma piuttosto che l'eccezione. Un'istanza di HDFS può consistere in un elevatissimo numero di macchine, ognuna delle quali memorizza una parte del file system. Pertanto il detect di fallimenti e un recupero rapido e automatico fanno parte delle funzionalità core del modulo;

- streaming data access: HDFS è progettato per il batch processing piuttosto che per l'interazione con l'utente. L'enfasi è pertanto su un alto throughput di accesso ai dati, fornendo così un accesso streaming ad essi;

- large data set: le applicazioni lanciate su HDFS in generale fanno uso di data set molto grandi, pertanto HDFS è costruito appositamente per supportare grandi file e per scalare orizzontalmente, fornendo un' elevata banda aggregata;

- simple coherency model: le applicazioni HDFS necessitano di un modello di accesso ai file del tipo write-once-read-many: un file una volta creato, scritto e chiuso non ha necessità di essere modificato. Questa assunzione semplifica molto i problemi di coerenza dei dati, e permette di erogare un elevato throughput di accesso ai dati;

- moving computation rather than moving data: una computazione richiesta da un'applicazione è molto più efficiente se eseguita vicino ai dati su cui deve operare: ciò minimizza la congestione della rete e aumenta sensibilmente il throughput del sistema;

- portability: HDFS è costruito per essere portabile da una piattaforma all'altra.

HDFS ha un'architettura master/slave. Un cluster HDFS consiste in un singolo NameNode, che è un master server che amministra il file system e regola l'accesso ai file dai client; inoltre sono presenti diversi DataNode, tipicamente uno per ogni nodo nel cluster, che amministrano lo storage.

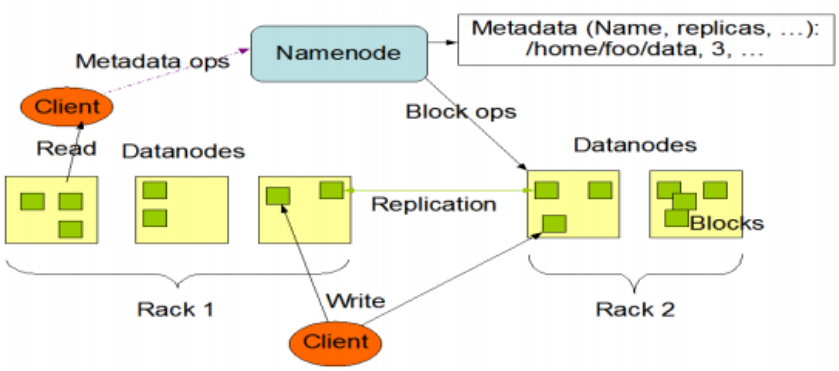


Figura 1.1: Architettura HDFS

HDFS espone un file system namespace e permette ai dati utente di essere memorizzati come file. Internamente, il file è diviso in uno o più blocchi distribuiti e memorizzati in una serie di DataNode. Il NameNode esegue operazioni come l'apertura, chiusura e rinomina di file e directory, e determina la mappa dei blocchi nei DataNode. I DataNode sono responsabili sia di servire richieste di tipo read/write provenienti dal client, sia di eseguere la creazione, cancellazione e replicazione di blocchi da parte del NameNode.

1.1.2 YARN

È la piattaforma responsabile per la gestione delle risorse tra tutte le applicazioni nel cluster e del loro scheduling per le applicazioni utente. Il Job scheduling e il monitoraggio delle funzionalità sono distribuite tra:

- resource manager: ha l'arbitrio delle risorse tra tutte le applicazioni del sistema; le risorse sono assegnate sulla base della nozione di 'resource container'. È a sua volta composto da:
  - scheduler: alloca le risorse alle diverse applicazioni soggette a vincoli di capacità e code. È uno scheduler puro, non esegue altre funzioni;
  - applications Manager: negozia con il container l'esecuzione degli application manager specifici per la singola applicazione; è inoltre in grado di restituire il container AM in seguito al fallimento.
- node manager: per-node slave responsabile del monitoraggio dell'uso delle risorse, riportandone l'uso al resource manager;
- application manager: negozia le risorse del Resorurce Manager e lavora con il Node Manager per eseguire il monitoraggio dei task.

1.1.3 Map Reduce

MapReduce è un modello di programmazione utilizzato per processare e generare big data sets con algoritmi paralleli e distribuiti su un cluster. Un programma MapReduce è composto da una funzione di Map, che effettua operazioni di filtering e sorting, e da una funzione di Reduce, che effettua una summary operation (simile al GroupBy relazionale). L'infrastruttura orchestra il calcolo lanciando e distribuendo diversi task in parallelo, amministrando tutte le comunicazioni e i trasferimenti di dati tra le varie parti del sistema, fornendo anche proprietà di fault tolerance e ridondanza.

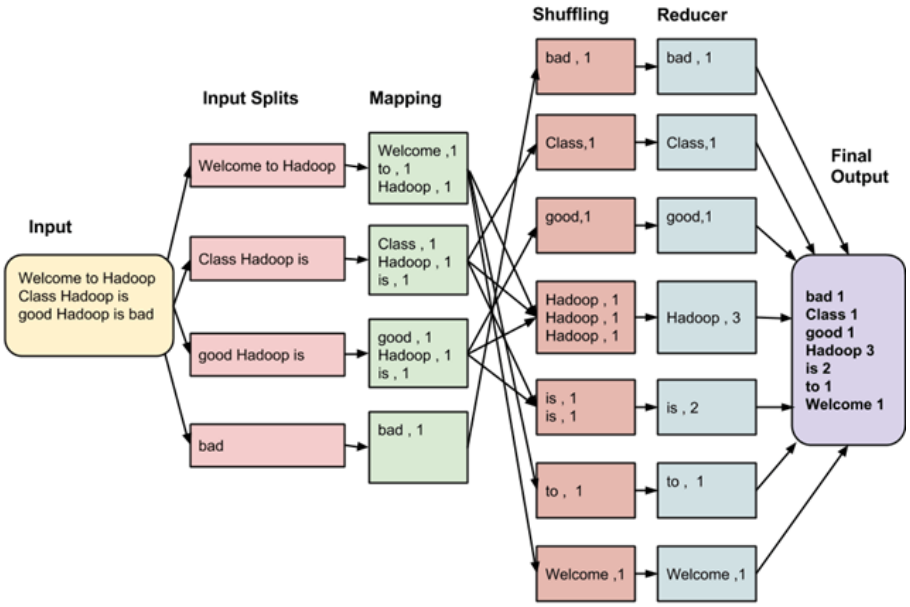


Figura 1.2: Esecuzione di una istanza MapReduce

L'implementazione di MapReduce in ambiente HADOOP è definita come JobMapReduce. Gli elementi che compongono un job sono generalmente:

- input reader: divide l'input in chunk di una appropriata dimensione, per poi assegnarli a una finzione di map. Tipicamente l'input reader legge i dati da un file system distribuito e genera delle coppie chiave/valore;
- funzione di map: prende in ingresso una serie di coppie chiave/valore, le processa, e restituisce in output zero o più nuove coppie. Tipicamente i tipi di dato dall'input all'output vengono cambiati (trasformati);
- funzione di partizione: ogni output delle funzioni di map è allocato ad un particolare reducer dalla funzione di partizione per ragioni di sharding. La funzione di partizione, a partire dalla chiave dalla chiave e dal numero di reducer, ritorna l'indice del reducer desiderato. Tra le operazioni di map e reduce viene effettuata un'operazione di shuffle

(sorting parallelo e scambio di dati tra i nodi) in modo da spostare i dati dal nodo che ha prodotto i risultati alla shard nella quale verranno ridotti;

- funzione di comparazione: l'input di ogni riduzione viene fornito dal nodo che ha lanciato il Map, e poi ordinato usando la funzione di comparazione;
- funzione di reduce: il framework chiama la funzione di riduzione una volta per ogni chiave unica fornita in output dall'ordinamento. La riduzione puó iterare attraverso i valori associati ad una chiave e puó produrre zero o più outputs;
- output writer: scrive l'output della riduzione sullo storage (tipicamente HDFS).

1.2 Spark

Apache Spark è una piattaforma di cluster computing costruita per essere veloce e general-purpose. Dal lato della velocità, Spark estende il popolare paradigma MapReduce per supportare in maniera efficiente diversi tipi di computazioni, includendo query interattive e stream processing. Una delle funzionalità principali che Spark offre è la possibilità di lanciare computazioni in-memory, ma il sistema è anche più efficiente di MapReduce per applicazioni complesse che operano su disco. Dal lato della generalità, Spark è costruito per coprire una grande quantità di workload che precedentemente avrebbero richiesto sistemi distribuiti separati, includendo operazioni batch, algoritmi interattivi, query interattive e streaming. Supportando questi workload nello stesso engine, Spark rende semplice combinare diversi tipi di data processing. Il progetto Spark contiene diversi componenti integrati. Il suo core è un 'computational engine' responsabile dello scheduling, distribuzione e monitoraggio di applicazioni che consiste in diversi task distrubuiti su molti worker machines. Dato che il core engine di spark è sia veloce che

general purpose, esso permette di utilizzare diversi componenti di alto livello specializzati per workload:

- Spark SQL: pacchetto che permette di lavorare con dati strutturati, consentendo di interrogare i dati sia utilizzando l' SQL standard che una versione estesa basata su Hive (HQL);
- Spark Streaming: modulo che permette di processare stream di dati in tempo reale. Esempi di stream di dati includono file di log di web server di produzione, o code di messaggi contenenti aggiornamenti di stato postati da utenti di un web service;
- Mlib: modulo contenente le più comuni funzionalità di Machine Learning. Include algoritmi di classificazione, regressione, clustering e collaborative filtering;
- GraphX: modulo che permette di manipolare grafi e performare computazioni graph-parallel, include anche una libreria dei più comuni algoritmi (PageRank e triangle counting);

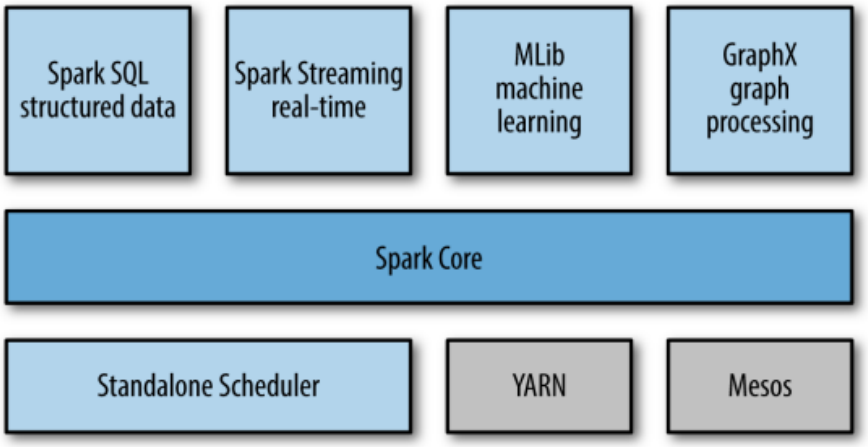


Figura 1.3: Moduli principali di Spark

1.2.1 Resilient Distributed Datasets

I Resilient Distributed Dataset (RDD) sono la principale astrazione di Spark per manipolare i dati. In Spark tutto il lavoro è espresso in termini di creazione, trasformazione e computazione di risultati sugli RDDs. Un RDD è una collezione immutabile e distribuita di oggetti: è diviso in diverse partizioni, che possono essere computate su più nodi del cluster. Una volta creato, un RDD offre due tipi di operazioni: trasformazioni e azioni. Una trasformazione costruisce un nuovo RDD dal precedente; ad esempio, una trasformazione comune consiste nel filtrare i dati che corrispondono ad un certo predicato di selezione. Viceversa, le azioni computano un risultato basandosi su un RDD, e possono sia ritornare un risultato al driver, sia salvarlo un uno storage system esterno (ad esempio HDFS). In modalità distribuita Spark fa uso di un'architettura master/slave, con un coordinatore centrale (Spark Driver) e diversi worker (Executor) distribuiti sul cluster. Un'applicazione Spark è lanciata su un set di macchine utilizzando un cluster manager (YARN nel nostro caso).

1.2.2 Spark Driver

Lo Spark Driver è il processo che crea SparkContext in un'applicazione Spark per l'esecuzione di uno o più job sul cluster. Esso è essenzialmente il coordinatore dell'applicazione, e si occupa di principalmente di:

- convertire un programma utente in task: lo spark driver è responsabile della conversione dei progammi utente in unità di esecuzione fisica chiamati tasks. Un programma Spark crea implicitamente in directed acyclic graph (DAG) delle operazioni da eseguire. Durante l'esecuzione il driver converte il grafo logico in un piano di esecuzione fisico, le cui operazioni creano e trasformano un RDD a partire da un determinato input. Queste operazioni vengono eseguite sequenzialmente e, al loro completamento, l'applicazione termina;

- schedulare i task sugli executor: dato un piano di esecuzione fisico, uno Spark driver deve coordinare lo scheduling dei task individuali sugli executor. Al momento della creazione ogni executor deve registrarsi all'application driver, fornendogli una visione di insieme dell'architettura dell'applicazione. Il driver sfrutta queste informazioni per applicare il principio del *locality aware scheduling*, assegnando i task direttamente ai nodi che contengono direttamente le informazioni da processare.

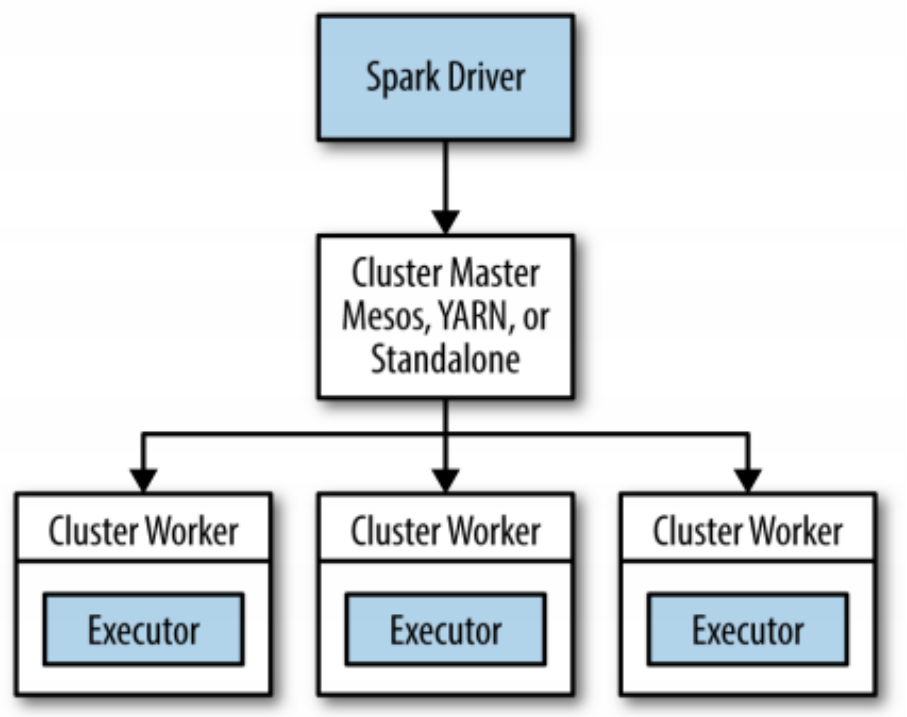


Figura 1.4: Architettura di Spark

1.2.3 Executor

Gli executor di Spark sono processi worker responsabili di eseguire i task individuali in un determinato Job Spark. Questi sono lanciati una volta all'avvio di un'applicazione Spark e tipicamente durano fino al termine dell'applicazione. In ogni caso Spark è in grado di allocare dinamicamente gli executor

durante il ciclo di vita dell'applicazione, garantendo un buon livello di fault tolerance. Gli executor possiedono due ruoli:

- eseguire i task assegnati dall'application driver e aggiornarlo sullo stato dell'esecuzione. Durante il suo ciclo di vita, un executor puó eseguire più tasks, sia in parallelo che in sequenza. Al termine di ogni task, l'executor ha il compito di inviare il risultato al driver;

- fornire un in-memory storage per gli RDDs memorizzati in cache dai programmi utente, attraverso un servizio chiamato Block Manager che vive in ogni executor. Dato che gli RDDs sono memorizzati nella cache degli executor, i task possono essere eseguiti direttamente sui dati in cache.

Inoltre la memoria assegnata ad ogni executor puó essere utilizzata per:

- creare un *Aggregation Buffer*, necessario per eseguire operazioni di *shuffle* oppure per memorizzare i risultati intermedi di un'operazione di aggregazione;

- fornire al codice utente lo spazio necessario per eseguire le proprie istruzioni (allocando risorse come array, set...).

1.2.4 Architettura ad alto livello

Al più alto livello di astrazione una computazione Spark è organizzata in *job*. Un job viene creato quando un'azione viene richiesta su un RDD; esso è composto da unità più semplici di esecuzione chiamate *stage*, connesse tra di loro da un DAG. Una stage è comunque una unità di lavoro logica, composta da una pipeline di trasformazioni applicate ad un RDD di input. L'unità fisica di lavoro, usata per eseguire una stage su ogni partizione RDD, si chiama *task*. I task sono distribuiti sul cluster e eseguiti in parallelo.

1.3 Spark SQL

Spark SQL è il nuovo modulo di Apache Spark che integra la possibilità di esprimere computazioni in linguaggio relazionale con le API funzionali di Spark. In rapporto ai recenti moduli sviluppati nell'ecosistema Hadoop, Spark SQL aggiunge due importanti contributi. In primo luogo offre un legame molto più stretto tra il mondo relazionale e la computazione procedurale, attraverso le API dichiarative dei DataFrame che si integrano perfettamente con il codice procedurale di Spark. Infine Spark SQL include un ottimizzatore modulare ed estensibile, Catalyst, costruito utilizzando le feature del linguaggio Scala, che permette di aggiungere facilmente regole, controllare la generazione del codice e definire le estensioni. In questa sezione sono esaminate nel dettaglio tutte le componenti principali di Spark SQL, con l'aggiunta di Hive poichè parte importante del progetto di tesi.

1.3.1 Datasets e DataFrames

Un dataset è una collezione immutabile e "strongly-typed" di oggetti mappati in uno schema relazionale. Al centro delle API del dataset troviamo il concetto di "encoder", che è responsabile della conversione tra oggetti JVM a rappresentazione tabulare. La rappresentazione tabulare puó essere memorizzata in seguito utilizzando il formato binario interno di Spark: Tungsten. Questo tipo di rappresentazione porta con se diversi vantaggi:

- i dataset hanno accesso a tutto il potere espressivo di un engine di esecuzione relazionale;
- i dataset, grazie alle informazioni sulla struttura dei dati, possono creare un layout più efficiente e ridurre drasticamente l'uso della memoria, fino ad un fattore 4.5;
- l'utilizzo di encoders ottimizzati per effettuare una generazione di codice runtime per costruire un bytecode custom permette di aumentare le

performance durante la serializzazione e la deserializzazione di oggetti, fino ad un fattore 20 rispetto a Java e Kyro.

Un dataframe è essenzialmente un dataset organizzato in colonne nominate, rendendolo concettualmente equivalente ad una tabella di un database relazionale. Questo tipo di rappresentazione rende il costrutto interoperabile con diverse sorgenti di dati, tra cui:

- file dati strutturati;
- tabelle Hive;
- database esterni;
- RDD esistenti.

1.3.2 Catalyst

Catalyst è il modulo di ottimizzazione delle interrogazioni di Spark SQL basato sul linguaggio funzionale Scala. Al contrario dei tradizionali ottimizzatori per RDBMS, Catalyst nasce come componente estremamente modulare, permettendo una semplice estensione da parte di sviluppatori terzi. Il core di Catalyst è costituito da una libreria generale per rappresentare alberi ed applicare regole di trasformazione su questi. Sulla base di questo framework si trovano delle librerie costruite specificatamente per processare query relazionali, oltre a diversi set di regole per gestire le diverse fasi di una interrogazione: analisi, ottimizzazione logica, physical planning e generazione di codice. Infine, Catalyst offre diversi punti di estensione, che includono l'aggiunta di sorgenti di dati esterni e tipi di dato user-defined.

Analisi

La fase di analisi prende in ingresso un piano logico non risolto, restituito dal parser SQL o da un DataFrame costruito attraverso le API. In entrambi i casi la relazione puó contenere referenze non risolte ad attributo o relazioni

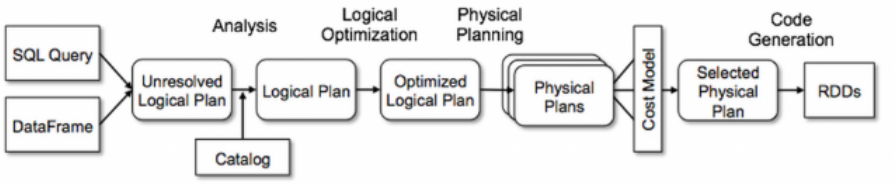


Figura 1.5: Workflow di trasformazione degli alberi di Catalyst

(a questo punto della computazione ancora non si sa se una colonna o attributo dichiarati nell'interrogazione esistono effettivamente in una qualche relazione). Spark SQL usa le regole di Catalyst e del Catalog per risolvere questi attributi, e in seguito applica un set di regole che esegue le seguenti operazioni:

- effettuare il look-up delle relazioni per nome dal Catalog;
- mappare gli attributi risolti nell'input dato dall'operatore figlio;
- determinare quali attributi si riferiscono allo stesso valore, per poi assegnargli un ID unico;
- propagare i tipi di dato attraverso le espressioni (non si conosce il tipo di dato di una espressione fino a che tutte le sotto-espressioni non sono risolte e convertite e tipi di dato compatibili).

Ottimizzazione Logica

La fase di ottimizzazione logica applica un insieme di regole standard al piano logico. Le regole, similari a quelle di un ottimizzatore relazionale, includono il folding delle costanti, il pushdown dei predicati, il pruning delle proiezioni e la semplificazione di espressioni Booleane.

Physical Planning

Durante la fase di physical planning, Spark SQL prende in ingresso un piano logico e genera uno o più piani fisici, utilizzando operatori fisici in accordo

all'engine di esecuzione di Spark. In seguito seleziona uno tra i piani utilizzando un semplice modello di costo, che al momento ha il solo compito di selezionare gli algoritmi di join. Il planner fisico esegue anche ottimizzazioni rule-based, come effettuare il pipelining delle proiezioni o dei filtri in una sola operazione di map di Spark, oppure effettuare il push delle operazioni dal piano logico alle sorgenti dei dati che supportano il pushdown delle proiezioni o dei predicati.

Generazione di Codice

La fase finale dell'ottimizzazione di una query è costituita dalla generazione del bytecode Java da lanciare su ogni macchina. Catalyst fa uso di una feature del linguaggio Scala, i "quasiquotes", per rendere la generazione di codice più semplice. I quasiquotes permettono la costruzione programmatica di AST nel linguaggio Scala, che possono essere poi forniti in input al compilatore per generare il bytecode. Catalyst trasforma un albero, che rappresenta una espressione in SQL, in un AST Scala per la valutazione e la compilazione.

1.3.3 Join in Spark

In generale un'operazione di join ha il compito di combinare le colonne di due tabelle per produrre un risultato basandosi su un predicato che permette di identificare valori comuni sull'attributo di join. In Spark, a causa della distribuzione degli RDD in più partizioni memorizzate in diversi nodi, l'operazione di join risulta più delicata e complessa. In particolare, a differenza delle classiche architetture centralizzate, Spark necessita di eseguire un'operazione aggiuntiva: lo *shuffle* dei dati. Lo shuffling non è altro che il trasferimento dei dati da un nodo all'altro attraverso il cluster, e pertanto dipende sia dalla dimensione dei dati che dal throughput di rete. Spark puó eseguire i join attraverso due algoritmi:

- **Broadcast Hash Join**: questo algoritmo puó essere utilizzato solo quando una delle due tabelle ha dimensioni sufficientemente ridotte da

stare in memoria centrale. In questo caso la tabella più piccola viene inviata in broadcast ad ogni executor in modo che ogni task possa effettuare il join tra la propria partizione della tabella più grande e i dati ricevuti in broadcast.

- **Shuffle Hash Join**: questo algoritmo prevede che entrambe le tabelle vengano ordinate/hashate sugli attributi di join e poi "splittate" nello stesso numero di chunk. I chunk vengono salvati sul disco locale degli executor. Durante l'operazione di shuffling, tutti i chunk con lo stesso range di attributi appartenenti a tabelle differenti vengono inviati ad un task di riduzione, che verifica il predicato di join e salva i dati filtrati su disco.

1.3.4 Hive

Apache Hive è un data warehouse system facente parte dell'ecosistema Hadoop. Hive fornisce una interfaccia SQL-like per interrogare e analizzare dati memorizzati in vari database e file system integrati con Hadoop. Tradizionalmente le query SQL dovevano essere implementate utilizzando le API Java di MapReduce, per poi eseguire la computazione sui dati distribuiti. Hive fornisce le astrazioni SQL necessarie per integrare query SQL-like (HiveQL) nell'ambiente Java sottostante, senza il bisogno di utilizzare API di basso livello.

Features

Tra le feature principali di Hive menzioniamo:

- possibilità di indicizzare i dati attraverso le tradizionali tecniche di indicizzazione (ad esempio la bitmap);

- compatibilità con diversi tipi di storage, come il plain text, HBase, ORC, RCFile e altri;

- memorizzazione di metadati, che permettono di ridurre in maniera significativa il tempo di esecuzione dei controlli semantici durante l'esecuzione di una query;
- possibilità di operare su dati compressi memorizzati nell'ecosistema Hadoop usando algoritmi come DEFLATE, BWT, snappy e altri;
- query SQL-like nel linguaggio HiveQL, che possono essere implicitamente convertite in job MapReduce e, dalle release più recenti, in job Spark.

Architettura

L'architettura di Hive si articola nei seguenti componenti:

- metastore: memorizza i metadati di tutte le tabelle, come lo schema e la location. Include anche la partizione di metadata, che permette al driver di tenere traccia del progresso dei diversi data set distribuiti nel cluster;
- driver: agisce come un controller che riceve le query HiveQL. Inizia l'esecuzione creando la sessione e monitorando il ciclo di vita e il progresso.;
- compiler: effettua la compilazione della query HiveQL, che converte l'interrogazione in un piano di esecuzione. Il compiler converte la query in un AST per eseguire controlli a tempo di compilazione, per poi generare un directed acyclic graph (DAG). Il DAG divide gli operatori in stage MapReduce e tasks basati sulla query di input e sui dati;
- optimizer: effettua varie trasformazioni al piano di esecuzione per ottenere un DAG ottimizzato;
- executor: dopo la compilazione e l'ottimizzazione, l'executor esegue il task assegnatogli. Interagisce con il job tracker di Hadoop per far si

che tutte le dipendenze derivanti da eventuali pipelining siano verificate prima effettuare la computazione;

- CLI,UI e Thrift: la command-line interface (CLI) fornisce una interfaccia utente per usi e interazioni dall'esterno, come la sottomissione di interrogazioni, istruzioni o semplicemente per questioni di monitoraggio. Il server Thrift permette l'interazione con client esterni attraverso la rete, in maniera simile ai protocolli JDBC o ODBC.

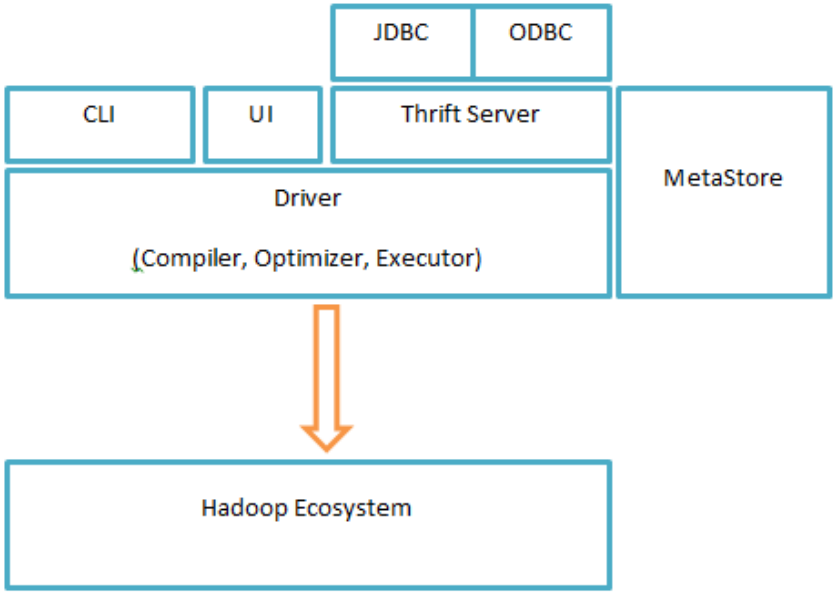


Figura 1.6: Architettura di Hive

Il formato Parquet

Apache Parquet è un formato di memorizzazione di dati column-oriented sviluppato per l'ecosistema Hadoop. È molto simile agli altri metodi di memorizzazione colonnari come RCFile e Optimized RCFile, ed è compatibile con molti dei framework di data processing nell'ambiente Hadoop, tra cui Hive. È implementato utilizzando il concetto di record-shredding e l'algoritmo

assembly, che accomoda le strutture dati complesse che possono essere usate per memorizzare i dati. I valori in ogni colonna sono fisicamente memorizzati in aree di memoria contigue, fornendo i seguenti benefici:

- la compressione per colonna è efficiente e permette il risparmio di una significativa quantità di spazio;
- algoritmi di compressione specifici possono essere applicati quando i tipi di dato per colonna tendono ad essere gli stessi;
- interrogazioni che effettuano ricerche su colonne specifiche non necessitano di leggere le intere righe;
- diversi tipi di encoding possono essere applicati a diverse colonne.

Oltre a ció, Apache Parquet supporta la possibilità di effettuare il push-down delle selezioni alla sorgente dei dati, migliorando drasticamente le performance in lettura.

Capitolo 2

Il modello di costo per Spark SQL

Nonostante i sistemi Hadoop e Spark siano largamente utilizzati, sono ancora in fase di sviluppo e in continua evoluzione. Il sistema Spark SQL non puó essere considerato al livello dei tradizionali RDBMs, e diversi miglioramenti sono possibili. In particolare Catalyst, l'engine che trasforma una query Spark SQL in comandi Spark, utilizza ancora un ottimizzatore rule-based, mentre il modulo cost-based si occupa solamente della scelta dell'algoritmo di Join. In questo capitolo è analizzato nel dettaglio il modello di costo per Spark SQL, proposto nel paper [1], che copre la classe di query GSPJ (Generalized Projection / Selection / Join), costituite da una proiezione effettuata su una selezione del risultato di un set di join. Il tempo di esecuzione è ottenuto sommando il tempo necessario all'esecuzione dei nodi dell'albero che codifica il piano fisico prodotto da Catalyst.

2.1 Grammatica GPSJ

Ogni piano fisico di Spark generato da Catalyst che modella una query GSPJ puó essere rappresentato come un albero, i cui nodi applicano una o più operazioni ad una tabella di input, che sia fisica (ovvero recuperata direttamente dalla base dati) oppure risultante da una serie di operazioni. Tutti gli alberi

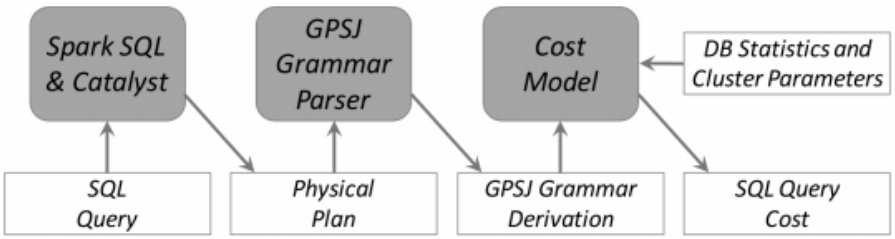


Figura 2.1: Workflow del modello di costo. In grigio i blocchi che rappresentano i moduli del sistema, mentre in bianco i dati in input/output.

```
<GPSJ>::=<Expr> | <GB(<Expr>)>
<Expr>  ::=<SJ(<Expr>,<Expr1>,F)> | <Expr1> |
           <BJ(<Expr3>,<Expr2>,F)>
<Expr1> ::=<SC(<Table>,F)>
<Expr2> ::=<SB(<Table>)>
<Expr3> ::=<SC(<Table>,T)> | <SJ(<Expr>,<Expr1>,T)>
           | <BJ(<Expr3>,<Expr2>,T)>
<Table> ::={pipe-separeted set of database tables}
```

Figura 2.2: Rappresentazione Backus-Naur della grammatica GPSJ

possibili sono rappresentati dalla Context Free Grammar (CFG) in figura 2.2. Ogni derivazione della grammatica è composta da cinque *task type*:

- table scan SC()
- table scan and broadcast SB()
- shuffle join SJ()
- broadcast join BJ()
- group by GB()

SC() e SB() sono sempre i nodi foglia dell'albero di esecuzione, dato che si occupano di leggere le tabelle dallo store fisico. SJ() e BJ() costituiscono

sempre i nodi interni dell'albero, e possono essere composti per creare alberi di esecuzione left-deep. Infine il GB(), se presente, è sempre l'ultimo nodo. Per ragioni di chiarezza nella figura 2.2 sono stati omessi i parametri che caratterizzano i task type, poichè non impattano sulla struttura dell'albero. Questi parametri sono rappresentati nella figura 2.3:

- *pred*: parametro opzionale che rappresenta un predicato di filtro o di join;
- *cols*: rappresenta il subset di colonne da restituire in output;
- *group*: identifica un group by set di una proiezione. SC(), SJ() e BJ() effettuano il grouping, poichè Catalyst puó effettuarne il pushdown per ragioni di efficienza;
- *pipe*: parametro booleano per le operazioni di SC(), SJ() e BJ(). Quando *pipe* è settato a false, il task type non effettua la scrittura dei dati su disco dato che il task type genitore puó essere eseguito in pipeline. Il pipelining puó essere sfruttato solo prima di un broadcast join, dato che tutti gli altri task type o corrispondono a nodi foglia dell'albero, o necessitano di uno shuffle.

Task Type	Additional params	Basic bricks
SC()	pred, cols, groups	Read, Write
SJ()	pred, cols, groups	Shuffle Read, Write
SB()	pred, cols	Read, Broadcast
BJ()	pred, cols, groups	Write
GB()	pred, cols, groups	Shuffle Read, Write

Figura 2.3: Caratteristiche dei Task Types

2.2 Astrazioni del Cluster e Parametri del Modello

Per calcolare correttamente il costo di una esecuzione di una query GSPJ è necessario definire un'astrazione del cluster e l'insieme dei parametri che il modello necessita in input. Con riferimento alla figura 2.4 un cluster è composto da #N nodi equamente distribuiti in #R rack. Ogni nodo ha #C core. Si assume inoltre che tutti i rack e tutti i nodi abbiano le stesse specifiche in termini di Hardware. I dati sono memorizzati nel file system HDFS in blocchi con ridondanza *rf*. Il throughput gioca un ruolo fondamentale nella computazione del tempo di esecuzione, ed è modellato attraverso due funzioni: $\delta_r(\#\text{Proc})$ e $\delta_w(\#\text{Proc})$ che ritornano il throughput per-processo in MB/s. I valori delle funzioni sono ottenuti attraverso un test delle performance che deve essere svolto in ogni cluster. In particolare sono stati considerati il tempo necessario per il caricamento dei dati dal disco e il tempo necessario a renderlo disponibile in un RDD per il processing; perció il throughput del disco considera implicitamente il tempo CPU necessario per svolgere la serializzazione e la decompressione. Naturalmente il throughput in Lettura/Scrittura diminuisce all'aumentare del numero di processi che contendono le risorse.

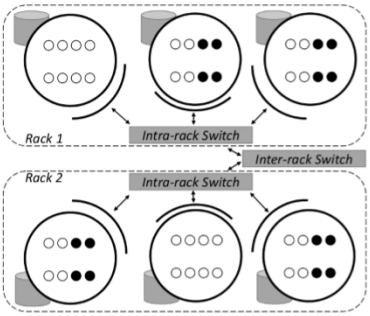


Figura 2.4: Astrazione di un cluster formato da due racks. I cerchi rappresentano gli executors, i punti neri rappresentano i cores che si occupano della computazione.

Parameter	Description
$\#R$	Number of racks composing the cluster.
$\#RN$	Number of nodes in each rack.
$\#N$	Overall number of nodes (i.e., $\#R \cdot \#RN$).
$\#C$	Number of cores available on each node.
rf	Redundancy factor for HDFS.
$\delta_r(\#Proc)$	Disk read throughput (in MB/sec) as a function of the number of concurrent processes.
$\delta_w(\#Proc)$	Disk WRITE throughput (in MB/sec) as a function of the number of concurrent processes.
$\rho_i(\#Proc)$	Network throughput (in MB/sec) between nodes in the same rack as a function of the number of concurrent processes.
$\rho_e(\#Proc)$	Network throughput (in MB/sec) between nodes in different racks as a function of the number of concurrent processes.
$\#SB$	Number of buckets used for shuffling.
$nCmp$	Percentage of data reduction due to compression when transmitting on the network.
$fCmp$	Average size reduction achieved by a compressed file format.
$hSel$	Constant selectivity for HAVING clauses.
$\#RE$	Number of executors allocated to the Spark application in each rack.
$\#E$	Overall number of executors allocated to the Spark application (i.e., $\#RE \cdot \#R$).
$\#EC$	Number of cores for each executor allocated to Spark application.
$t.Attr$	Set of attributes in table t.
$t.Size$	Size (in MB) of table t stored in a uncompressed file format.
$t.PSize$	Average size (in MB) of RDD partitions for table t.
$t.Card$	Number of tuples in table t.
$t.Part$	Number of partitions table t is composed of.
$a.Card$	Number of distinct values for attribute a.
$a.Len$	Average length (in byte) of attribute a .

Figura 2.5: Parametri e funzioni di base del modello di costo. Le linee orizzontali dividono i parametri relativi rispettivamente al cluster, all'applicazione e ai dati.

I nodi del cluster sono connessi tra di loro attraverso la rete; il modello adottato è point-to-point con un limite di banda per ogni connessione. Analogamente al comportamento del disco, il throughput della rete dipende dal numero di processi che concorrono alla trasmissione tra una coppia di nodi. Si assume inoltre che la velocità intra-rack (*IntraRSpeed*) sia maggiore o uguale alla velocità inter-rack (*ExtraRSpeed*). Le formule per il throughput per-processo sono:

- $\pi_i(\#Proc) = \frac{IntraRSpeed}{\#Proc}$
- $\pi_e(\#Proc) = \frac{ExtraRSpeed}{\#Proc}$

Naturalmente le performance della rete e del disco devono essere calcolate a livello di nodo piuttosto che al livello di core, dato che tengono conto delle risorse condivise tra tutti i core all'interno di un nodo. Ogni applicazione

Spark lanciata su un cluster ha il suo set di risorse e parametri. Nel modello si assume che, una volta assegnate, queste risorse non possano essere modificate durante l'esecuzione. Le due risorse principali assegnate all'applicazione sono:

- il numero di executor (#E), che rappresenta il numero di nodi che effettivamente partecipa alla computazione. Questo parametro infatti non puó superare il numero totale di nodi del cluster.

- il numero di core (#EC) attivi per ogni Executor. Anche questo parametro non puó eccedere il numero effettivo di core presenti su ogni nodo del cluster.

Dato che tipicamente il numero delle partizioni RDD è maggiore del numero di core disponibili per la computazione, il resource manager schedula i task sui core in più *wave*. Una wave identifica l'esecuzione parallela di un set di tasks dello stesso tipo, uno per ogni core dell'esecutor. Ogni task processa una partizione distinta del RDD. Tutte le esecuzioni nella stessa wave sono considerate avere lo stesso comportamento. Pertanto dato che tutti i core di un executor lavorano in parallelo durante una wave, il tempo necessario per la sua esecuzione si puó stimare come il tempo necessario ad eseguire un task su un singolo core.

2.3 Mattoni di base del Modello

Modellare esattamente Azioni e Trasformazioni porterebbe ad una inutile complessità dato che il modello si basa sui costi di accesso al disco e alla rete. Per questa ragione, il focus è su un set di *basic brick* che determinano questi tipi di costi e sono usati per computare il tempo di esecuzione delle query GSPJ. Ogni mattone di base modella l'esecuzione di una operazione su una singola partizione RDD, tenendo comunque in considerazione la condivisione delle risorse data dall'eventuale esecuzione parallela. I basic brick

non dipendono dall'SQL, e necessitano solamente dei parametri di Spark e del cluster.

2.3.1 Read

Dato che Spark applica il principio della "data locality", carica sempre le partizioni RDD dalla posizione più vicina. Il tempo di lettura varia se l'executor legge la partizione dal disco locale, dal disco di un nodo appartenente allo stesso rack oppure da un nodo appartenente ad un rack diverso. *Read(Size,X)* computa il tempo di lettura per una partizione di una determinata *Size* e appartenente alla posizione X dove i dati sono memorizzati. X appartiene all'insieme L,R,C, che stanno rispettivamente per Loca, Rack e Cluster. Se da un lato quando i dati sono letti localmente non bisogna considerare nessuna trasmissione, dall'altro in caso di caricamento dal Rack o dal Cluster, il tempo di trasmissione deve essere tenuto in considerazione. Dato che la lettura da disco e la trasmissione dei dati avvengono in pipeline, il tempo totale dell'operazione di Read si puó calcolare come:

$$Read(Size,X) = MAX(ReadT_x; TransT_x)$$

In una wave locale le partizioni RDD sono lette dal disco locale e nessun dato è trasmesso attraverso la rete, pertanto $\mathrm{TransT}_x = 0$. Durante una wave rack ogni core di ogni executor riceve una partizione RDD da un nodo del suo rack che non ospita un executor. I dischi degli executor non prendono parte alla computazione dato che, se una copia della partizione RDD richiesta venisse memorizzata su di essi, sarebbe letta durante una wave locale in accordo al principio della data locality. È chiaro che se tutti i nodi hanno un executor, allora tutte le wave sono locali. Nelle wave di rack e cluster bisogna tenere conto del tempo di trasferimento. In particolare, considerando un executor eseguente una rack wave, ognuno dei suoi #EC cores riceve una partizione RDD da uno dei rimanenti #RN-#RE nodi che non ospitano un executor. Assumendo una distribuzione uniforme delle partizioni RDD tra i nodi del

rack, il numero di cores che condividono la stessa connessione nodo-nodo puó essere approssimata come $\lceil \frac{\#EC}{\#RN-\#RE} \rceil$, permettendo cosí di stimare il tempo di trasmissione per ogni collegamento. Lo stesso ragionamento puó essere applicato per le cluster waves, dove ogni nodo che non ospita un executor puó potenzialmente ricevere una richiesta da tutti i core degli executor appartenenti ad un rack diverso (ovvero:(#R-1)·#RE·#EC). Queste richieste sono distribuite equamente tra (#R-1)·(#RN-#RE) nodi al di fuori del rack che non ospitano un executor, permettendo cosí di stimare il throughput del disco di ogni nodo che non partecipa alla computazione e con questo il tempo necessario ad accedere al disco. Durante una cluster wave #EC partizioni sono trasferite in parallelo ai core di ognuno degli (#R-1)·(#RN-#RE) executor che non partecipano alla computazione ad un rack diverso del cluster. Pertanto il numero di core che condividono la connessione nodo-nodo puó essere stimato come $\lceil \frac{\#EC}{(\#R-1)\cdot(\#RN-\#RE)} \rceil$, permettendo di calcolare il tempo di trasmissione per ogni connessione.

2.3.2 Write

Una volta lette e processate in memoria centrale, le partizioni RDD vengono scritte sul disco locale. Il tempo di scrittura dipende naturalmente dalla *Size* dei dati, dal fattore di compressione e dal numero dei core dell'executor che esegue la computazione. Si noti che il fattore di compressione è una configurazione opzionale in Spark.

2.3.3 Shuffle Read

Durante l'esecuzione di uno shuffle read, Spark genera #SB task che hanno il compito di processare i #SB bucket precedentemente creati durante la fase di shuffle write. Ogni bucket è distribuito equamente tra gli executor, e pertanto ognuno di essi memorizza una porzione di ogni bucket. *SRead(Size)* modella il tempo necessario per leggere un singolo bucket di *Size* MBs. La lettura di un bucket a la trasmissione dei dati all'executor avvengono in pi-

peline e pertanto, in accordo al modello Volcano, il tempo di caricamento è computato come il massimo tra il tempo necessario per effettuare le due operazioni.

$SRead(Size) = \mathrm{MAX}(ReadT, TransT)$

È importante notare che solamente i nodi del cluster che ospitano un executor sono coinvolti in uno shuffle read. Ogni executor si comporta allo stesso modo e il principio della data locality non si puó applicare dato che i dati non sono replicati; inoltre ogni singolo bucket è distribuito tra tutti gli executor. Ogni executor memorizza $Size/\#\mathrm{E}$ MBs per ogni bucket. Ogni core richiede in parallelo agli altri executor la porzione del bucket, pertanto ogni executor deve rispondere a $\#\mathrm{E}\cdot\#\mathrm{EC}$ richieste. Pertanto il tempo di lettura sarà proporzionale alla dimensione della porzione di bucket e inversamente proporzionale al throughput dato da $\#\mathrm{E}\cdot\#\mathrm{EC}$ richieste. Per quanto riguarda il tempo di trasmissione, il ruolo di ogni executor è simmetrico e ogni connessione nodo-nodo è condivisa tra #EC processi. Per la stessa ragione, il numero di processi che trasmettono e ricevono su ogni connessione è lo stesso. Pertanto, assumendo che la velocità della rete inter-rack sia minore di quella intra-rack, è possibile utilizzare il throughput inter-rack per limitare superiormente il tempo necessario a completare la trasmissione, escludendo il caso in cui tutti gli executor risiedano sullo stesso rack.

2.3.4 Broadcast

Un broadcast su un RDD colleziona sull'application driver tutte le partizioni RDD, ognuna di dimensione *Size*; inoltre distribuisce l'intero RDD agli executor per il processing. Il tempo necessario per completare un broadcast è la somma del tempo necessario per collezionare le partizioni e distribuirle, dato che solo quando l'intero RDD è stato caricato nell'application driver viene inviato agli executor:

$Broadcast(Size) = CollectT + DistributeT$

#E·#EC partizioni sono collezionate in parallelo e ogni connessione nodo-driver è condivisa tra #EC processi. Similarmente al caso dello shuffle read, è possibile limitare superiormente il throughput di rete attraverso la velocità inter-rack, escludendo il caso in cui sia l'executor che il driver risiedano sullo stesso rack. Per quanto riguarda il tempo di distribuzione, lo scopo è calcolare il costo per distribuire l'intera tranche di #E·#EC partizioni. Dato che l'application driver manda tutti i dati collezionati a ogni nodo, solo un processo è attivo su ogni connessione. Si noti che *DistributeT* non è il tempo richiesto per distribuire l'intero RDD, ma è piuttosto il tempo necessario per distribuire i dati collezionati in una wave.

2.4 Modello di Costo

In questa sezione è illustrato il processo che permette di comporre i basick brick mostrati nella sezione precedente per modellare il piano di esecuzione di una query GSPJ. Il tempo di esecuzione somma il tempo necessario per eseguire i nodi dell'albero che codifica il piano di esecuzione. A ogni nodo corrisponde un task type, e per ognuno di essi si calcola non solo il costo, ma anche le proprietà della tabella prodotta in output. Una visita depth-first dell'albero permette di avere tutte le proprietà delle tabelle necessarie per computare il nodo corrente.

2.4.1 SC()

Lo Scan accede ad una tabella t memorizzata in HDFS. La funzione *SC(t,pred,cols,groups,pipe)* ritorna il tempo necessario ad eseguire il task. Le operazioni di base che esegue lo scan sono:

1. recuperare le partizioni RDD che mantengono t in memoria. Il recupero include l'accesso a HDFS per recuperare le partizioni RDD e

inviarle agli executor incaricati di processarle. La trasmissione dei dati attraverso la rete è richiesta solamente per quelle partizioni che non sono memorizzate localmente all'executor. Dato che Spark applica il modello Volcano, il tempo di esecuzione dello scan sarà il massimo tra il tempo di accesso e il tempo di trasferimento;

2. effettuare l'operazione di Filter (opzionale) in accordo ai predicati *pred*. Dato che Catalyst applica il push down della selezione, il filtering è eseguito non appena la tupla non è più necessaria per la computazione. Quando supportato dal formato di memorizzazione (es. Parquet), Spark puó effettuare il push down del filtering alla sorgente dei dati; in questo modo le tuple inutili non vengono nemmeno lette;

3. l'operazione di proiezione (opzionale) sulla tabella permette di eliminare le colonne inutilizzate. Dato che Catalyst applica il push down della proiezione, l'eliminazione è eseguita non appena una colonna non è più necessaria per la computazione. Quando supportato dal formato di memorizzazione (es. Parquet), Spark puó effettuare il push down della proiezione alla sorgente dei dati; in questo modo le colonne inutili non vengono nemmeno lette;

4. l'operazione di aggregazione (opzionale) delle tuple avviene quando Catalyst effettua il push down di una proiezione in modo da ridurre la quantità di dati da gestire durante la computazione;

5. la scrittura (opzionale) su disco delle tuple rimanenti per le prossime computazioni o per salvare il risultato finale. La scrittura viene evitata quando un broadcast join è in pipeline.

2.4.2 SB()

Uno Scan & Broadcast accede ad una tabella t memorizzata in HDFS e la invia all'application driver, che colleziona le partizioni RDD ed effettua il broadcast dell'intera tabella a tutti gli executor. La funzione *SB(t,pred,cols)*

ritorna il tempo necessario ad eseguire il task. I passi di collezione dei dati sono gli stessi dello SC(); tuttavia la dimensione di ogni partizione RDD su cui effettuare il broadcast puó essere ridotta dai predicati di filtro e di proiezione. Dato che le operazioni di filtering e projection vengono effettuate o in memoria centrale o direttamente dal file system, non impattano direttamente sul costo del task; inoltre, considerando che il Broadcast e il Retrieve dei dati avvengono in pipeline, il costo finale sarà dato dal massimo tra le due operazioni.

2.4.3 SJ()

Uno Shuffle Join esegue l'operazione di join tra due tabelle t_1 e t_2 le cui partizioni sono già state hashate in #SB bucket. Le partizioni RDD in input sono memorizzate nel disco locale degli executor. La funzione *SJ(t_1,t_2,pred,cols,group,pipe)* restituisce il tempo necessario ad eseguire il task. Le operazioni eseguite dallo SJ() in una wave sono:

1. Shuffle Read: i bucket corrispondenti da t_1 e t_2 vengono recuperati. Una porzione di ogni bucket è memorizzata in ogni executor;

2. Join: una volta che due bucket corrispondenti da t_1 e t_2 sono disponibili, il predicato *pred* viene utilizzato per eseguire il merge delle tuple;

3. Project (opzionale): le colonne non più utili per la rimanente parte della query vengono eliminate. Solo le colonne appartenenti all'insieme *cols* vengono mantenute;

4. Aggregating (opzionale): l'aggregazione delle tuple avviene quando Catalyst effettua il push down della proiezione in modo da ridurre la quantità di dati gestita dalla computazione seguente;

5. Writing (opzionale): scrittura dei dati su disco delle tuple rimanenti. Questa operazione viene evitata se l'operazione di broadcast join è in pipeline.

2.4.4 BJ()

L'operazione di broadcast join esegue l'operazione di join tra due tabelle t_1 e t_2 quando una delle due, ad esempio t_1, è abbastanza piccola da per essere inviata in broadcast e tenuta completamente in memoria centrale dagli executor. La funzione *BJ(t_1,t_2,pred,cols,groups,pipe)* restituisce il tempo necessario ad eseguire il task. In questo caso l'unica operazione rilevante è la scrittura dei dati sul disco. Questo perchè il costo per il caricamento delle due tabelle è affidato ai nodi figli dell'albero di esecuzione. In accordo alla grammatica GPSJ t_1 è caricata attraverso un'operazione di SB(), mentre t_2 è caricata mediante l'operazione di SC() o è risultante da un task precedente (come ad esempio SJ() o BJ()). In maniera simile allo shuffle join, il filtering, la proiezione e il grouping possono essere opzionalmente eseguiti in memoria centrale. Il broadcas join è sempre in pipeline con un'altra operazione, che sia uno scan, uno shuffle join o un broadcast join, e pertanto il numero di wave dipendono dal numero di partizioni di t_2.

2.4.5 GB()

L'operazione di group by esegue il grouping finale in una query GPSJ. Le tuple della tabella di input t sono state preventivamente hashate in #SB bucket. Le partizioni RDD in input sono memorizzate nel disco locale degli executors. La funzione *GB(t,pred,cols,group)* ritorna il tempo necessario per eseguire il task. Ad ogni wave, le operazioni eseguite dal GB() sono:

1. Shuffle Read dei bucket corrispondenti da t. Una porzione di ogni bucket viene memorizzata su ogni executor;
2. raggruppamenti dei dati in accordo all'attributo *groups* e i valori aggregati per i rimanenti attributi *cols-groups* vengono computati;
3. scrittura su disco delle tuple aggregate.

Il raggruppamento non puó iniziare prima che tutte le tuple dei bucket corrispondenti siano state caricate. Di conseguenza, il tempo necessario per

eseguire una wave è la somma del tempo necessario per effettuare lo shuffle read e la scrittura delle tuple processate.

Capitolo 3

Simulazione di performance

Gli esempi utilizzati per illustrare i risultati ottenuti fanno riferimento al workload utilizzato in [1] per valutare l'accuratezza del modello nel misurare il tempo di esecuzione delle query GPSJ. Il workload è composto dalle query q_1, q_3, q_6, q_{10} del noto benchmark TPC-H di dimensione 100 GB. Mentre le query q_1 e q_6 risultano piuttosto semplici e sono composte da pochi task types, q_3 e q_{10} mostrano tutta l'espressività delle query GPSJ. Infine la tabella 3.1 mostra le caratteristiche del cluster su cui sono stati effettuati i test delle funzionalità del tool.

Installazione	$\#R$	$\#N$	$\#C$	*MainMem.*	*Disco*	*ReleaseSoftware*
OnPremises	1	11	8	$32GB$	$6TB$	$Hadoop2.6.0 + Spark2.2.0$

Tabella 3.1: Caratteristiche del cluster.

3.1 Analisi del Workload

L'analisi del workload ha come obiettivo il calcolo del tempo di esecuzione di un insieme di query mediante il modello di costo e la visualizzazione dei dettagli più rilevanti della computazione. Il tool fornisce una vasta quantità di informazioni, tra cui:

- il tempo totale di esecuzione dell'intero workload;
- l'albero di esecuzione per ogni query del workload;
- il tempo di esecuzione di una query diviso in Lettura, Scrittura e Rete;
- varie altre informazioni, come il numero di wave, la selettività e la cardinalità delle tabelle per ogni nodo dell'albero di esecuzione.

L'analisi del workload richiede una serie di parametri in ingresso per eseguire la simulazione:

- i parametri del cluster;
- le statistiche relative al database su cui eseguire la simulazione;
- il workload soggetto della simulazione;
- il numero di executor e di core per executor da utilizzare per calcolare il tempo di esecuzione.

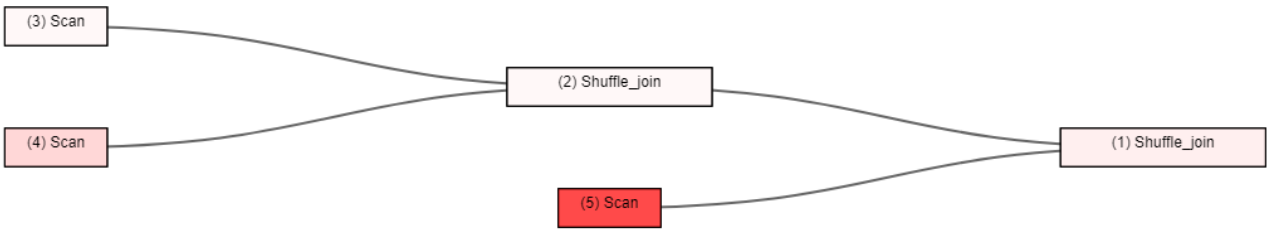
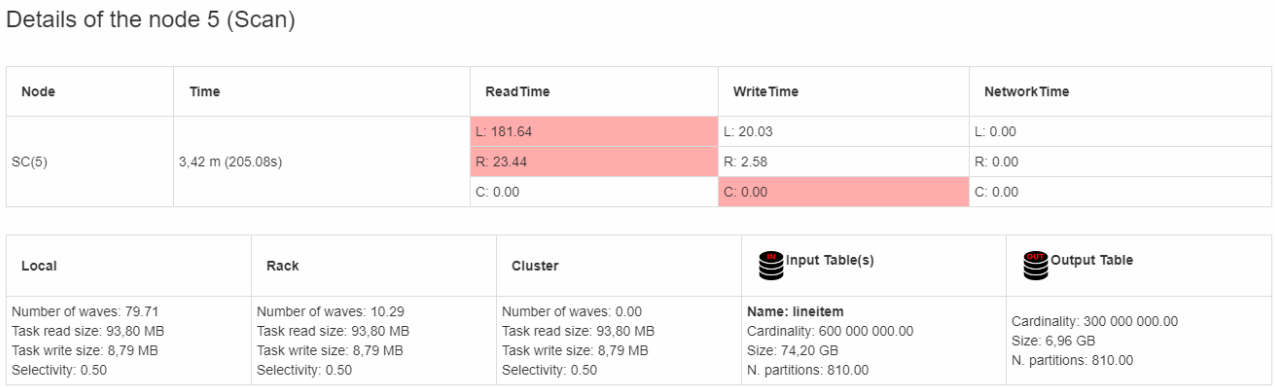


Figura 3.1: Albero di esecuzione della query q_3; la tonalità di rosso dei nodi indica il peso sul costo totale della query.

Le informazioni ricavate dal tool danno la possibilità all'utilizzatore di identificare rapidamente i fattori che incidono maggiormente sul tempo di esecuzione di una query, permettendo di effettuare operazioni di tuning mirate. In riferimento alla figura 3.1, possiamo osservare che una parte consistente del tempo di esecuzione della query q_3 è costituita dal nodo (5)Scan.

Esplorando il nodo in questione (figura 3.2) notiamo che esso è composto da un'operazione di SC(), divisa in tempo di esecuzione Locale, per Rack e per Cluster; in questo caso l'intero costo è dato solamente dal tempo di lettura (in rosso chiaro) effettuato sulla tabella più grande del sistema, **lineitem**, con una cardinalità di $6*10^8$ records.


Details of the node 5 (Scan)

Node	Time	ReadTime	WriteTime	NetworkTime
SC(5)	3,42 m (205.08s)	L: 181.64	L: 20.03	L: 0.00
		R: 23.44	R: 2.58	R: 0.00
		C: 0.00	C: 0.00	C: 0.00

Local	Rack	Cluster	Input Table(s)	Output Table
Number of waves: 79.71 Task read size: 93,80 MB Task write size: 8,79 MB Selectivity: 0.50	Number of waves: 10.29 Task read size: 93,80 MB Task write size: 8,79 MB Selectivity: 0.50	Number of waves: 0.00 Task read size: 93,80 MB Task write size: 8,79 MB Selectivity: 0.50	**Name: lineitem** Cardinality: 600 000 000.00 Size: 74,20 GB N. partitions: 810.00	Cardinality: 300 000 000.00 Size: 6,96 GB N. partitions: 810.00

Figura 3.2: Dettaglio dell'esecuzione dello scan(5) della query q_3.

3.2 Analisi di Costo

L'analisi del costo ha come obiettivo il calcolo del costo in denaro dell'esecuzione di un particolare workload su un cluster cloud, configurato in modo da essere il più simile possibile al cluster selezionato come input nell'applicazione. Inoltre, attraverso la visualizzazione di due grafici, l'utente è in grado di valutare nel complesso il costo al variare delle risorse assegnate(executor e core), e di rapportare il costo in denaro al tempo di esecuzione. Il tool fa uso di un modello di costo che converte il tempo di esecuzione in costo (in dollari) e prende in considerazione due grandi provider cloud, Amazon e Google. Inoltre l'analisi di costo richiede una serie di parametri in ingresso per eseguire la simulazione:

- i parametri del cluster;

- le statistiche relative al database su cui eseguire la simulazione;
- il workload soggetto della simulazione;
- il provider utilizzato per eseguire la simulazione (Amazon o Google).

3.2.1 Il modello

Il modello di costo sviluppato, seppur con alcune semplificazioni, è in grado di trasporre il cluster selezionato dall'utente (tra i cluster caricati sull'applicazione) su un cloud provider, che ne eredita le caratteristiche principali, permettendo di ottenere la previsione del costo di un determinato workload. Il modello ha come punto di partenza le seguenti assunzioni:

- **la dimensione del disco è stabilita a priori per ogni nodo**: in generale nel modello di costo(in tempo di esecuzione) non è considerata la dimensione del disco come parametro in ingresso. Nella realtà di un cluster, in contrasto con un modello di cluster classico, la dimensione del disco incide sul suo throughput; pertanto semplificando, si assume che la dimensione del disco sia quella necessaria a raggiungere da velocità di throughput massima (1.5 TB);
- **la quantità di memoria centrale per nodo è fissa**: nel modello di costo la quantità di memoria centrale a disposizione dell'applicazione non è considerata. Si assume pertanto che la quantità di memoria disponibile su ogni nodo del cluster equivalga a quella del cluster utilizzato nelle prove sperimentali (30 GB). Si noti che in generale, per applicazioni di analisi di dati distribuite su grandi cluster cloud, si tende a non eccedere una tale dimensione per nodo, considerando che sperimentalmente si preferisce aumentare la quantità di nodi distribuendo la computazione orizzontalmente, piuttosto che implementare un'architettura verticale;
- **il throughput di rete viene considerato identico al cluster selezionato dall'utente**: in generale risulta difficile stabilire con esattezza

il throughput della rete di un cluster cloud; i cloud provider rilasciano dati riguardo le velocità massime, ma chiaramente, a causa delle virtualizzazione delle risorse, i risultati sperimentali posso variare sensibilmente. Pertanto semplificando, si assume che il throughput dei nodi in un cluster cloud equivalga alla velocità del cluster di partenza;

- **si assume che un cluster cloud abbia un numero di nodi equivalente alla somma totale dei nodi del cluster selezionato dall'utente**: in generale nell'architettura cloud il concetto di rack non esiste, infatti tutti i nodi istanziati fanno sempre parte di un unico cluster. Si considera pertanto che l'architettura sia equivalente alla semplice somma dei nodi del cluster di partenza (numero di rack moltiplicato il numero di nodi per rack).

- **la tipologia delle macchine cloud scelte per istanziare i nodi è selezionata a priori**: i vari provider cloud mettono a disposizione degli utenti diverse tipologie di macchine per diverse esigenze. Per semplificare il modello, la tipologia delle macchine per i provider Google e Amazon sono selezionate a priori tra le macchine con le caratteristiche più adeguate in un contesto di analisi di dati. In particolare per quanto riguarda Google cloud si è scelto di utilizzare le macchine della generazione n1, personalizzabili in termini di risorse mediante appositi paramentri, mentre per quanto riguarda Amazon cloud si è scelto di utilizzare la categoria di macchine di generazione R4. Entrambe le tipologie sono descritte nel dettaglio nelle sezioni a seguire, insieme ad un esempio pratico per calcolare il costo di un'esecuzione.

Google Cloud

Le macchine standard della serie n1 (figura 3.3) hanno un certo numero di cpu virtuali. Queste sono implementate come un singolo hardware hyper thread su vari tipi di core. Pertanto, per il nostro modello una cpu virtuale corrisponde ad un core.

Machine name	Description	Virtual CPUs[1]	Memory (GB)	Max number of persistent disks (PDs)[2]	Max total PD size (TB)
n1-standard-1	Standard machine type with 1 virtual CPU and 3.75 GB of memory.	1	3.75	16 (32 in Beta)	64
n1-standard-2	Standard machine type with 2 virtual CPUs and 7.5 GB of memory.	2	7.50	16 (64 in Beta)	64
n1-standard-4	Standard machine type with 4 virtual CPUs and 15 GB of memory.	4	15	16 (64 in Beta)	64
n1-standard-8	Standard machine type with 8 virtual CPUs and 30 GB of memory.	8	30	16 (128 in Beta)	64
n1-standard-16	Standard machine type with 16 virtual CPUs and 60 GB of memory.	16	60	16 (128 in Beta)	64
n1-standard-32	Standard machine type with 32 virtual CPUs and 120 GB of memory.	32	120	16 (128 in Beta)	64
n1-standard-64	Standard machine type with 64 virtual CPUs and 240 GB of memory.	64	240	16 (128 in Beta)	64
n1-standard-96 (Beta)	Standard machine type with 96 virtual CPUs and 360 GB of memory.	96	360	16 (128 in Beta)	64

Figura 3.3: Caratteristiche delle macchine Google standard di generazione n1 .

Il parametro memory si riferisce alla quantità di memoria centrale disponibile alle applicazioni sul nodo. Per semplificare il modello si considera una macchina avente 32 GB di memoria centrale. Per quanto riguarda il disco, Google fa uso dell'astrazione di Persistent disk (figura 3.4). La performance

degli standard persistent disk non dipende dal numero di core della macchina, ma dalla quantità di dati memorizzati nell'istanza:

Volume Size (GB)	Monthly Price	Sustained Random IOPS		Sustained Throughput (MB/s)	
		Read	Write	Read	Write
10	$0.40	*	*	*	*
32	$1.28	24	48	3	3
64	$2.56	48	96	7	7
128	$4.00	96	192	15	15
256	$10.24	192	384	30	30
512	$20.48	384	768	61	61
1000	$40.00	750	1500	120	120
1500	$80.00	1500	3000	180	120
2048	$81.92	1536	3072	180	120
4000	$160.00	3000	6000	180	120
8192	$327.68	3000	12288	180	120
10000	$400.00	3000	15000	180	120
16384	$655.36	3000	15000	180	120
32768	$1,310.72	3000	15000	180	120
65536	$2,621.44	3000	15000	180	120

Figura 3.4: Caratteristiche degli standard persistent disks di Google.

Dato che il nostro modello di costo considera query di tipo GPSJ, consideriamo la computazione come di tipo "Streaming Large read/write", tenendo in considerazione il throughput in relazione alla dimensione dei dati. Il massimo si raggiunge a 1.5 TB, mentre la velocità si intende per istanza, non per disco (la quantità di dischi impatta solo sulla memoria fisica). Per quanto riguarda invece il network throughput, esso è misurato tra macchine appartenenti alla

stessa zona (ovvero che fanno parte dello stesso cluster). Anche dividendo la zona in più sottoreti, nello specifico tutto è organizzato come un singolo cluster formato da un solo rack. Nel modello Google, e in generale in tutte le architetture cloud, non esiste un'astrazione di rack. La velocità teorica del network tra le varie istanze è 2Gbits/second moltiplicato il numero di vCPU per istanza. A questo costo bisogna sottrarre la velocità di scrittura dei dati su persistent disk. Questo perchè i dischi hanno un fattore di ridondanza 3.3, inviando i dati attraverso la rete. A parte per le istanze con un solo core, per cui la velocità di scrittura è 78 MB/s, la velocità teorica massima è 120 MB/s. Per concludere il prezzo mensile/orario di una macchina su misura è indicato in figura 3.5.

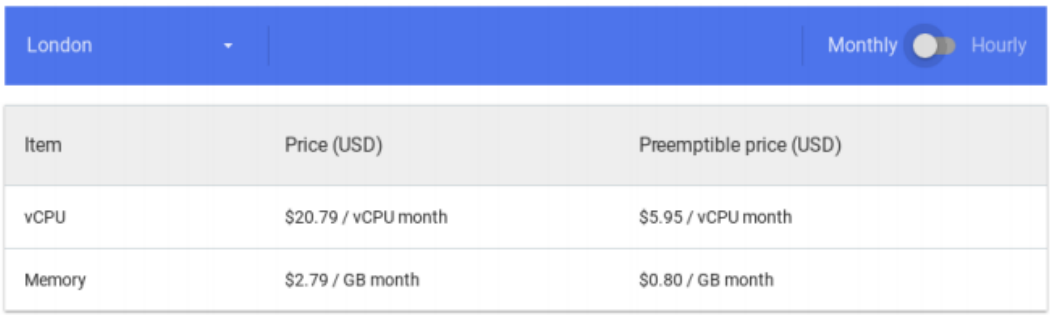


Figura 3.5: Prezzo mensile di una macchina Google, suddiviso per numero di cores e memoria centrale.

E' importante tenere conto che il costo dei presistent disk è a parte, ed è indicato in figura 3.4. Ad esempio, se volessimo costruire un cluster di 11 nodi, di tipologia n1, con 30GB di memoria centrale, 1.5TB di disco, e 8 core, spenderemmo:

```
(Costo RAM/mese * N.GB ram) / (86,400 secondi/giorno * 30 giorni al mese) +
(Costo vCPU/mese * N.vCPU)/ (86,400 secondi/giorno * 30 giorni al mese) +
(Costo Memoria/mese) / (86,400 secondi/giorno * 30 giorni al mese)
```

Il tutto moltiplicato per il numero di nodi e per il numero di secondi impiegati dal cluster ad effettuare la computazione. Nel nostro caso, immaginando di avere un workload di 30 minuti:

```
((2.79 * 30) / (86,400 * 30 ) +
(20,79*8) (86,400 * 30 ) +
(80)/(86,400 * 30 ))*11*30*60 = 128,3$
```

Amazon Cloud

Amazon mette a disposizione vari tipi di istanze ottimizzate per diversi casi d'uso. L'istanza scelta per il calcolo del costo è la R4, mostrata in figura 3.6.

R4

Le istanze R4 sono ottimizzate per le applicazioni che fanno uso intensivo di memoria e offrono inoltre un prezzo migliore per GiB di RAM rispetto alle istanze R3.

Caratteristiche:

- Processori ad alta frequenza Intel Xeon E5-2686 v4 (Broadwell)
- Memoria DDR4
- Supporto per funzioni di rete avanzate

Modello	vCPU	Mem (GiB)	Prestazioni di rete	Storage SSD (GB)
r4.large	2	15,25	Fino a 10 Gigabit	Solo EBS
r4.xlarge	4	30,5	Fino a 10 Gigabit	Solo EBS
r4.2xlarge	8	61	Fino a 10 Gigabit	Solo EBS
r4.4xlarge	16	122	Fino a 10 Gigabit	Solo EBS
r4.8xlarge	32	244	10 Gigabit	Solo EBS
r4.16xlarge	64	488	25 Gigabit	Solo EBS

Figura 3.6: Caratteristiche delle macchine Amazon di tipologia R4.

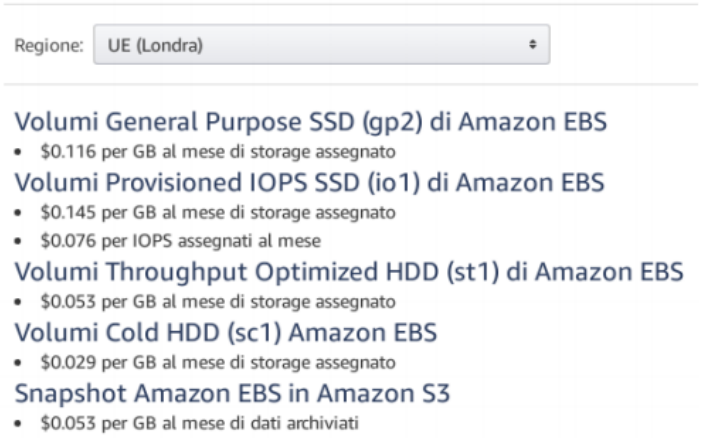
Anche in questo caso le vCpu indicano il numero di core dell'istanza. Amazon permette di creare delle reti di cluster che, utilizzando determinate istanze, permettono di raggiungere elevate prestazioni intra-cluster. Le istanze R4 supportano le reti cluster. Le istanze avviate su un gruppo di collocazione

cluster comune sono posizionante in un cluster logico che offre elevata larghezza di banda e reti a bassa latenza tra tutte le istanze del cluster. La larghezza di banda che può essere usata da un'istanza in un gruppo di collocazione dipende dal tipo di istanza e dalle prestazioni di rete. Se avviate in un gruppo di collocazione, determinate istanze possono usare fino a 10 GB/s per traffico a flusso singolo e 25 GB/s per traffico full duplex. Il traffico di rete all'esterno di un gruppo di collocazione cluster (ad esempio internet) è limitato a 5 GB/s (full duplex). Per quanto riguarda lo storage, è possibile scegliere diversi tipi di dischi ottimizzati per vari casi d'uso. Per la nostra applicazione è possibile scegliere tra gli SSD Provisioned e i Throughput Optimized (figura 3.7). Chiaramente le macchine ottimizzate per il throughput risultano essere più appropriate per la nostra tipologia di computazione.

	Dischi a stato solido (SSD)		Dischi rigidi (HDD)	
Tipo di volume	Provisioned IOPS SSD EBS (io1)	General Purpose SSD (gp2) di EBS*	Throughput Optimized HDD (st1)	Cold HDD (sc1)
Breve descrizione	Volumi SSD con le prestazioni più elevate per carichi di lavoro transazionali sensibili alla latenza	Volumi SSD ad utilizzo generico con buon rapporto prezzo/prestazioni per carichi di lavoro transazionali	Volumi HDD a costo ridotto per carichi di lavoro ad elevati throughput e accessi frequenti	Volume HDD con il prezzo più basso per carichi di lavoro con pochi accessi
Casi d'uso	Database relazionali e NoSQL con requisiti I/O elevati	Volumi di avvio, app interattive a bassa latenza, testing e sviluppo	Big Data, data warehousing, elaborazione di log	Dati inattivi con poche scansioni giornaliere
Nome API	io1	gp2	st1	sc1
Dimensioni volume	4 GB – 16 TB	1 GB – 16 TB	500 GB – 16 TB	500 GB – 16 TB
IOPS**/volume max	20.000	10.000	500***	250***
Throughput/volume max	320 MB/s	160 MB/s	500 MB/s***	250 MB/s***
IOPS/istanza max	75.000	75.000	75.000	75.000
Throughput/istanza max	1.750 MB/s	1.750 MB/s	1.750 MB/s	1.750 MB/s
Prezzo	0,125 USD GB/mese 0,065 USD/Provisioned IOPS	0,10 USD GB/mese	0,045 USD GB/mese	0,025 USD GB/mese
Caratteristica principale	IOPS	IOPS	MB/s	MB/s

Figura 3.7: Tipologie di storage utilizzabili per le macchine Amazon.

Allo stesso modo di google, la figura 3.8 mostra i prezzi di una macchina su misura, divisi per storage e istanze. assegnate



Memoria ottimizzata – Generazione attuale

x1.16xlarge	64	174.5	976	1 x 1920 SSD	$8.403 all'ora
x1.32xlarge	128	349	1952	2 x 1920 SSD	$16.806 all'ora
r4.large	2	7	15.25	Solo EBS	$0.156 all'ora
r4.xlarge	4	13.5	30.5	Solo EBS	$0.312 all'ora
r4.2xlarge	8	27	61	Solo EBS	$0.624 all'ora
r4.4xlarge	16	53	122	Solo EBS	$1.248 all'ora
r4.8xlarge	32	99	244	Solo EBS	$2.496 all'ora
r4.16xlarge	64	195	488	Solo EBS	$4.992 all'ora

Figura 3.8: Dettaglio del costo di una macchina Amazon, diviso per tipologia e storage.

Sulla traccia dell'esempio di Google, calcoliamo il potenziale costo per un cluster Amazon. Se volessimo costruire un cluster di 11 nodi, di tipologia R4, con 30GB di memoria centrale, 1.5TB di disco, e 8 core, spenderemmo:

```
(Costo macchina/ora)/ (60 secondi in un'ora) +
(Costo Memoria Throughput ottimizzata/mese * NGB per istanza) /
(86,400 secondi/giorno * 30 giorni al mese)
```

Il tutto moltiplicato per il numero di nodi e per il numero di secondi impiegati dal cluster ad effettuare la computazione. Nel nostro caso, immaginando di

avere un workload di 30 minuti:

```
((0.624) / (60) + (0.053 * 1500) / (86,400 * 30 ))*11*30*60 = 206$
```

3.2.2 Risultati Sperimentali

L'obiettivo del test è verificare e comparare il comportamento del modello per i due provider cloud presi in considerazione: Google e Amazon. Il test misura il costo in denaro, a partire dal tempo di esecuzione del workload,al variare del numero di executors assegnati e al numero di cores per executors. In seguito i costi vengono messi in relazione ai tempi di esecuzione in un grafico 2D, mostrando per ogni punto le risorse assegnate. Le figure 3.9 e 3.11 mostrano l'andamento del costo (in denaro) di un'esecuzione del workload su cluster rispettivamente Google e Amazon al variare delle risorse assegnate. I grafici forniscono una visione completa sull'efficienza dell'assegnamento delle risorse, permettendo all'utente di scegliere il trade-off migliore tra performance e costo.

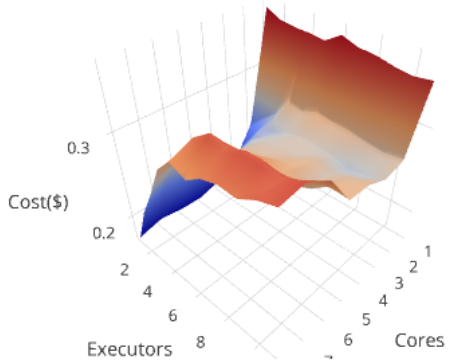


Figura 3.9: Grafico 3D del costo di un cluster Google al variare delle risorse assegnate. Il costo varia da 0,36 a 0,17 dollari per-esecuzione.

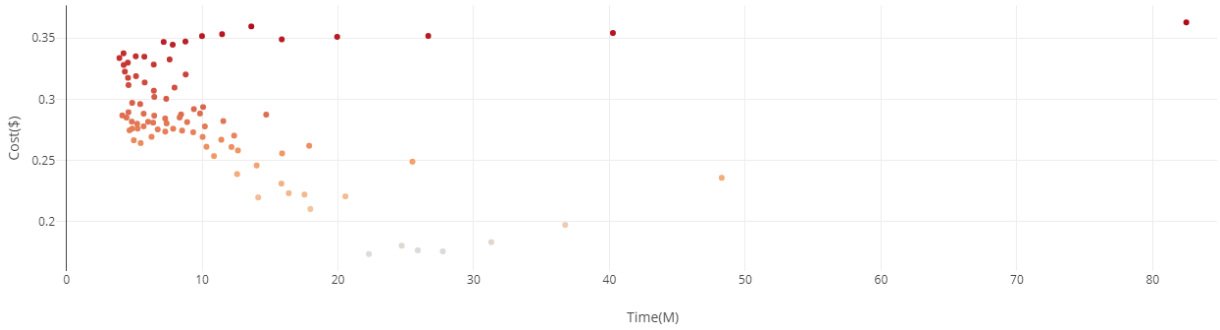


Figura 3.10: Grafico 2D della relazione tra il costo in denaro e il tempo di esecuzione del workload sul cluster Google. L'hover sui punti (non in figura) mostra le risorse assegnate per ogni punto.

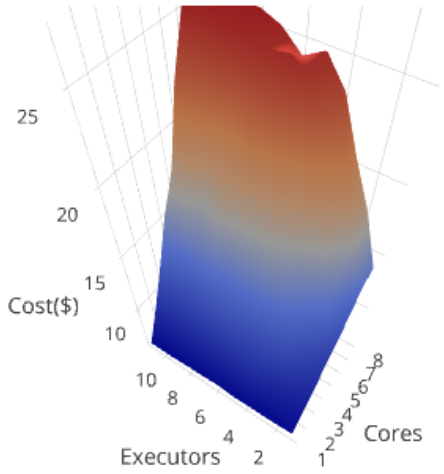


Figura 3.11: Grafico 3D del costo di un cluster Amazon al variare delle risorse assegnate. Il costo varia da 0,36 a 0,17 dollari per-esecuzione.

La figura 3.10 mette in luce il rapporto tra il costo in denaro di un particolare workload e il suo tempo di esecuzione sul cluster Google. Sperimentalmente otteniamo il costo massimo assegnando solamente 1 executor con 1

core, poichè in questo caso il tempo di esecuzione elevato ha un peso considerevole. Il caso diametralmente opposto, nel quale il tempo di esecuzione è minimo (11 executor con 8 core ciascuno), presenta anch'esso un costo elevato, influenzato dall'assegnazione di una quantità considerevole di risorse. Viceversa il costo minore si ottiene assegnando ad un solo executor 5 cores, mentre nell'insieme la maggior parte dei punti a costo minimo ha un numero limitato di executor (fino a 6) e un numero di cores compresi tra 3 e 5; il risultato è apprezzabile anche dalla "valle" mostrata in figura 3.9.

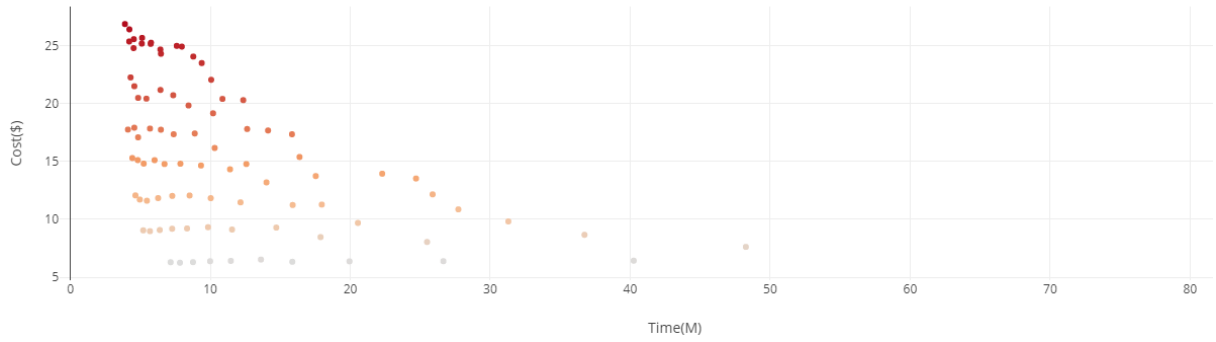


Figura 3.12: Grafico 2D della relazione tra il costo in denaro e il tempo di esecuzione del workload sul cluster Amazon. L'hover sui punti(non in figura) mostra le risorse assegnate per ogni punto.

La figura 3.11 evidenzia il rapporto tra il costo in denaro di un particolare workload e il suo tempo di esecuzione sul cluster Amazon. Sperimentalmente otteniamo il costo massimo assegnando 11 executor con 8 cores ciascuno, mentre i costi minimi si ottengono per tutti quei punti che minimizzano il numero di cores per executor. Questo accade perchè il prezzo dei core per ogni executor ha un peso molto elevato (il costo è superiore di circa un ordine di grandezza a quello di Google). Il risultato è anche apprezzabile in figura 3.11, dove la "valle" in questo caso corrisponde a tutte le istanze che possiedono solamente 1 core per executor.

In generale il tool si rivela estremamente potente, permettendo all'utente non solo di prevedere il costo di un'eventuale workload su un cluster cloud, ma anche di scegliere il corretto trade-off tra efficienza e convenienza valutando le informazioni fornite da un'interfaccia semplice ed intuitiva.

3.3 Analisi del Cluster

L'analisi del cluster ha come obiettivo il calcolo del tempo di esecuzione di un workload al variare del throughput di disco e di rete. In particolare l'interfaccia permette scegliere la configurazione di un cluster utilizzando un workload ed un set di risorse assegnate come metro della valutazione. L'analisi richiede un set di parametri in ingresso per computare la simulazione:

- i parametri del cluster;
- le statistiche relative al database su cui eseguire la simulazione;
- il workload soggetto della simulazione;
- il numero di executor e di core per executor da utilizzare per calcolare il tempo di esecuzione.

La figura 3.13 mostra l'andamento del tempo di esecuzione del workload al variare del throughput di rete e del disco. In particolare i throughput hanno come valore iniziale il 100%, e vengono aumentati con uno step del 20% fino a raggiungere il tetto del 300%. In questo caso possiamo notare, come ci si aspettava, che le prestazioni calano simil-linearmente sia all'aumentare della velocità di rete che della velocità del disco.

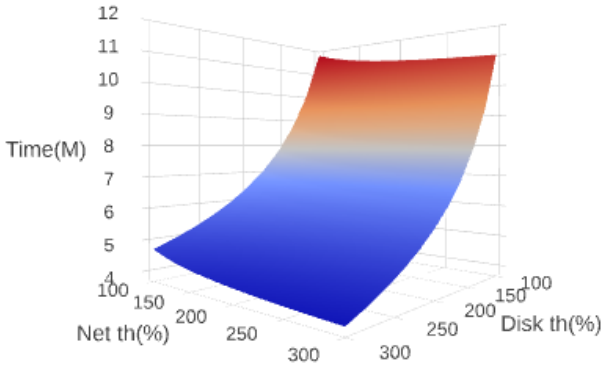


Figura 3.13: Grafico 3D dell'andamento del tempo di esecuzione del workload al variare del throughput. All'esecuzione sono stati assegnati 3 executors con 3 cores ognuno.

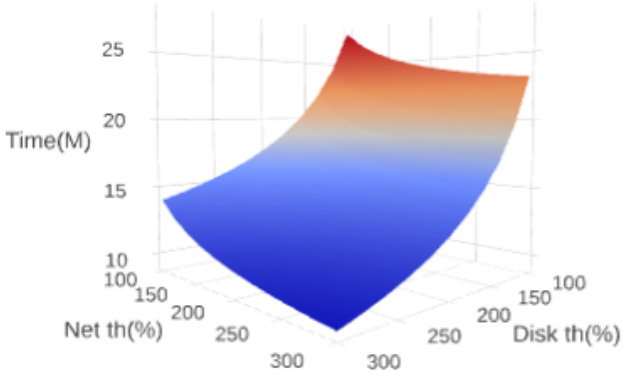


Figura 3.14: Grafico 3D dell'andamento del tempo di esecuzione del workload usato in [2] al variare del throughput. All'esecuzione sono stati assegnati 3 executors con 3 cores ognuno.

La figura 3.14 mostra un grafico simile, realizzato facendo uso del workload visto in [2]. Anche in questo caso possiamo notare come le prestazioni dipendano quasi linearmente dal throughput del disco e della rete. Tuttavia al contrario dell'esempio precedente, questo workload ha una dipendenza dal throughput di rete più caratterizzante, dovuta al fatto che diverse query includono task types che ne influenzano il costo.

In generale il tool si rivela estremamente utile, permettendo all'utente di prevedere l'eventuale impatto che avrebbe un determinato miglioramento delle prestazioni (rete o disco) sul workload a cui il cluster è sottoposto. Infine alcune note finali sul modello di costo e la relativa rappresentazione nel grafico:

- il mattone di base *read* del modello di costo è influenzato, come visto nelle sezioni precedenti, da un certo tempo di trasferimento. In particolare a livello di task type tutte le operazioni caratterizzate da una componente di *read*, come SC(), vengono considerate interamente come costi di lettura; questo implica che solamente le operazioni che contengono un mattone di base *broadcast* influenzeranno il costo di rete (e di conseguenza il throughput);

- la possibilità di portare al livello di task type il costo di rete contenuto nel mattone di base *read* è ancora motivo di dibattito. Nelle future release dell'applicazione non è escluso che questa modifica venga apportata.

3.4 Analisi di Performance

L'analisi di performance ha come obiettivo il calcolo del tempo di esecuzione di un workload al variare del numero di executor e di core per executor. L'analisi richiede un set di parametri in ingresso per computare la simulazione:

- i parametri del cluster;

- le statistiche relative al database su cui eseguire la simulazione;
- il workload soggetto della simulazione;

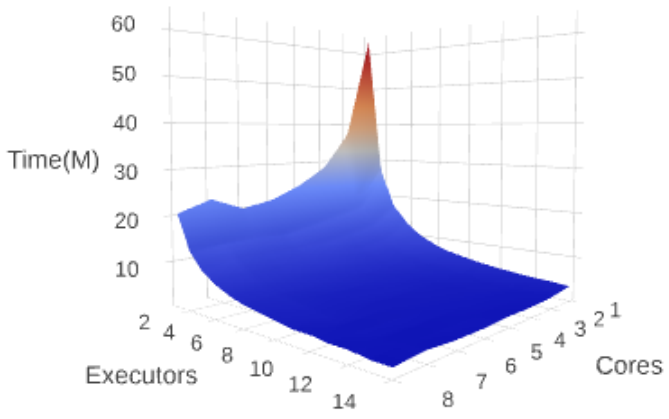


Figura 3.15: Grafico 3D dell'andamento del tempo di esecuzione del workload al variare del numero di executors e di cores per executors.

La figura 3.15 mostra l'andamento del tempo di esecuzione del workload al variare delle risorse. In generale, come si nota anche dall'esempio, il tempo di esecuzione di un workload non cala linearmente all'aumentare delle risorse, ma piuttosto presenta una discesa iniziale esponenziale, per poi appiattirsi rapidamente. Ciò accade perchè le risorse di sistema (disco e rete) vengono inizialmente utilizzate appieno dalla ridotta quantità di processi che accedono concorrentemente, causando così la discesa esponenziale del costo, per poi essere saturate non appena il numero di processi concorrenti supera una certa soglia, provocando l'"appiattimento" visto nell'esempio.

In generale il tool si dimostra estremamente utile non solo per valutare le performance di un determinato workload su un cluster, ma anche per agevolare la scelta del corretto trade-off tra prestazioni e risorse assegnate, attraverso un'interfaccia semplice ed estremamente intuitiva.

Capitolo 4

L'interfaccia Utente

In questo capitolo sono illustrate le varie funzionalità dell'applicazione a livello di interfaccia, a partire dalla sezione di impostazioni e concludendo con la descrizione delle quattro pagine "core" descritte nel capitolo precedente. Ogni pagina è illustrata partendo da una visione di insieme, per poi analizzare i vari componenti di cui è costituita.

4.1 Impostazioni

Le pagine di impostazioni permettono all'utente di configurare diversi parametri necessari all'applicazione per eseguire le varie procedure di analisi. Oltre al management del workload, l'interfaccia da la possibilità di inserire tutti i parametri del cluster e del database, visti nel capitolo 2, necessari al modello di costo per eseguire la computazione.

4.1.1 Amministrazione dei Workload

La pagina di amministrazione dei workload permette all'utente di inserire uno o più workload nell'applicazione per eseguire i vari tipi di analisi. In riferimento alla figura 4.2, il primo riquadro permette di inserire un piano di esecuzione e la sua relativa query SQL, mentre il secondo riquadro mostra i piani di esecuzione correnti, e oltre alle operazioni di base (cancellazione e

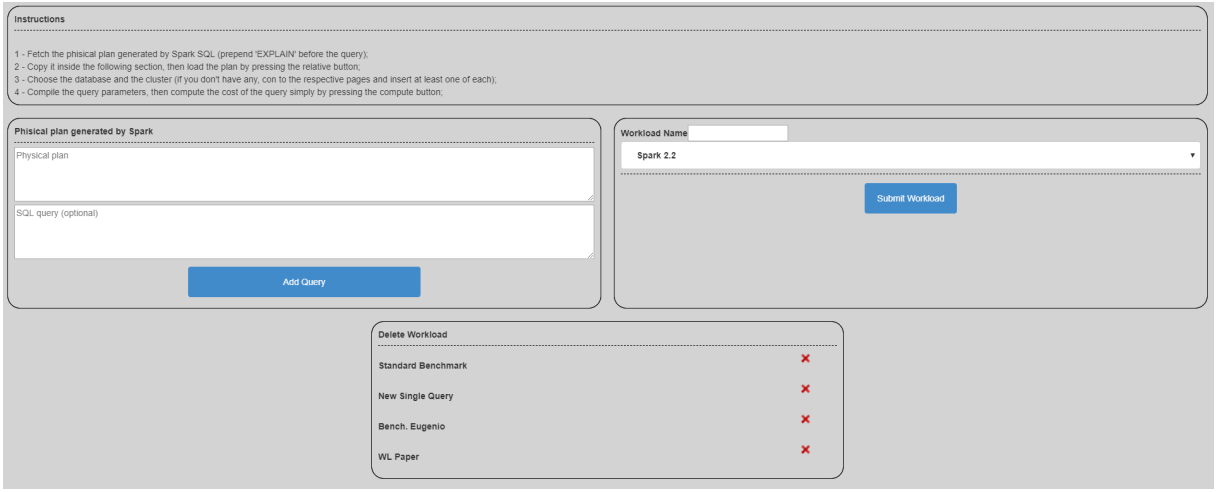


Figura 4.1: Interfaccia completa per l'inserimento di un workload nell'applicazione.

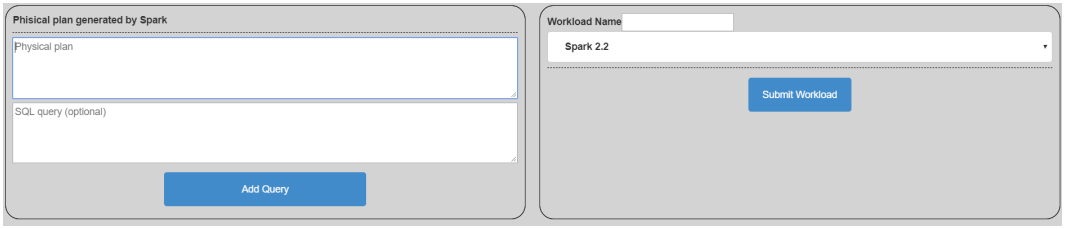


Figura 4.2: Interfaccia per l'inserimento dei piani di esecuzione.

visualizzazione) permette di specificare la versione di Spark che ha generato i piani ed effettuare il submit all'applicazione. Tutti i workload inseriti e salvati tramite la procedura sono legati all'utente che ha effettuato l'operazione, e vengono mantenuti anche tra sessioni di lavoro diverse. L'interfaccia (figura 4.3) fornisce in output anche una lista di tutti i workload salvati, riferiti all'utente corrente, permettendo anche di cancellarli. L'operazione elimina l'intero workload, inclusi tutti i suoi piani di esecuzione.

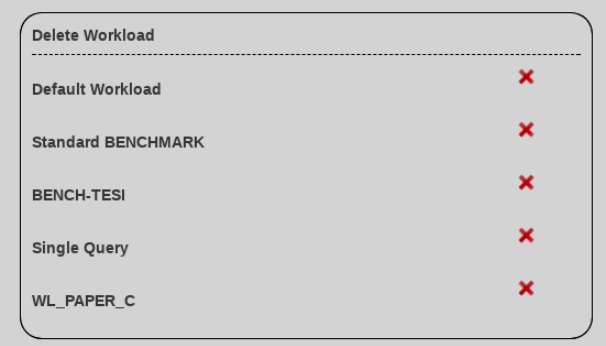


Figura 4.3: Interfaccia per la cancellazione dei workload.

4.1.2 Amministrazione dei Cluster

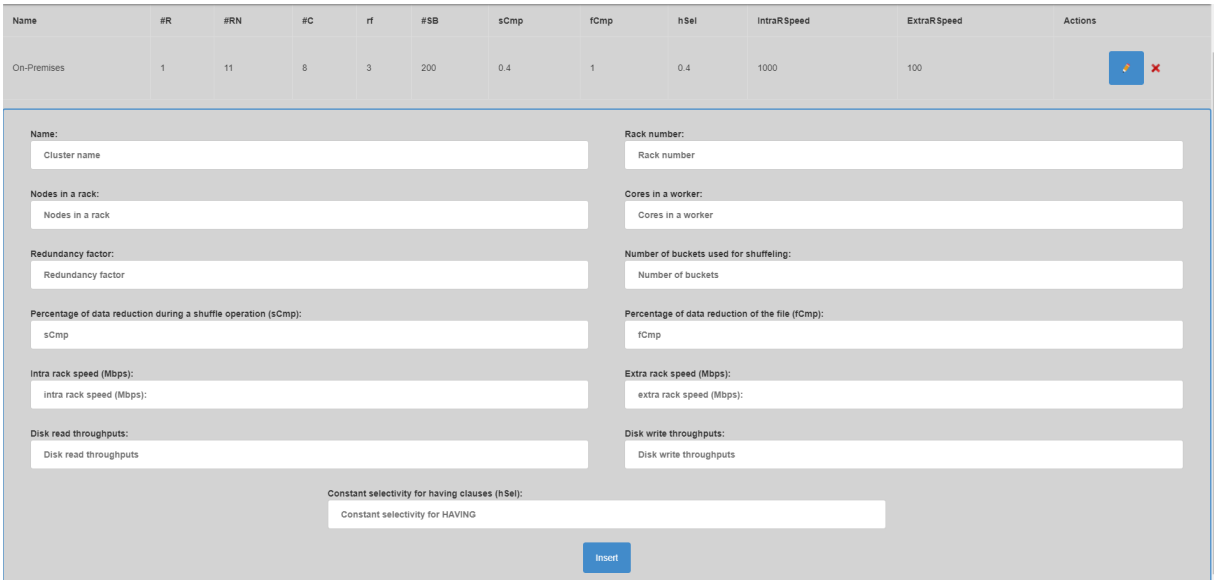


Figura 4.4: Interfaccia per l'inserimento di un cluster nel sistema.

La pagina di amministrazione dei cluster permette all'utente di creare, modificare o eliminare un cluster. Ogni cluster mostra un insieme di parametri (figura 4.5) necessari al modello di costo per eseguire la computazione, inseribili attraverso un insieme di campi appositi, mostrati in figura 4.4. Come

per i workload, anche i cluster sono legati all'utente, e vengono mantenuti tra le varie sessioni.

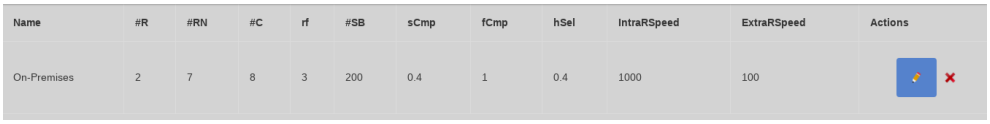


Figura 4.5: Interfaccia per la visualizzazione dei Cluster memorizzati nel sistema.

4.1.3 Amministrazione dei Database

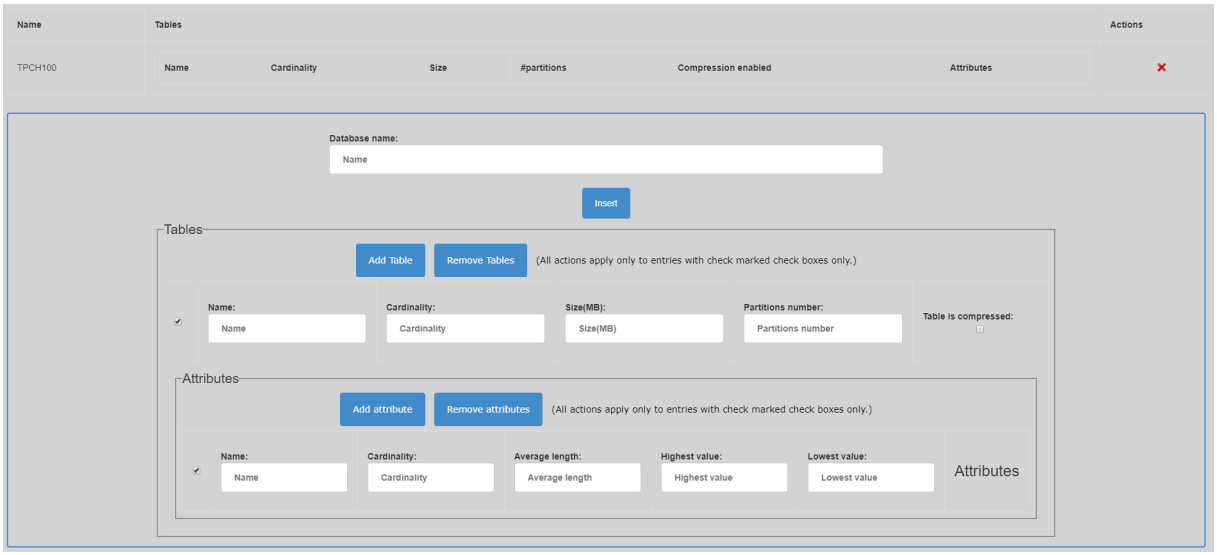


Figura 4.6: Interfaccia per l'inserimento di un database nell'applicazione.

La pagina di amministrazione dei database permette all'utente di creare, modificare o eliminare un database. Ogni database contiene l'insieme delle feature e delle statistiche relative ad uno specifico database fisico. Le figure 4.7 e 4.8 mostrano la porzione di interfaccia che visualizza, in questo caso, le informazioni relative alle tabelle del database TPC-H, come il nome, la

cardinalità, le dimensioni, il numero di partizioni in cui sono memorizzate e l'insieme di attributi di cui sono costituite.

Figura 4.7: Interfaccia per la visualizzazione dei Database memorizzati nel sistema.

Figura 4.8: Lista delle tabelle, con le relative informazioni.

La figura 4.9 mostra invece la porzione di interfaccia che fornisce le informazioni relative agli attributi delle tabelle di un database, come la cardinalità, la lunghezza media e i valori minimi e massimi. Allo stesso modo dei workload e dei cluster, anche i database sono legati all'utente e vengono conservati tra le varie sessioni. Alla versione attuale dell'applicazione tutte le informazioni devono essere inserite manualmente compilando un apposito form per ogni database, ma nelle future release questa operazione sarà effettuata automaticamente.

Name	Cardinality	Avg length	Highest value	Lowest value
l_comment	142,510,152	11	zzle? Unusual	Tiresias
l_commitdate	2,466	11	1998-10-31	1992-01-31
l_discount	11	11	0.1	0.0
l_extendedprice	3,786,026	11	104948.5	900.05

Figura 4.9: Lista degli attributi delle tabelle, con le relative informazioni.

4.2 Analisi del Workload

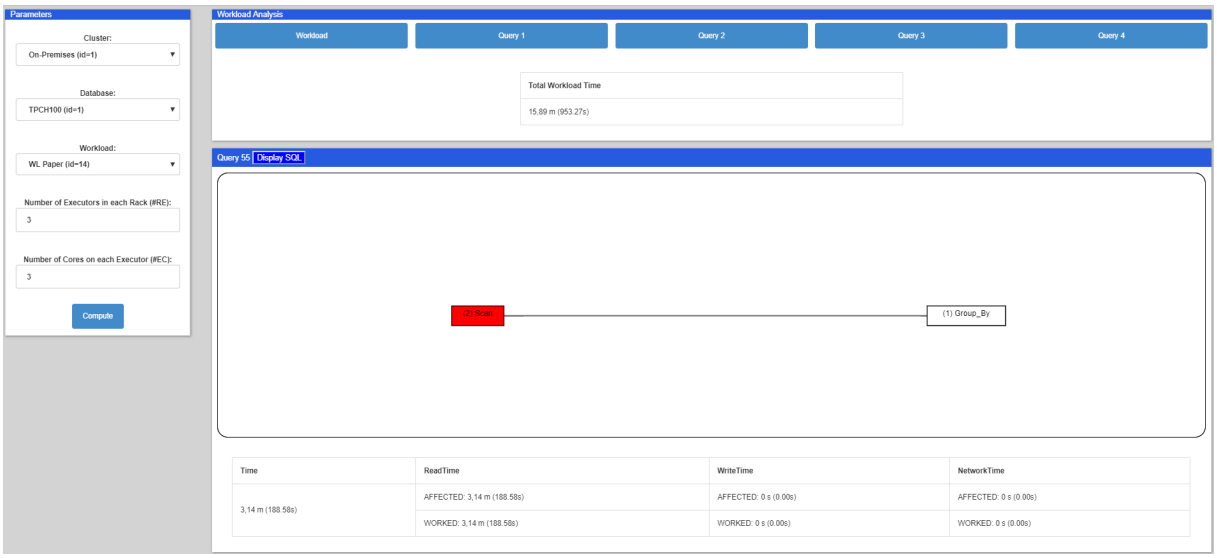


Figura 4.10: Interfaccia completa dell'analisi del workload.

La pagina di analisi del workload permette all'utente di effettuare un'analisi approfondita delle query che compongono un workload. Per effettuare questa computazione si inseriscono le informazioni necessarie, illustrate nel capitolo 3, tramite il menù laterale (figura 4.10). In output (figura 4.11) si ha il tempo totale di esecuzione e un elenco selezionabile delle query di cui il workload è composto.

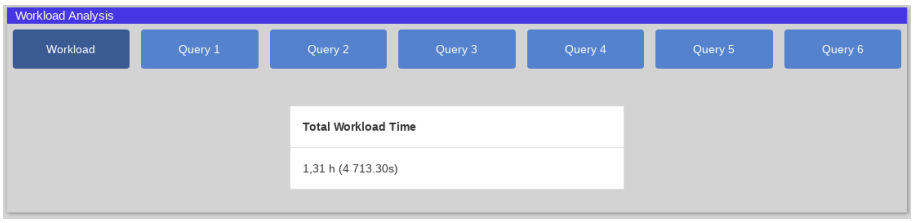


Figura 4.11: Riquadro per la selezione della query di cui visualizzare il dettaglio.

Selezionando una query dall'apposito riquadro, l'interfaccia ne mostra le informazioni. In particolare:

- il primo riquadro mostra la rappresentazione grafica dell'albero di esecuzione della query selezionata, indicando sia l'ordine che in quale modo vengono combinati i vary task types.
- i nodi dell'albero sono colorati con una tonalità di rosso (dalla più opaca alla meno opaca) in base al peso, in termini di tempo, che ha il nodo sul costo finale di esecuzione.
- la tabella in basso mostra la distribuzione del costo della query in termini di tempo di lettura, di scrittura e di network. Il costo è inoltre diviso nelle classi "affected" e "worked", che rispettivamente indicano la porzione della computazione delle varie operazioni che incide effettivamente sul tempo di esecuzione finale e il tempo totale speso dalle operazioni per effettuare il calcolo; la differenza è importante perchè alcuni task types effettuano operazioni in pipeline.

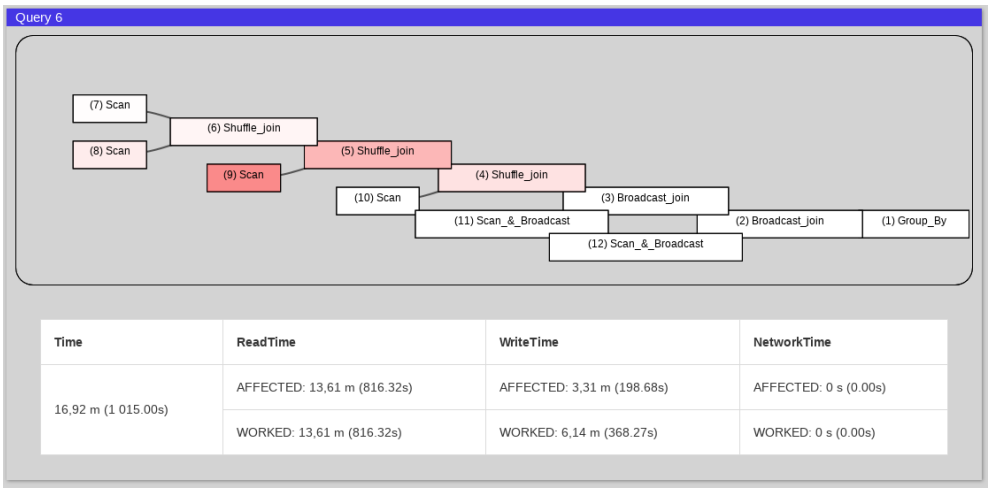


Figura 4.12: Riquadro contenente il dettaglio dell'esecuzione di una query.

Infine selezionando uno dei nodi dell'albero l'interfaccia ne mostra (figura 4.13) tutti i dettagli, che includono:

- il totale del tempo di esecuzione per il nodo, diviso in tempo di lettura, di scrittura e di network. In particolare, come visto nel capitolo 2, il costo di uno SC() è anche suddiviso nelle categorie Local, Rack e Cluster.
- il numero di wave eseguite dal nodo e le informazioni relative alle dimensioni dei dati in lettura e scrittura, includendo anche la slettività di eventuali predicati. Anche in questo caso se il nodo è uno SC(), le informazioni sono divise il Local, Rack e Cluster.
- le caratteristiche della tabella di input, che includono la cardinalità della selezione, la dimensione dei dati e il numero di partizioni sulle quali è memorizzata. Nel caso di nodi relativi ad operazioni di Join saranno presenti due tabelle di input.
- Le caratteristiche della tabella di output, che includono la cardinalità della selezione, la dimensione dei dati in output e il numero di partizioni sulle quali è memorizzata.

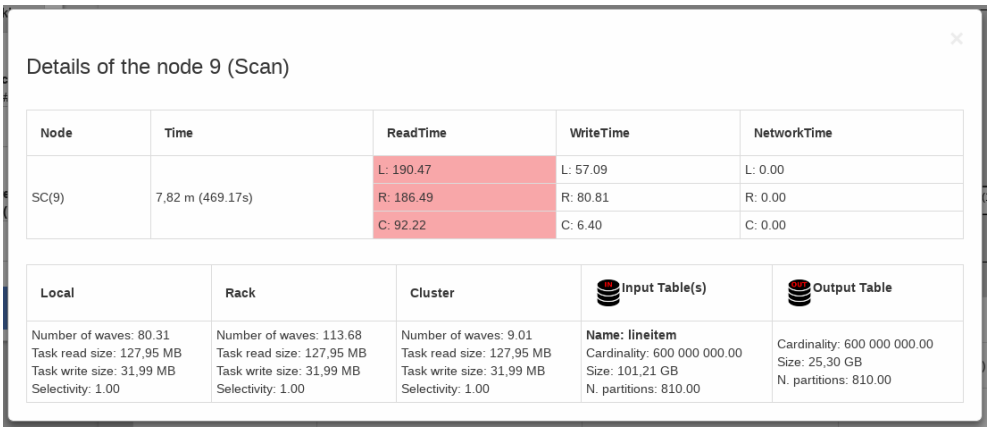


Figura 4.13: Riquadro contenente le informazioni che riguardano un nodo specifico dell'albero di esecuzione.

4.3 Analisi di Costo

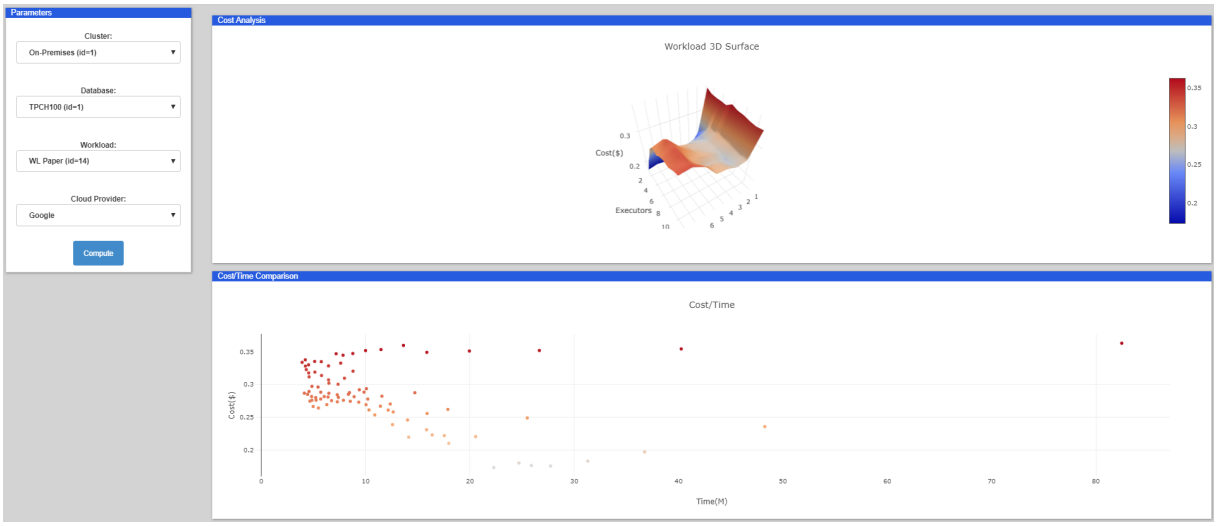


Figura 4.14: Interfaccia completa dell'analisi di costo.

La pagina di analisi di costo permette all'utente di effettuare un'analisi approfondita del costo dell'esecuzione di un determinato workload su un cluster

Cloud. Le informazioni necessarie, di cui si è parlato nel capitolo 3, vengono selezionate tramite il menù laterale (figura 4.14). In output si ottengono due grafici (figure 4.15, 4.16).

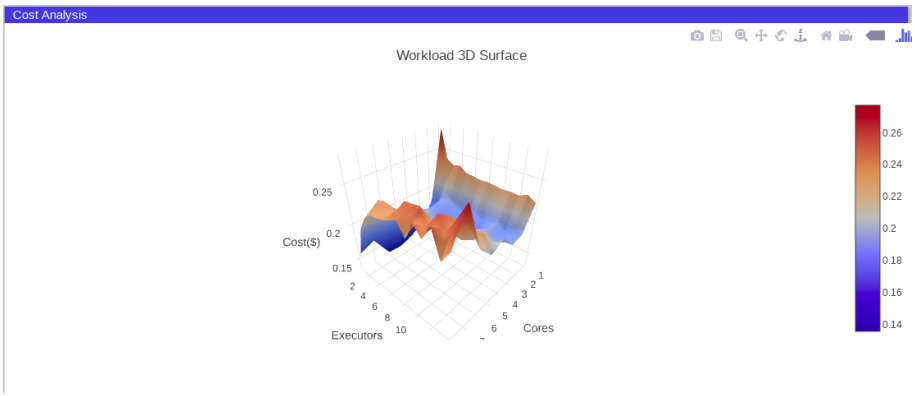


Figura 4.15: Grafico 3D della distribuzione del costo in denaro di un workload eseguito su un cluster Google

Il primo (figura 4.15) è un grafico 3D del costo dell'esecuzione calcolato utilizzando il modello visto nel capitolo 3. In particolare:

- l'asse **X** è costruito al variare del numero di executor del cluster di partenza;
- l'asse **Y** è costruito al variare del numero di core per executor del cluster di partenza;
- l'asse **Z**, ovvero la superficie vista in figura, è costruito incrociando i valori degli assi **X** e **Y** insieme agli altri parametri forniti in input al modello di costo;
- la superficie così costruita è colorata mediante un gradiente, che spazia dal blu al rosso, ottenuto facendo uso dell'intervallo dei valori prodotti in output dal modello di costo e disegnati sull'asse **Z**.

Il grafico 3D, e in generale tutti i grafici presenti nell'applicazione, offrono all'utente diverse funzionalità, tra cui:

- possibilità di manipolare il grafico mediante la rotazione degli assi;
- possibilità di effettuare zoom in e out;
- possibilità di effettuare il download dello snapshot del grafico;
- salvare il grafico su un servizio cloud esterno (plotly).

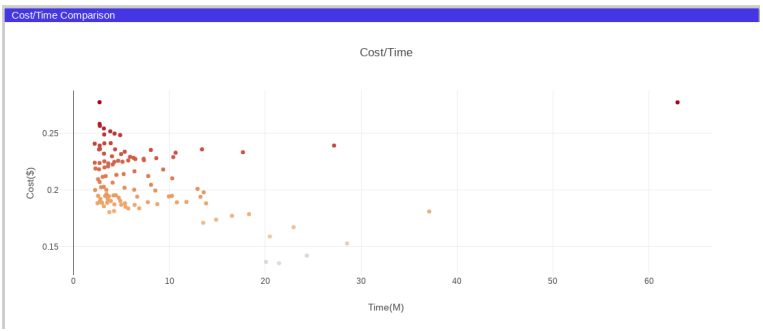


Figura 4.16: Grafico 2D della distribuzione del costo di un workload, messo in relazione al tempo di esecuzione.

Il secondo (figura 4.16) è un grafico 2D, costruito nel seguente modo:

- l'asse **X** è costituito dai valori ottenuti dal modello di costo (in tempo di esecuzione) al variare delle risorse assegnate per l'esecuzione;
- l'asse **Y** è costruito dai valori ottenuti dal modello di costo (in denaro) che, come spiegato nel capitolo 3, è una funzione dei parametri del cluster e del tempo di esecuzione;
- tutti i punti identificati dalle due coordinate mostrano, effettuando un hover, il numero di executor e il numero di core assegnati per ogni istanza;
- i punti del grafico sono colorati utilizzando un gradiente, che spazia dal bianco al rosso, ottenuto utilizzando l'intervallo dei valori prodotti dal modello di costo (in denaro) e disegnati sull'asse **Y**.

4.4 Analisi del Cluster

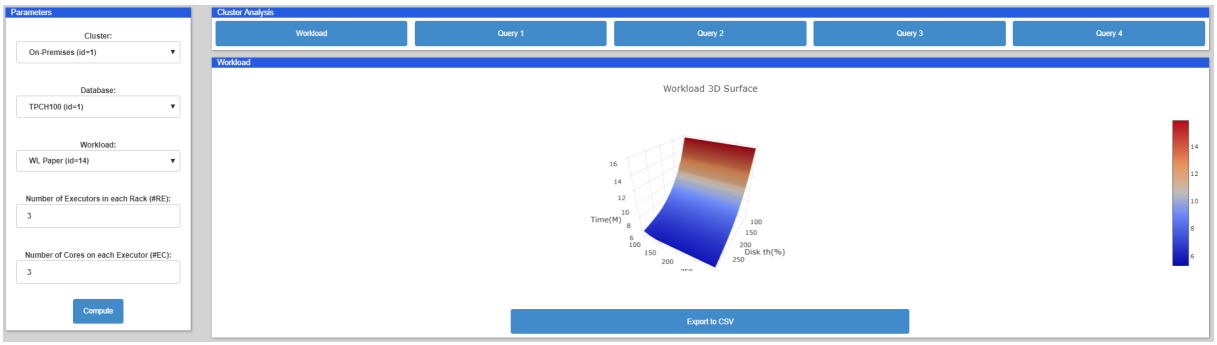


Figura 4.17: Interfaccia completa per l'analisi del cluster.

La pagina di analisi del cluster permette all'utente di valutare approfonditamente le prestazioni di un cluster prendendo come riferimento un dato workload. Per effettuare questa computazione si inseriscono le informazioni necessarie, illustrate nel capitolo 3, tramite il menù laterale (figura 4.17).

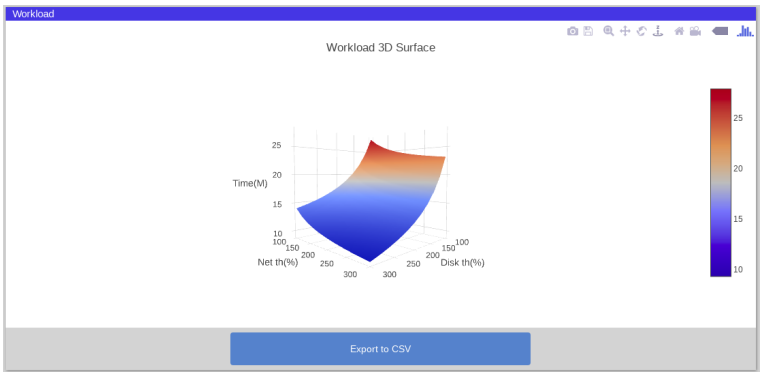


Figura 4.18: Grafico 3D della distribuzione del tempo di esecuzione di un workload, al variare dei throughput di disco e rete.

In output (figura 4.18) si ha un grafico 3D, costruito nel seguente modo:

- l'asse **X** è costruito variando il throughput della rete in un range che spazia dal 100% al 300%, con uno step di 20;

- analogamente all'asse **X**, l'asse **Y** è costruito variando il throughput del disco in un range che spazia dal 100% al 300%, con uno step di 20;
- l'asse **Z** è ottenuto incrociando i parametri degli assi **X** e **Y** insieme ai parametri dati in input per la computazione del modello di costo, espresso in termini di tempo di esecuzione;
- la superficie mostrata nel grafico è poi colorata mediante un gradiente che spazia dal blu al rosso, ottenuto utilizzando l'intervallo dei valori prodotti dal modello di costo e disegnati sull'asse **Z**.

L'interfaccia permette inoltre, analogamente all'analisi del workload, di eseguire la computazione per singola query piuttosto che per l'intero workload, selezionabile nel menù in alto. Oltre alle funzionalità di base dei grafici 3D, l'interfaccia da la possibilità di scaricare il risultato in formato CSV, permettendo di effettuare analisi sui dati più specifiche e approfondite.

4.5 Analisi di Performance

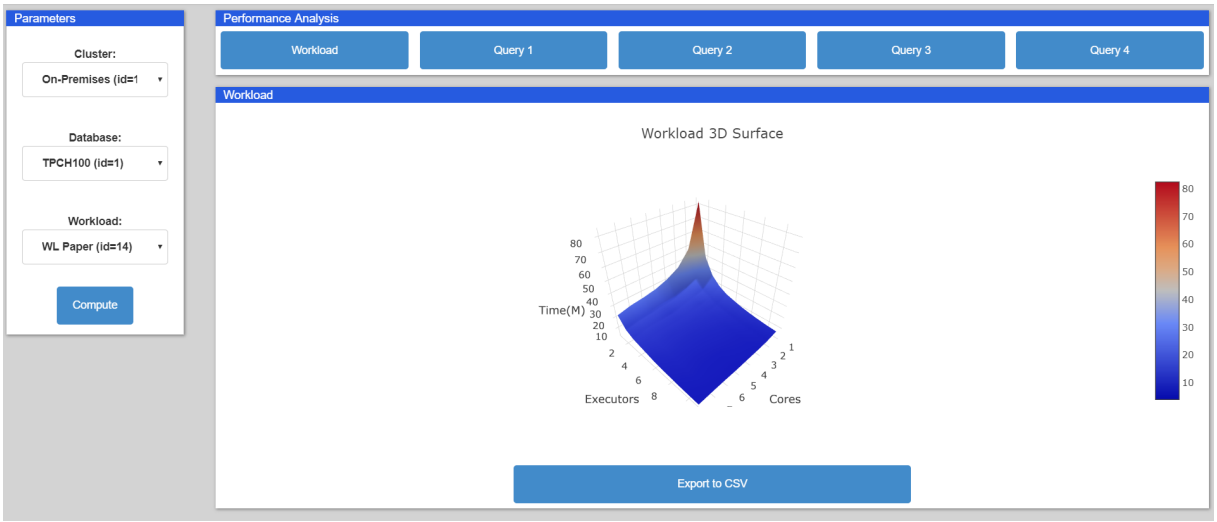


Figura 4.19: Interfaccia completa per l'analisi di performance.

La pagina di analisi delle performance permette all'utente di visualizzare le prestazioni di un determinato workload al variare delle risorse assegnatogli. Per effettuare questa computazione si inseriscono le informazioni necessarie, illustrate nel capitolo 3, tramite il menù laterale (figura 4.19).

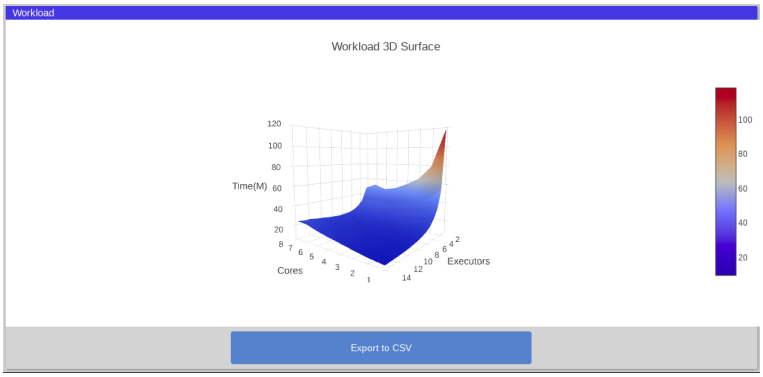


Figura 4.20: Grafico 3D che mostra il tempo di esecuzione di un determinato workload al variare delle risorse assegnate.

In output (figura 4.20) si ha un grafico 3D, costruito nel seguente modo:

- l'asse **X** è costruito variando il numero di executor a disposizione dell'applicazione, dove il massimo rappresenta il numero di nodi disponibili nel cluster;
- l'asse **Y** è costruito in maniera analoga all'asse **X**, ma al variare del numero di core per executor;
- i valori sull'asse **Z** sono ottenuti dal modello di costo incrociando i valori degli assi **X** e **Y** con gli altri parametri forniti i input. Il risultato è espresso in termini di tempo di esecuzione;
- la superficie costruita sull'asse **Z** è poi colorata mediante un gradiente che spazia dal blu al rosso, ottenuto utilizzando l'intervallo dei valori prodotti dal modello di costo.

Analogamente all'analisi del cluster, anche in questo caso l'interfaccia permette all'utente di scaricare i dati in formato CSV per effettuare analisi più approfondite.

Capitolo 5

Risultati Sperimentali

La maggioranza dei modelli di costo creati per SparkSQL calcolano i costi assumendo che i valori degli attributi siano uniformemente distribuiti, e il modello illustrato nel capitolo 2 non fa eccezione. In particolare, se l'assunzione di distribuzione uniforme non vale per un determinato attributo, il modello tenderà a sovrastimare o sottostimare le cardinalità in output delle selezioni e dei join. Per valutare l'impatto di di tali errori è stato eseguito un test modificando la distribuzione dei dati nel workload in modo che in ogni query il primo predicato di selettività e il primo predicato di join lavorino su un attributo non uniformemente distribuito. Le tabelle 5.1 e 5.2 mostrano i valori ottenuti dal modello senza la presenza dell'anomalia, che verranno poi confrontati con i valori ottenuti modificando la selettività.

Tabella 5.1: Risultati ottenuti in assenza di anomalie. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	1				2			
	#C	2	4	6	8	2	4	6	8
Q1		469.17	296.56	238.44	242.79	223.22	175.18	161.73	147.33
Q3		625.91	418.56	350.77	358.8	316.18	262.62	245.64	233.4
Q6		472.65	299.23	240.57	244.71	224.97	176.52	162.79	148.32
Q10		786.5	558.29	477.99	483.09	415.02	353.88	332.06	320.05

Tabella 5.2: Risultati ottenuti in assenza di anomalie. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	5				6			
	#C	2	4	6	8	2	4	6	8
Q1		98.23	82.03	75.66	70.17	80.64	65.94	64.72	57.42
Q3		131.05	114.58	112.41	104.26	107.83	92.61	90.52	89.24
Q6		98.94	82.58	76.1	70.59	81.22	66.39	65.09	57.76
Q10		163.18	142.22	142.09	132.64	134.76	118.03	110.78	114.96

5.1 Benchmark di Riferimento

Come già illustrato nel capitolo 3, il benchmark di riferimento è composto dalle query q_1, q_3, q_6 e q_{10} del benchmark TPC-H 100GB. Tuttavia in questo capitolo si vogliono analizzare con maggior dettaglio sia l'SQL che la relativa traduzione in piano di esecuzione per comprendere appieno la natura del workload e il relativo comportamento a fronte dei risultati ottenuti.

```
SELECT
l_returnflag,l_linestatus,sum(l_quantity),sum(l_extendedprice),
sum(l_extendedprice*(1-l_discount)),
sum(l_extendedprice*(1-l_discount)*(1+l_tax)),
avg(l_quantity),avg(l_extendedprice),
avg(l_discount),count(*)
FROM
tpch_100gb.lineitem
WHERE
l_shipdate<='1998-09-01'
GROUP BY
l_returnflag,l_linestatus
```

Figura 5.1: Rappresentazione SQL della query Q1.

Figura 5.2: Rappresentazione grafica dell'esecuzione della query Q1.

La prima query, q1 (figure 5.1 e 5.2), a livello di task type è piuttosto semplice ed è composta solamente da due operazioni: uno scan seguito da un group by. Come si evince dalla corrispondente query SQL, lo scan rappresenta le varie operazioni di selezione e aggregazione eseguite sull'unica tabella interrogata, **lineitem**, che di fatto costituisce la tabella con la maggior quantità di record del database. La selezione computata dall'operazione di Scan ha una selettività pari circa a 1.

```
SELECT
l_orderkey,sum(l_extendedprice*(1-l_discount)),
o_orderdate,o_shippriority
FROM
tpch_100gb.customer,tpch_100gb.orders,tpch_100gb.lineitem
WHERE
c_mktsegment='BUILDING' and c_custkey = o_custkey
and l_orderkey = o_orderkey and o_orderdate<'1995-03-15'
and l_shipdate>'1995-03-15'
GROUP BY
l_orderkey,o_orderdate,o_shippriority
```

Figura 5.3: Rappresentazione SQL della query Q3.

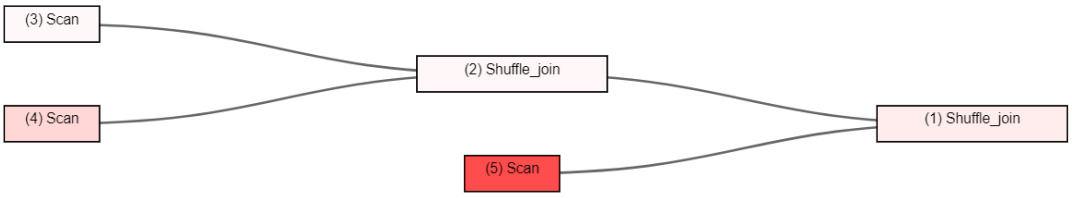


Figura 5.4: Rappresentazione grafica dell'esecuzione della query Q3.

La seconda query, q3 (figure 5.3 e 5.4), a livello di task type è più complessa e rispecchia perfettamente la struttura della query SQL da cui è derivata. Essa infatti è composta da tre operazioni di scan sulle tabelle **customer**, **orders** e **lineitem**, e da due operazioni di Join (shuffle join) che corrispondono ai due predicati di join tra le tabelle. In questo caso, in presenza di più operazioni di scan, teniamo in considerazione solamente la prima, che ha effetto sulla tabella **orders**, i cui predicati generano una selettività di 0.5. Allo stesso modo teniamo in considerazione solamente la prima operazione di join, computato tra le tabelle **customer** e **orders**, che ha una cardinalità pari a $15*10^6$.

```
SELECT
sum(l_extendedprice*l_discount)
FROM
tpch_100gb.lineitem
WHERE
l_shipdate>='1994-01-01'
and l_shipdate<'1995-01-01'
and l_discount between 0.05 and 0.07
and l_quantity<24
```

Figura 5.5: Rappresentazione SQL della query Q6.

Figura 5.6: Rappresentazione grafica dell'esecuzione della query Q6.

La terza query, q6 (figure 5.5 e 5.6), è estremamente semplice e a livello di task type è costituita da una singola operazione di scan sulla tabella lineitem. In questo caso i predicati generano una selettività pari a 0.05.

```
SELECT
c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)),
c_acctbal, n_name, c_address, c_phone, c_comment
FROM
tpch_100gb.customer, tpch_100gb.orders,
tpch_100gb.lineitem, tpch_100gb.nation
WHERE
c_custkey = o_custkey and l_orderkey = o_orderkey
and o_orderdate>='1993-10-01' and o_orderdate<'1994-01-01'
and l_returnflag='R' and c_nationkey = n_nationkey
GROUP BY
c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment
```

Figura 5.7: Rappresentazione SQL della query Q10.

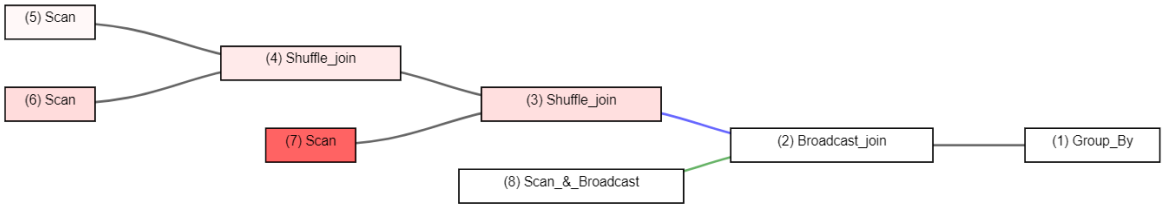


Figura 5.8: Rappresentazione grafica dell'esecuzione della query Q10.

L'ultima query, q10 (figure 5.7 e 5.8), è la più complessa ed è costituita dall'aggregazione del join a tre vie tra le tabelle **customer**, **orders**, **lineitem** e **nation**. Anche in questo caso, in presenza di più operazioni di Scan (e ScanBroadcast) teniamo in considerazione soltanto la prima, che ha effetto sulla tabella **orders**, i cui predicati generano una selettività di 0.38. Allo stesso modo teniamo in considerazione solamente il primo join, computato tra le tabelle **customer** e **orders**, che ha una cardinalità pari a 41 666 666.

5.2 Variazione della cardinalità delle Selezioni

Il test di variazione della cardinalità delle selezioni è stato eseguito sull'intero workload, alle cui query è stata applicata un'anomalia sul numero totale delle tuple restituite dalla prima operazione di scan. Questa incide solamente sul primo predicato di selettività, ed è pari al 200% nella prima prova e al 50% nella seconda. Le tabelle 5.3 e 5.4 mostrano i risultati ottenuti applicando uno scostamento del 200%, mentre le tabelle 5.5 e 5.6 mostrano i risultati ottenuti con una variazione del 50%.

Tabella 5.3: Risultati ottenuti in presenza di un'anomalia del 200% sul primo predicato di selettività. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	1				2			
	#C	2	4	6	8	2	4	6	8
Q1		469.17	296.56	238.44	242.79	223.22	175.18	161.73	147.33
Q3		655.18	447.72	380.18	385.56	335.7	281.41	264.09	252.14
Q6		473.71	300.05	241.23	245.31	225.5	176.93	163.12	148.62
Q10		978.68	725.66	633.8	633.65	536.18	466.99	440.6	429.19

Tabella 5.4: Risultati ottenuti in presenza di un'anomalia del 200% sulla cardinalità in output della prima selezione. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	5				6			
	#C	2	4	6	8	2	4	6	8
Q1		98.23	82.03	75.66	70.17	80.64	65.94	64.72	57.42
Q3		137.01	119.96	118.39	110.07	112.67	97.4	94.49	94.36
Q6		99.15	82.75	76.23	70.72	81.4	66.52	65.21	57.86
Q10		202.14	176.23	178.93	168.02	167.16	149.07	135.75	146.79

Tabella 5.5: Risultati ottenuti in presenza di un'anomalia del 50% sulla cardinalità in output della prima selezione. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	1				2			
	#C	2	4	6	8	2	4	6	8
Q1		469.17	296.56	238.44	242.79	223.22	175.18	161.73	147.33
Q3		611.27	405.38	338.32	346.48	306.42	253.23	236.41	224.03
Q6		471.41	298.28	239.82	244.03	224.34	176.04	162.41	147.97
Q10		690.41	474.6	401.05	407.81	354.44	297.33	277.79	265.47

Tabella 5.6: Risultati ottenuti in presenza di un'anomalia del 50% sul primo predicato di selettività. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	5				6			
	#C	2	4	6	8	2	4	6	8
Q1		98.23	82.03	75.66	70.17	80.64	65.94	64.72	57.42
Q3		128.07	111.9	109.42	101.35	105.41	90.21	88.53	86.68
Q6		98.69	82.39	75.94	70.44	81.01	66.23	64.96	57.63
Q10		143.7	125.22	123.66	114.95	118.56	102.51	98.3	99.05

Confrontando i risultati ottenuti con le prove effettuate in un ambiente privo di anomalie, si possono fare le seguenti considerazioni:

- **I costi relativi alle query Q1 e Q6 non variano**: ciò accade perchè il predicato di selettività non influenza il costo di lettura, poichè le tabelle non sono memorizzate in un formato compresso tale da permettere il pushdown della selezione direttamente alla sorgente dei dati. Inoltre, nonostante la tabella in output sia influenzata dall'anomalia, questa non viene utilizzata in nessun'altra operazione e pertanto il costo non subisce alcuno scostamento.

- **I costi relativi alle query Q3 e Q10 subiscono delle variazioni**: al contrario delle query Q1 e Q6, in questo caso la tabella di output ottenuta applicando l'anomalia alla selezione influenza le operazioni a seguire, variando pertanto il costo totale. In entrambi i casi il modello rimane consistente (il costo aumenta se la cardinalità raddoppia e si abbassa se la cardinalità dimezza), e provoca un aumento dell'errore di previsione, evidenziato in [1], dell' 8.7%. In definitiva l'errore causato dall'anomalia è sufficientemente contenuto da permettere al modello di fornire una previsione accettabile, anche se alcuni attributi non sono distribuiti uniformemente.

5.3 Variazione delle cardinalità dei Join

Il test di variazione della cardinalità dei join è stato eseguito sulle query Q3 e Q10, alle quali è stata applicata un'anomalia sul numero totale delle tuple restituite dalla prima operazione join. L'anomalia incide solamente sul primo di essi, ed è pari al 200% nella prima prova e al 50% nella seconda. Le tabelle 5.7 e 5.8 mostrano i risultati ottenuti applicando uno scostamento del 200%, mentre le tabelle 5.9 e 5.10 mostrano i risultati ottenuti con una variazione del 50%.

Tabella 5.7: Risultati ottenuti in presenza di un'anomalia del 200% sulla cardinalità in output del primo join. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	1				2			
	#C	2	4	6	8	2	4	6	8
Q3		640.55	430.28	360.74	368.22	324.13	269.37	251.59	239.18
Q10		966.48	713.47	619.44	620.97	526.53	456.96	430.18	418.39

Tabella 5.8: Risultati ottenuti in presenza di un'anomalia del 200% sulla cardinalità in output del primo join. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	5				6			
	#C	2	4	6	8	2	4	6	8
Q3		133.99	116.94	114.77	106.45	110.41	94.89	92.22	91.34
Q10		199.63	173.72	175.91	165.01	165.27	146.98	133.86	144.28

Tabella 5.9: Risultati ottenuti in presenza di un'anomalia del 50% sulla cardinalità in output del primo join. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	1				2			
	#C	2	4	6	8	2	4	6	8
Q3		618.58	412.70	345.78	354.09	312.20	259.25	242.66	230.51
Q10		696.51	480.7	407.27	414.15	359.27	302.34	282.99	270.87

Tabella 5.10: Risultati ottenuti in presenza di un'anomalia del 50% sulla cardinalità in output del primo join. #E indica il numero di executor, mentre #C indica il numero di core.

	#E	5				6			
	#C	2	4	6	8	2	4	6	8
Q3		129.58	113.40	111.23	103.16	106.54	91.46	89.67	88.19
Q10		144.95	126.48	125.17	116.46	119.5	103.55	99.24	100.31

L'anomalia applicata ai join influenza le operazioni a seguire, variando pertanto il costo totale. In entrambe le prove il modello rimane consistente (il costo aumenta se la cardinalità raddoppia e si abbassa se la cardinalità dimezza), e provoca un aumento dell'errore di previsione, evidenziato in [1], del 9.8% . In conclusione, analogamente al caso precedente, l'errore causato dall'anomalia è sufficientemente contenuto da permettere al modello di fornire comunque una previsione accettabile.

Conclusioni e Sviluppi Futuri

L'avvento Big Data ha generato una rivoluzione nella gestione delle grandi quantità di dati. Negli ultimi anni questo fenomeno ha spinto le aziende e i centri di ricerca allo sviluppo di un insieme di tecnologie in grado di gestire i Big Data, tra le quali Hadoop si è affermato come una delle piattaforme più apprezzate. Le ragioni di questo successo sono legate a diversi fattori: la piattaforma, oltre ad essere open-source, è semplice da estendere nella maggior parte delle sue componenti ed è caratterizzata dalla possibilità di eseguire la computazione su un cluster composto da hardware di commodity. Tra i componenti più importanti dell'ecosistema Hadoop figura Spark, una piattaforma che rende il cluster computing semplice e flessibile, supportando diversi tipi di computazioni. Nonostante l'ecosistema Hadoop sia largamente adottato, esso risulta ancora immaturo e privo di funzionalità di alto livello per l'analisi complessa di dati.

In questo contesto è stata sviluppata un'applicazione che permette all'utente di svolgere analisi complesse sia dal punto di vista delle performance di un workload che dal lato delle prestazioni di un cluster. In particolare sono state implementante quattro interfacce web, che mettono a disposizione dell'utilizzatore altrettante funzionalità. La prima, l'analisi del workload, permette all'utente di calcolare il tempo di esecuzione di un insieme di query e visualizzare i dettagli più rilevanti della computazione. La seconda, l'analisi di costo, permette all'utente di calcolare il costo in denaro di un particolare workload su un cluster cloud, mettendolo a confronto con il relativo tempo di esecuzione al variare delle risorse assegnate. La terza, l'analisi del cluster,

dà la possibilità all'utilizzatore di calcolare il tempo di esecuzione di un workload al variare dei throughput del disco e della rete, permettendo di valutare l'impatto di un eventuale upgrade dell'hardware di un determinato cluster. Infine la quarta interfaccia, l'analisi di performance, permette all'utente di calcolare il tempo di esecuzione di un dato workload al variare delle risorse assegnate. In generale l'applicazione si dimostra estremamente semplice da utilizzare e interessante, mettendo a disposizione dell'utilizzatore strumenti per l'analisi molto potenti, nascondendo tutti i dettagli di programmazione tipici della piattaforma Spark. Il tool ha inoltre permesso di valutare in maniera semplice l'accuratezza del modello di costo a fronte dell'introduzione di anomalie.

Lo sviluppo dell'applicazione si apre a diversi sviluppi futuri. In particolare sarebbe interessante permettere l'integrazione automatica del tool con il modulo Spark SQL, permettendo di passare direttamente dalle query SQL al workload pronto all'uso. Attualmente è necessario inserire la query SQL sul modulo e generare un piano di esecuzione, da inserire manualmente nell'applicazione. Infine l'implementazione del modello di costo utilizzata computa il costo analitico di una query GPSJ assumendo che le risorse siano sufficienti per fare si che nessuno *straggler* si verifichi. Uno *straggler* è un task che svolge la computazione in maniera molto meno efficiente rispetto ai propri simili, a causa del numero insufficiente di risorse assegnate. Pertanto, nelle future release dell'applicazione, l'implementazione del modello di costo terrà in considerazione il tempo di esecuzione in presenza di questi eventi.

Bibliografia

[1] Matteo Golfarelli, Lorenzo Baldacci. A Cost Model for SPARK SQL, 2017.

[2] Eugenio Pierfederici. Progettazione e sviluppo di un simulatore di costo per Spark SQL, 2017.

[3] Hadoop Architecture. hadoop.apache.org, 2017.

[4] Apache Hive. hive.apache.org, 2018.

[5] Faster Big Data on Hadoop with Hive and RCFile. Christian Prokopp, 2014.

[6] Thusoo, Ashish; Sarma, Joydeep Sen; Jain, Namit; Shao, Zheng; Chakka, Prasad; Anthony, Suresh; Liu, Hao; Wyckoff, Pete; Murthy, Raghotham. Hive: a warehousing solution over a map-reduce framework, 2009.

[7] Apache Parquet. parquet.apache.org, 2018.

[8] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin†, Ali Ghodsi, Matei Zaharia. Spark SQL: Relational Data Processing in Spark, 2015.

[9] Deep Dive into Spark SQL's Catalyst Optimizer. databricks.com, 2015.

[10] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, 2004.

[11] Spark SQL, DataFrames and Datasets. spark.apache.org, 2018.

Ringraziamenti

Innanzitutto desidero ringraziare i miei genitori, Elisabetta e Vincenzo, e mia sorella Camilla, per il loro amore e per il grande supporto che mi hanno dato durante tutto il corso dei miei studi. Vorrei ringraziare la mia ragazza, Marina, per il suo grande amore e per il supporto che mi ha dato in tutti questi anni; senza di lei sarebbe stato tutto molto più difficile. Vorrei ringraziare tutti i miei amici di corso che, nonostante la mia indole scontrosa, mi hanno sempre sostenuto e apprezzato per quello che sono. Vorrei ringraziare i miei nonni, per l'amore, la sicurezza e il sostegno che mi hanno dato in tutti questi anni. Ringrazio di cuore il professor Matteo Golfarelli per la grande opportunità di crescita che mi ha dato, nonchè per la sua disponibilità e grande professionalità. Ringrazio infine il mio caro nonno Gigi, che ora non è più tra noi, per la passione e la forza che mi ha trasmesso; senza il suo consiglio non sarei la persona che sono.