Alma Mater Studiorum Università di Bologna

Campus di Cesena

Scuola di Scienze

Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione e sviluppo di un simulatore di costo per Spark SQL

Presentata da:

Eugenio Pierfederici

Relatore:

Matteo Golfarelli

2 marzo 2017

Alla mia famiglia, alla mia fidanzata e ai compagni di corso che hanno creduto in me e mi hanno sempre supportato durante questo primo percorso universitario.

Indice

Capitolo 1

Introduzione

Si è sempre più consapevoli del fatto che la quantità dei dati generati dalle nuove tecnologie cresce a dismisura. In questo fenomeno il ruolo principale va sicuramente attribuito ai social network che producono quotidianamente una mole impressionante di dati. Negli ultimi anni a questi si sta aggiungendo il numero sempre crescente di dispositivi di IoT (*Internet Of Things*), che periodicamente generano ed inseriscono in rete nuove informazioni.

Le aziende stanno sempre più valorizzando l'importanza di una buona gestione di tali dati e della necessità di adottare strumenti che siano capaci di ricavarne informazioni. Gestire bene i dati infatti, significa per un'azienda fornirsi di uno strumento in grado di migliorarne organizzazione e competitività. Per rispondere a questa esigenza sono nati negli anni sempre più numerosi sistemi che permettono l'archiviazione ed analisi di tali informazioni in maniera molto efficiente. Si tratta di framework sia proprietari, sia open-source. Fra questi ultimi il sistema di memorizzazione dei dati più diffuso è Hadoop, cui si accompagna Spark come strumento di analisi. La capacità di memorizzazione dei dati e l'efficienza nell'effettuazione di analisi su dei cluster come Hadoop è riconducibile alla distribuzione del carico di dati su diversi host. Sfruttando tali host, Spark è in grado di garantire elevate prestazioni in fase di analisi, che molto spesso sono eseguite direttamente in memoria centrale. All'interno di Spark, un modulo importante ai fini della tesi è Spark SQL. Tale componente permette l'esecuzione di query SQL all'interno dei suddetti cluster.

L'interesse crescente verso questo settore ha portato, soprattutto in sistemi open-source come Hadoop, ad uno straordinario sviluppo che ha avuto come conseguenza il proliferare di studi scientifici. Diversi di questi studi si concentrano su sulle funzioni di costo Map-Reduce, ma non esistono ancora studi che abbiano generato funzioni di costo per Spark SQL. Lo scopo di questa tesi è la presentazione di una funzione di costo per Spark SQL e lo sviluppo di un'applicazione che, sfruttando tale funzione, stimi il tempo di esecuzione di una query SQL. Un simile modello di costo permetterebbe all'utente, utilizzatore

dell'applicazione, non solo di conoscere il tempo complessivamente impiegato per eseguire la query sul suo cluster, ma anche i tempi parziali di tale esecuzione. Questi ultimi dati forniscono ad un ottimizzatore le informazioni necessarie e sufficienti per migliorare la performance di esecuzione della query stessa.

L'elaborato è strutturato in 4 capitoli. Nel primo tratto i Big Data, concetto fondamentale nella tesi, la struttura di Hadoop, HDFS e la funzione MapReduce. Gli ultimi due sono elementi che vanno notevolmente ad influire sulle performance di esecuzione di una query, perciò è importante comprenderne il funzionamento. Nel secondo capitolo tratto la struttura di Spark, il modo in cui esegue le query e, soprattutto, il modulo Spark SQL. Questo modulo è costituito da Catalyst, traduttore ed ottimizzatore di query SQL. È questo il componente che genera il piano di esecuzione che andremo ad analizzare. Il terzo capito tratta il modello di costo che è stato teorizzato. Al suo interno vengono trattate la grammatica adottata, le funzioni per il calcolo dei costi ed i parametri principali che li influenzano. Nell'ultimo capitolo tratto gli aspetti fondamentali che hanno caratterizzato lo sviluppo dell'applicazione, la sua struttura ed architettura.

Capitolo 2

I Big data e la piattaforma Hadoop

2.1 L'evoluzione dei dati e della loro gestione

Negli ultimi anni si è presentata sempre più spesso la necessità di analizzare enormi quantità di dati, non sempre strutturati. La quantità di dati memorizzati aumenta ogni giorno; i tradizionali strumenti consentono di memorizzare ed accedere a questa mole di informazioni, ma non sono in grado di analizzarla. Come conseguenza si ha che la percentuale di dati che il business è in grado di processare sta calando molto velocemente. Questa quantità di informazioni non analizzabili ha fatto nascere il fenomeno dei Big Data: quantità tali di dati ed informazioni che non possono essere analizzati utilizzando processi e strumenti tradizionali.

Tradizionalmente le basi di dati risiedono su *database relazionali* e vengono gestiti dai RDBMS (*Relational Database Management System*). Si tratta di sistemi OLTP (*On-Line Transation Processing*: sistemi ottimizzati principalmente per le operazioni transazionali, cioè sono studiati ed ottimizzati per facilitare le operazioni di inserimento, modifica e cancellazione dei dati. Sono quindi resi molto efficienti e sicuri in queste operazioni a discapito delle operazioni di analisi. Per questo motivo già in passato si sono iniziati a sfruttare nuovi ed appositi database: i (*Data Warehouse*), che permettono di ottimizzare le performance di interrogazione. Questi sistemi sono però inadatti all'inserimento di grandi quantità di dati in un arco piuttosto ristretto di tempo e nel tempo il problema potrebbe diventare ingestibile sia in termini di risorse che di costi. Un esempio di questo problema lo troviamo nell'ambito produttivo: si vogliono memorizzare tutte le informazioni generate da un elevato numero di sensori (anche migliaia). Ogni sensore genera dati periodicamente; le rilevazioni possono avvenire ad intervalli di tempo anche molto piccoli e ciò, insieme alla quantità di sensori, porta a produrre una gran quantità di informazioni da memorizzare in pochissimo tempo.

Le problematiche nate da queste nuove necessità hanno richiesto nuove tecnologie (diverse dai RDBMS) che consentano di ottenere potenza di calcolo e scalabilità senza costi proibitivi. Inoltre le nuove esigenze di analisi di tutti questi dati, per esempio per ottenere statistiche di mercato, ha richiesto sistemi non più OLTP bensì OLAP (*On-Line Analytical Processing)*, che sono ottimizzati proprio per un'analisi interattiva e veloce di grandi quantità di dati.

2.2 Caratteristiche dei Big Data

I Big Data rappresentano tutti quei dati presenti in enormi volumi e sono caratterizzati da tre fattori: volume, varietà e velocità.

- *volume*: con l'avvento delle tecnologie di IoT e degli *smart device*, la quantità di dati quotidianamente generati e memorizzati è diventata mostruosa. Un aspetto fondamentale dei Big Data è che venga memorizzato ogni singolo dato. È un aspetto fondamentale nel mondo del business in quanto più sono i dati e maggiore è la precisione delle previsioni che vengono effettuate.

- *varietà*: un altro aspetto fondamentale dei Big Data è la varietà dei dati. I dati possono essere più o meno complicati ed in certi casi risulta pressoché impossibile dargli una struttura predefinita. I formati che possono avere sono strutturati, semi-strutturati o non strutturati. Non essendo possibile fornire a priori una struttura dei dati nella maggior parte delle situazioni, i RDBMS non sono più sufficienti. La gestione dei dati semi-strutturati è possibile anche con database NoSql, che al contrario dei RDBMS non richiedono di definire la struttura prima dello sviluppo, ma sarebbe una soluzione parziale in quanto i database NoSql non risolvono gli altri problemi posti dai Big Data. In effetti la quantità di dati non strutturati è notevolmente maggiore rispetto a quelli strutturati; per restare concorrente, un'azienda deve quindi essere in grado di gestire sia dati strutturati che non.

- *velocità*: l'ultima caratteristica, ma non per questo meno importante, dei Big Data è la velocità. Si pensi anche solo al numero di transazioni quotidiane che una banca deve memorizzare, o alla quantità di informazioni provenienti dai sensori di una catena di montaggio o anche al numero di post che quotidianamente Facebook deve occuparsi di salvare. Il business richiede di poter analizzare queste informazioni quasi in real-time; è spesso incisiva la velocità con la quale si individuano tendenze o opportunità.

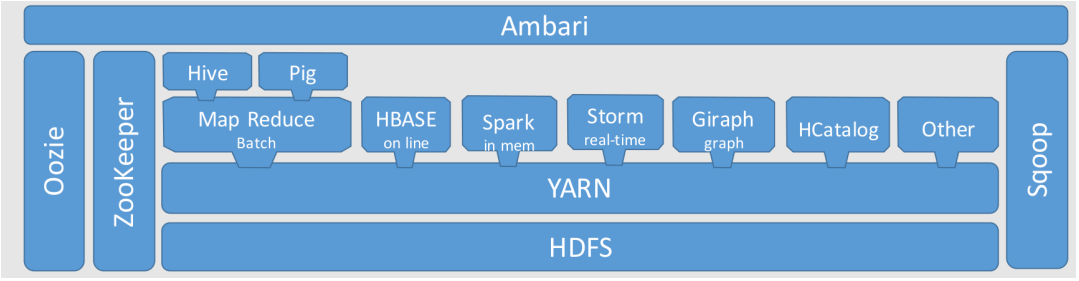


FIGURA 2.1

In conclusione quando si parla di Big Data si parla di enormi quantità di dati di diversa forma che vanno memorizzate ed analizzate a notevoli velocità.

2.3 Un framework open-source: Hadoop

Le tecnologie che si sono sviluppate nel tentativo di soddisfare queste nuove necessità non sono molti. Ad ogni modo il framework più diffuso in ambito Open Source è, al momento, Hadoop. Hadoop è un framework di Apache affidabile e scalabile che permette la gestione di grandi quantità di dati in maniera distribuita.

Hadoop è nato all'interno del progetto Nutch, un motore di ricerca open-source che si occupa di navigare il web in maniera sistematica. Hadoop è basato sulle tecnologie di Google File System e Google MapReduce. Alla nascita si trattava solo di un componente di Nutch che era in grado di migliorarne le prestazioni, ma è stato successivamente acquistato da Yahoo. Da allora Yahoo è il più grande contributore a questo progetto nonché principale utilizzatore. Prima di Hadoop le elaborazioni di grandi quantità di dati erano effettuate con sistemi di *Hight Performance Computing* e *Grid Computing*. Hadoop, a differenza di questi sistemi, fornisce un insieme di librerie di alto livello e sfrutta la replicazione dei dati sui singoli nodi per migliorare i tempi di accesso.

2.4 Architettura di Hadoop

I moduli principali presenti fin dalla prima versione di Hadoop sono:

- **Hadoop common**: uno strato software comune che fornisce funzioni di supporto agli altri moduli.
- **HDFS** (Hadoop Distributed File System): HDFS è il filesystem distribuito di Hadoop progettato appositamente per essere eseguito su commodity

hardware. La distribuzione del file system consiste nel memorizzare i dati all'interno dei nodi di una rete di macchine. Questa suddivisione dei dati in più nodi è necessaria quando la mole di dati diventa troppo grande per la capacità di memorizzazione di una singola macchina. Gestire un file system distribuito è ovviamente più complesso rispetto a quelli normali, in quanto si basa sulla comunicazione di rete, ma fornisce anche numerosi vantaggi come la velocità nel recupero dei dati e la scalabilità.

- **Hadoop YARN** (Yet Another Resource Negotiator): un framework per lo scheduling dei Job e la gestione delle risorse del cluster.

- **Hadoop MapReduce**: è un framework software sviluppando dalla Google che si occupa della schedulazione ed esecuzione dei calcoli su sistemi distribuiti. Il principio su cui si basa è quello del *divide et impera*: operazioni complesse, che lavorano su grandi moli di dati, vengono suddivise in operazioni più piccole, che possono essere completate autonomamente. Infine questi risultati parziali vengono riuniti in un unico risultato finale.

Questi strumenti rendono Hadoop un sistema altamente affidabile e scalabile. L'affidabilità è dovuta al fatto che è pensato per cluster di commodity hardware: più sono grandi i cluster maggiore è la probabilità di un guasto o di un problema, quindi il sistema permette la sostituzione dei nodi guasti nella maniera più semplice possibile. È scalabile in quanto permette di estendere o ridurre la capacità computazionale semplicemente aggiungendo o rimuovendo dei nodi al cluster.

Con la seconda versione di Hadoop sono arrivati altri nuovi moduli, in particolare è nato il modulo Spark sul quale si sviluppa la funzione di calcolo dei costi che vedremo nei prossimi capitoli.

Vi sono due tipologie di nodi in un cluster Hadoop:

- **Master**: vi sono eseguiti i processi di coordinamento di HDFS e MapReduce

- **Worker**: vengono usati per la memorizzazione dei dati e per il calcolo

2.5 HDFS: il *File System* distribuito

Hadoop Distributed File System (HDFS) è stato progettato per memorizzare files di grandi dimensioni e per la gestione dei flussi. I principali aspetti che lo caratterizzano sono:

- *Very large files*: non esiste un limite esplicito per la dimensione dei files, si tratta comunque di centinaia di megabytes, gigabytes, terabytes. Ad oggi ci sono clusters Hadoop (come ad esempio quello di Yahoo!) che memorizzano diversi petabytes di dati.

- *Streaming data access*: HDFS si basa sul *write-once, read-many-times* pattern. Questo pattern implica che un dataset viene copiato e analizzato (in parte o per intero) numerose volte, quindi la latenza di lettura del primo record è meno importante del tempo necessario ad accedere all'intero dataset.

- *Commodity hardware*: Hadoop non ha bisogno di usare hardware estremamente costoso ed altamente affidabile; infatti il suo file system è progettato in maniera tale da saper gestire la rottura di un nodo (molto frequente soprattutto in cluster di grandi dimensioni) ed in genere continua a lavorare senza che l'utente nemmeno se ne accorga.

Però, come sempre succede, ottimizzando determinati aspetti si perde efficienza in altri. È giusto fare una nota su quali sono le situazioni nelle quali un HDFS non è la miglior soluzione:

- *Low-latency data access*: HDFS è ottimizzato per fornire un altro throughput di dati e questo potrebbe andare a discapito della latenza. Applicazioni che richiedono una bassa latenza non lavoreranno bene con questo file system.

- *Lots of small files*: In un File System HDFS il *NameNode* è il pezzo che si occupa di memorizzare la struttura di tutti i files e la posizione di ognuno all'interno del cluster. Siccome il NameNode memorizza anche i metadata, maggiore è il numero di files e maggiore sarà la memoria occupata nel NameNode. Quindi il numero di files dipende dalla dimensione della memoria del namenode.

- *Multiple writers, arbitrary file modifications*: HDFS può essere scritto da un solo writer e la scrittura viene fatta sempre alla fine del file. Non è supportata in HDFS la scrittura da parte di più writer o la modifica in un punto specifico del file, per ora. Dunque è inadatto per quelle esigenze.

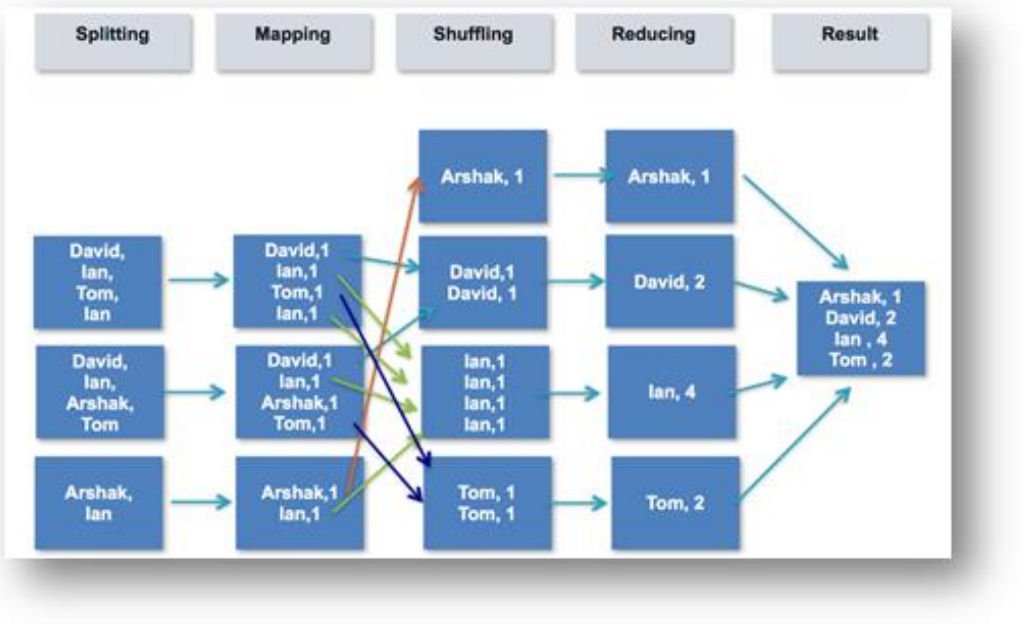


FIGURA 2.2

2.6 MapReduce

MapReduce è il framework Google attraverso il quale è possibile elaborare grandi quantità di dati in parallelo, sfruttando la computazione distribuita. Il principio sfruttato da MapReduce è il *Divide et impera*, ovvero la suddivisione di un'operazione di calcolo complessa in più operazioni computazionalmente più semplici. Queste operazioni vengono svolte in maniera autonoma e al termine di ciascuna parte i risultati parziali vengono uniti e riprocessati per portare al risultato finale.

L'applicazione che viene eseguita in ambiente Hadoop è definita come *Job MapReduce*. Gli elementi che compongono un Job sono generalmente:

- *Input reader*
- *Map function*
- *Partition function*
- *Compare function*
- *Reduce function*
- *Output writer*

Le fasi di map e di reduce vengono suddivise in un certo numero di task ed eseguite in parallelo sul cluster Hadoop. I task sono di due tipi in base all'attività che devono svolgere: *map task* e *reduce task*. I singoli task vengono poi eseguiti sui nodi del cluster adibiti al calcolo (che tipicamente sono gli stessi adibiti alla memorizzazione dei dati). Ogni Job è sempre caratterizzato da un *ApplicationMaster*: il responsabile della specifica applicazione richiesta. Inoltre per ogni nodo vi sono sempre un *ResourceManager*, che si occupa di allocare le risorse computazionali per ogni singolo container del cluster, ed un *NodeManager*, che lancia e monitora la computazione dei container sul nodo.

2.6.1 Map function

La funzione di Map ha il compito di processare i dati di input allo scopo di prepararli per la funzione di reduce. Generalmente i dati di input sono nella formi di un file o una cartella all'interno dell'Hadoop file system (HDFS). L'input file viene passatto alla funzione di map riga per riga. La funzione processa i dati e crea diversi piccoli chunks di dati. Per esempio, se l'applicazione deve effettuare un conteggio di tutte le parole, la funzione di map avrà in input ogni riga e restituirà in output un'accoppiata chiave/valore per ogni parola. In output ogni chiave sarà la parola e il valore è il numero di volte che la parola è ricorsa all'interno della linea.

2.6.2 Reduce function

La funzione di reduce si occupa di processare i valori intermedi prodotti dalla funzione di map. In particolare viene richiamata una sola volta per ogni chiave univoca generata. Dopo aver elaborato l'input produce nuovo set di dati che sarà il risultato in output. Nell'esempio del conteggio delle parole la funzione di reduce viene richiamata una sola volta per ogni chiave univoca, quindi somma i singoli risultati delle funzioni di map, infine restituisce il risultato finale con il conteggio delle parole.

Capitolo 3

Spark e Spark-SQL

3.1 Architettura

Spark fa uso di un'architettura Master-Slave con un solo coordinatore (il *Driver*) e diversi workers (gli *Executors*). L'insieme costituito dal driver e dai suoi executor altro non è che un'applicazione Spark. Ogni applicazione Spark viene lanciata su un insieme di macchine specifico dal *Cluster manager*.

3.1.1 Resilient Distributed Dataset (RDD)

In Spark è fondamentale il concetto di *Resilient Distributed Dataset* (*RDD*). Un RDD è una collezione di oggetti in sola lettura immutabile e distribuita. Immutabile in quanto una volta creata non può più essere modificata, distribuita perché partizionata su un insieme di macchine. Grazie a quest'ultima caratteristica ed alla ridondanza dei dati, è possibile ricostruire in maniera automatica l'intero RDD nel caso in cui venga persa qualche partizione (*Automatic fault tolerance*). Inoltre essere distribuita le garantisce una notevole scalabilità; è infatti sufficiente aggiungere o rimuovere dei nodi al cluster per poaumentare o ridurre lo spazio di memorizzazione in base alle esigenze. I tipi di operazioni che vengono offerti dagli RDDs sono:

- *Transformations*: sono tutte quelle operazioni che, una volta applicate ad un RDD, ne forniscono un altro (map, reduce, etc). Questo tipo di operazioni viene svolto in memoria su ogni RDD.
- *Actions*: quelle operazioni che calcolano un risultato che viene poi salvato, inviato ad un altro exevutor tramite shuffeling oppure restituito al driver (count, collect, etc).

È importantissimo ricordare che gli RDDs sono *lazy evaluated*, cioè non vengono eseguite le operazioni di trasformazione fino al momento in cui non è necessario ottenere il risultato di un'azione. Questo fornisce la possibilità di eseguire gran

parte delle operazioni in pipeline risparmiando tempo e risorse (è spesso possibile lavorare esclusivamente in memoria centrale, il che garantisce una notevole riduzione del tempo di esecuzione delle operazioni).

3.1.2 Cluster manager

Il cluster manager è un componente fondamentale nella struttura di Spark: è lui che ad esempio si occupa di avviare driver ed executor, specifica su quali macchine vanno eseguiti, gli alloca le risorse. Spark stesso lo ha sviluppato in maniera tale che fosse un componente esterno e collegabile. In questo modo è possibile usare lo *Standalone cluster manager*, di default fornito da Spark, oppure altri sviluppati esternamente (come YARN e Mesos).

3.1.3 Driver

Ogni volta che viene avviata un'applicazione le viene associato un driver, che viene poi chiuso al termine dell'applicazione. Questo driver è il processo principale che si interfaccia con l'utente, viene eseguito in un suo thread autonomo e funge da coordinatore centrale dell'applicazione. I servizi che il driver offre sono la conversione del programma, e la sua suddivisione in task, e lo scheduling dei task sui vari executor a sua disposizione.

- Conversione del programma utente e suddivisione in task: il driver si occupa di convertire il programma utente in un insieme di task (la più piccola unità di lavoro in Spark) e di assegnarli ai vari executor. Quando l'utente esegue un programma, implicitamente il driver crea un grafo aciclico diretto (DAG) di operazioni. Infatti ad alto livello il programma è composto di varie operazioni il cui compito è costruire un RDD partendo da un determinato input (un RDD su disco o il risultato di una precedente operazione). Vengono dunque eseguite queste operazioni sequenzialmente e il programma termina con il completamento di delle azioni. Quello che queste azioni restituiscono è il risultato finale del programma lanciato su quel determinato driver. Durante la fase di conversione del programma, il driver opera già alcune ottimizzazioni, come ad esempio il *pipeling*.

- Scheduling dei task: all'avvio ogni executor si registra al suo driver, in questa maniera il driver ha una visione completa dei suoi executor in ogni momento. Sfruttando questa conoscenza il driver cerca di ottimizzare ulteriormente l'esecuzione assegnando i task secondo il *locality aware scheduling*. Dunque cerca di assegnare i task in maniera tale che più dati

possibile siano memorizzati sullo stesso nodo o in un nodo 'vicino' per ridurre i tempi di caricamento delle partizioni RDD.

3.1.4 Executor

Gli executor sono processi distribuiti che si occupano di eseguire i task. Tipicamente restano in esecuzione fino al termine dell'applicazione Spark (*static allocation of executors*) ma possono anche essere allocati dinamicamente, e quindi essere creati e cancellati in base alle esigenze durante il ciclo di vita dell'applicazione. Spark garantisce un buon livello di fault tollerance: considera gli executor elementi sacrificabili all'interno di un'applicazione. Se infatti un executor dovesse fallire, l'applicazione continuerebbe a funzionare correttamente. Gli executor hanno due compiti principali:

- *Task execution*: ogni executor esegue i task che gli vengono direttamente assegnati dal driver e periodicamente lo aggiorna sullo stato. Infine, una volta terminato il task, invia il risultato al driver. Durante il suo ciclo di vita un executor può eseguire più task, sia in parallelo che sequenzialmente. Poiche gli executor sono configurabili tramite gli appositi parametri, modificare il parametro *executor.core* permette di decidere quanti task ogni executor può al massimo eseguire concorrentemente.

- *RDD storage*: il Block manager fornisce agli executor uno spazio di memorizzazione degli RDD. Se richiesto dall'applicazione utente, ogni executor può usare il suo spazio per memorizzare i dataset in memoria (cache di un RDD). Anche lo spazio di memorizzazione, così come il numero di core, può essere configurato. Il parametro per configurare la quantità di spazio è *executor.memory*.

La memoria fornita ad ogni executor può essere internalmente usata anche per altri due scopi:

- *Shuffle e Aggregation Buffer*: Spark ha uno spazio di memoria per l'allocazione di dei buffer. Questi buffer li crea in due casi: per memorizzare i dati prima di un'operazione di shuffle, oppure per memorizzare i risultati intermedi durante l'operazione di aggregazione.

- *User Code*: dato che l'applicazione utente potrebbe aver bisogno di uno spazio di memoria heap (possono essere allocati array o altre risorse), gli viene lasciato tutto lo spazio non occupato dai Buffer e dalla memorizzazione degli RDD.

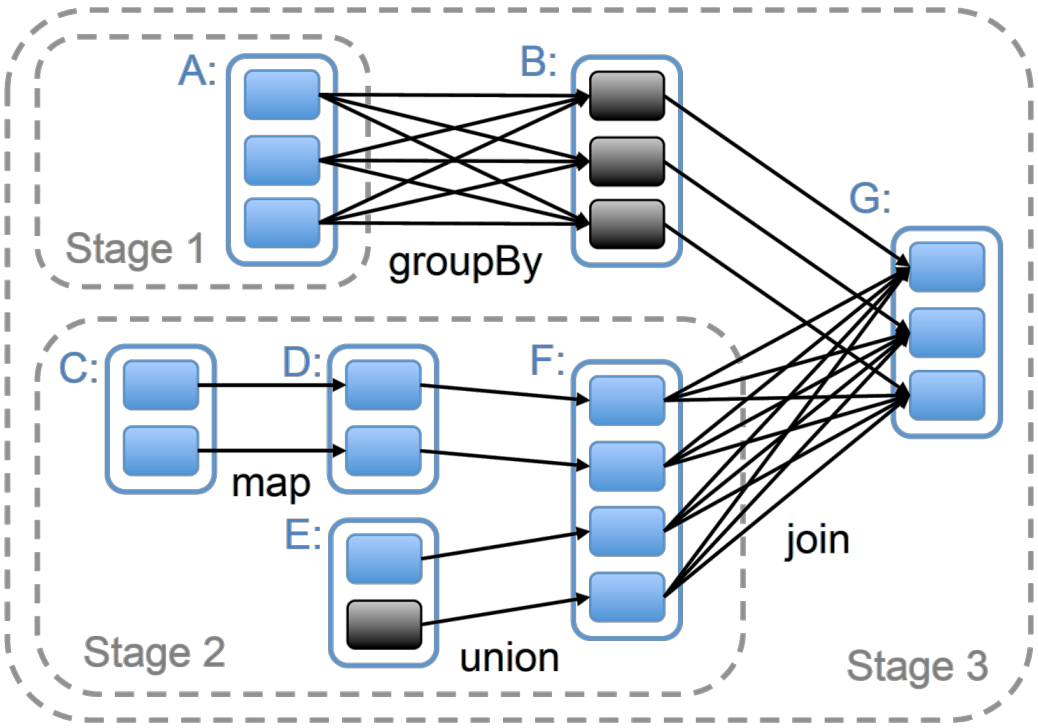


FIGURA 3.1: Esempio di suddivisione di un job in stage ed organizzazione degli stessi.

Naturalmente anche la percentuale di memoria allocata per ogni scopo (RDD storage, Buffer, User code) è modificabile tramite appositi parametri. Di default Spark lascia il 60% dello spazio per la memorizzazione degli RDD, il 20% per i buffer, il restante 20% per il codice dell'utente.

3.2 Architettura ad alto livello

Le operazioni che possono essere eseguite su ogni RDD sono di due tipi:

- *Action*: restituiscono un risultato al driver o effettuano uno shuffeling tra i dati distribuiti in diverse partizioni.
- *Transformation*: consistono nella trasformazione di un RDD in un altro RDD. Queste operazioni possono essere eseguite in parallelo, e ciò permette di mantenere i dati in memoria senza dover accedere al disco.

3.2.1 Job

Al livello più alto di astrazione, una applicazione Spark è suddivisa in *Job*. Il Job è strettamente associato ad una *action*, infatti invocare una action corrisponde

ad innescare un Job. Così come per le action, ogni applicazione può essere composta di diversi Job (per esempio se l'applicazione funge da interfaccia per delle richieste in rete, quindi ad ogni richiesta viene assegnato un Job). I Job possono essere eseguiti in maniera concorrente solo nel caso in cui vengano lanciati in thread differenti. Quindi l'esecuzione di una query all'interno di un'applicazione può essere vista come il lancio di un Job. Un caso particolare è quello dell'operazione di Broadcast Hash Join, che necessita di una fase di collect dei dati sul driver, quindi crea in maniera implicita un nuovo Job. Lo scheduler spark di default esegue i Job con una politica FIFO. Il Job in esecuzione dunque prende la priorità su tutte le risorse utilizzabili dai suoi stage. Questo implica che se il primo Job necessitasse di tutte le risorse e fosse piuttosto pesante, i Job successivi potrebbero subire significativi ritardi. I Job successivi possono partire non appena il Job precedente rilascia la priorità sulle risorse necessarie al successivo(non ha bisogno di usare tutto il cluster).

3.2.2 Stage

Gli *Stage* sono l'unità logica che compone un Job. Gli stage sono collegati tra di loro da un grafo aciclico diretto. Ogni stage è composto da un insieme di *Task* la cui caratteristica è quella di eseguire la stessa operazione sui vari subset di dati. Uno Stage è dunque composto da un insieme di *transformation* eseguite senza che sia necessario uno shuffeling dei dati; la necessità di uno shuffeling causa infatti la creazione di un nuovo stage. Quando lo scheduler analizza il DAG lo divide in stage ogni volta che è necessario uno shuffeling dei dati; esegue poi i diversi stage in ordine topologico. Gli stage possono essere di due tipologie:

- *Shuffle Map Stage*: in questo tipo di stage vengono solo eseguite delle trasformazioni ed i risultati ottenuti fungono da input per altri stage.
- *Result Stage*: in questo caso i dati in input vengono analizzati ed elaborati per poter restituire un risultato, tale stage si conclude con l'action che ha avviato il Job.

È piuttosto importante ricordare che i Job vengono eseguiti sequenzialmente ed hanno una priorità. Per evitare che si stage più vecchi si blocchino in attesa del risultato di altri determinati Stage, viene memorizzato l'ID del primo Job che li ha innescati. In questa maniera verranno sempre eseguiti e recuperati (nel caso di failure) per primi gli stage dei Job eseguiti per primi.

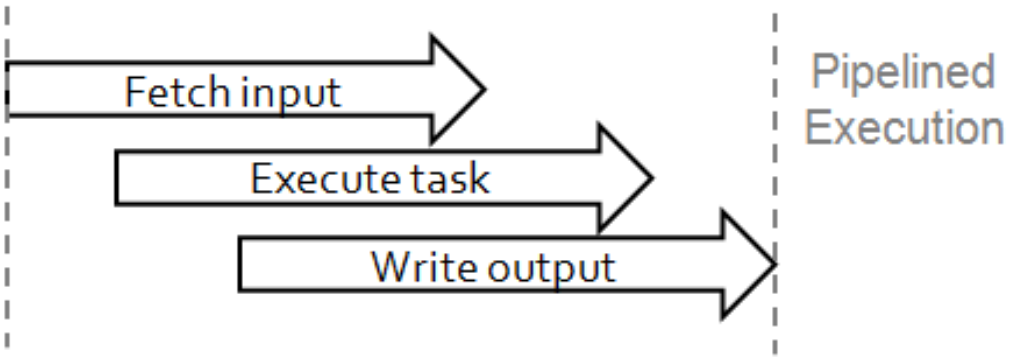


FIGURA 3.2: Principali fasi di un task.

3.2.3 Task

I *Task*, come già detto, sono l'unità di elaborazione più piccola all'interno di Spark. Vengono eseguiti all'interno degli executor e la loro funzione viene dettata dal driver. Anche i Task, come gli Stage, sono di due tipologie:

- *Shuffle Map Task*: l'operazione eseguita da questo tipo di task è la trasformazione, quindi elaborano degli input, li dividono in output ed inviano le varie parti ai vari executor che le richiedono (gli step successivi).
- *Result Task*: questi task corrispondono alle azioni, quindi elaborano degli input per fornire un output da inviare direttamente al driver.

Ogni Task è costituito da tre fasi principali (Figura 3.2):

- *Fetch input*: in questa fase vengono caricati dei dati da file o cache;
- *Execution*: viene poi eseguita la funzione su questi dati;
- *Write output*: infine il risultato viene inviato come dato di shuffle per i task seguenti oppure come risultato per il driver.

Ogni task si occupa di effettuare la sua operazione su una determinata partizione RDD. Uno stage è quindi composto da un discreto quantitativo di task che eseguono la stessa operazione sulle varie partizioni RDD. Questi task vengono eseguiti in parallelo ciascuno sul rispettivo executor.

3.3 Catalyst

Dato che le query SQL sono piuttosto elastiche sul modo in cui vengono eseguite (quello che conta è il risultato, non il percorso), quando si è studiato il modulo

per la traduzione delle query SQL si è pensato di sfruttarlo per ottimizzarle. Proprio per quest'ultima fase è nato all'interno di Spark un nuovo ottimizzatore apposito: *Catalyst*.

Catalyst è stato realizzato fin dal principio in maniera estendibile per due ragioni:

- Prima di tutto per permettere l'aggiunta di nuove tecniche di ottimizzazione e funzionalità a Spark. Si sapeva già agli inizi quanti problemi ed eccezzioni potessero generare i Big Data. È stata dunque implementato questo modulo in maniera tale da semplificare successive implementazioni delle nuove soluzioni a tali problemi.

- In secondo luogo per permettere e sviluppatori esterni di estendere l'ottimizzatore. Per esempio aggiungendo il supporto a nuovi tipi di dati o aggiungendo nuove regole specifiche per determinati data source che permettano il push down di operazioni di filtraggio o aggregazione in sistemi di storage esterni.

Il cuore di Catalyst contiene una libreria generica per rappresentare gli alberi ed applicare delle regole per manipolarli. Il principale tipo di dati in Catalyst sono per l'appunto gli alberi. Ogni albero è composto da un insieme di nodi oggetto. Ogni nodo è definito da un tipo ed ha 0 o più figli. L'albero è immutabile, e per questa ragione ogni trasformazione che gli viene applicata consiste nella creazione di un nuovo albero. Le modifiche che si vogliono apportare agli alberi le si applicano tramite l'uso delle *rules*: funzioni che, preso un albero, ne restituiscono un altro. Nonostante queste funzioni possano applicare un qualunque codice sul loro albero in input, l'approccio più comune è quello di usare un insieme di funzioni di pattern matching che trovino dei sotto-alberi e li sostituiscano con una specifica struttura.

Su questo framework è stato poi costruito un insieme di librerie specifiche, per processare le query relazionali, ed alcuni insiemi di regole che gestiscono diverse fasi dell'esecuzione delle query: analisi, ottimizzazione logica, pianificazione fisica e generazione del codice (il Java bytecode).

3.4 Catalyst in Spark SQL

Il framework per la trasformazione generica degli alberi di Catalyst viene usato in quattro fasi all'interno di Spark SQL:

- *Analysis*: sia che Spark SQL cominci con un AST (*Abstract Syntax Tree*) o da un oggetto Data Frame, la relazione potrebbe contenere riferimenti ad

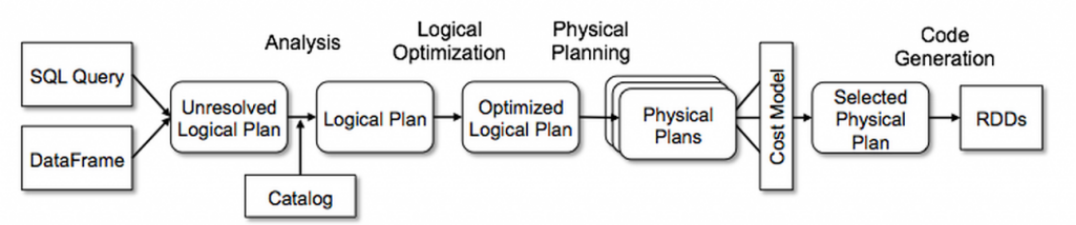


FIGURA 3.3

attributi non risolti. Per risolverli e produrre il piano logico, Spark SQL usa Catalyst. Per esempio il tipo della colonna ed anche se effettivamente è una colonna valida o meno, sono dati che vengono ottenuti usando le regole di Catalyst.

- *Logical Optimizations*: in questa fase vengono applicate le ottimizzazioni standard rule-based al piano logico. Queste includono il push-down dei predicati, la semplificazione delle espressioni booleane ed altre regole.
- *Physical Planning*: In questa fase Spark SQL prende il piano logico e produce uno o più piani fisici. Poi usa un modello di costo per selezionare il piano fisico migliore. Vengono anche applicate alcune ottimizzazioni rule-based in questa fase quali il pipelining delle proiezioni e dei filtri.
- *Code Generation*: In questa fase l'ottimizzazione viene solo effettuata a livello di generazione del codice. Il risultato di questa operazione sarà direttamente il JVM bytecode.

3.5 Join in Spark

Alla base di Spark abbiamo visto esserci gli RDD. Data la dimensione elevata che possono avere, vengono partizionati su diversi nodi in maniera da suddividere il carico e sfruttare la memoria di più macchine. Questa tecnica causa uno dei principali colli di bottiglia nell'esecuzione delle query: lo shuffeling dei dati. Lo shuffeling altro non è che l'operazione di trasfermiento dei dati da un nodo all'altro all'interno del cluster. A livello di performance vanno dunque ad influire due nuovi parametri: la quantità di dati da trasferire e la velocità di trasferimento (throughput). Le operazioni più pesanti a livello di shuffle sono i join, che vanno trattati in maniera più approfondita a causa della natura partizionante del cluster. Durante un join tra due tabelle è necessario scorrere tutti

i record di ciascuna delle due. Però gli RDD sono divisi in partizioni e memorizzati in vari nodi. Perciò prima di poter effettuare l'operazione, è necessario inviare tutti i dati al worker in maniera tale che possa eseguire il join in maniera corretta. Spark può eseguire il join in due modi differenti:

- **Shuffled Hash Join**: questo tipo di join riesce ad unire due tabelle indipendente dalla loro dimensione. Alla base vi è una funzione di hash. Tale funzione viene applicata all'attributo su cui le due tabelle fanno join ed indica per ciascun record a quale executor deve essere inviato. Quindi ogni executor carica prima in memoria i dati presenti su disco, vi applica la funzione di hash e poi effettua uno shuffeling di tali dati con gli altri nodi per inviare i dati agli altri executor. Completata questa fase, ciascun executor è in possesso dei record delle due tabelle con le stesse chiavi di join. In conclusione le varie partizioni vengono unite per completare la tabella risultante dal join. È importante sapere che l'operazione di shuffeling viene effettuata in contemporanea da tutti i nodi, quindi le performance di trasmissione dipendono anche dal numero di executor che condividono lo stesso nodo di rete.

- **Broadcast Hash Join**: quest'altro tipo di join può essere applicato quando una delle due tabelle è notevolmente piccola, tanto da poter essere memorizzata nella sua interezza in memoria centrale da ogni executor. Infatti la prima fase prevede che ogni executor contenente una partizione di tale tabella la invii al driver. Completata questa operazione, il driver invia tale tabella in broadcast a tutti gli executor incaricati di eseguire il join. Ciascun executor può caricare i dati della tabella di dimensioni maggiori e mano a mano confrontarli con la tabella in memoria con la certezzadi trovare la corrispondenza, se presente. Anche al termine di questa operazione vengono uniti i risultati parziali generati dai singoli executor per costituire la tabella risultato. In questo caso il traffico sulla rete è notevolmente ridotto, non vi è la necessità di inviare la tabella di dimensioni più elevate, ma vi è il limite sulla dimensione della tabella in memoria centrale.

È quindi importante operare una corretta scelta del tipo di join da utilizzare. Nel caso in cui entrambe le tabelle siano di dimensioni elevate è obbligatorio utilizzare uno Shuffled Hash Join, che suddivide le tabelle e distribuisce il carico di lavoro. Invece nel caso in cui una delle due tabelle sia sufficientemente piccola da essere caricata in memoria centrale, bisogna valutare la situazione: se la tabella è decisamente piccola, allora la quantità di dati inviati in rete è

notevolmente inferiore a quelli inviati in uno Shuffle Hash Join; se la differenza in dimensione non è eccessiva, il Broadcast Hash Join potrebbe risultare più lento di uno Shuffled Hash Join. Infatti gli executor non possono iniziare il join fino a quando non hanno tutta la tabella. Perciò il driver rischia di diventare un collo di bottiglia nella fase iniziale: quando deve raccogliere tutte le partizioni della tabella per poi inviarle agli executor.

Per definire se la tabella è troppo "piccola" Spark usa un parametro, di default settato a 10MB. Nel caso in cui l'utente voglia, può modificare anche questo valore secondo le proprie preferenze. Nel caso in cui la tabella sia inferiore a tale dimensione viene usato un Broadcast Hash Join, altrimenti uno Shuffled Hash Join.

Capitolo 4

Un modello di costo per Spar-SQL

In questo capitolo andiamo a parlare nello specifico del nostro modello di costo. Il modello di costo in questione è in grado di calcolare il tempo di esecuzione dei piani fisici di Spark (Figura 4.1). Il modello è in grado di manipolare un'ampia varietà di classi di query: le query di tipo GPSJ (*Generalized Projection / Selection / Join*).

Ogni piano fisico di Spark che rappresenti una query GPSJ può essere rappresentato in un albero nel quale i nodi applicano le operazioni ad una o più tabelle, che siano fisiche o il risultato delle operazioni eseguite nel sotto-albero. La grammatica adottata per rappresentare tali alberi è quella visibile in Figura 4.2. I tipi di operazione che possono comporre un possibile albero sono i seguenti:

- **SC()** Table Scan
- **SB()** Table Scan and Broadcast
- **SJ()** Shuffle Join
- **BJ()** Broadcast Join
- **GB()** Group By

SC() e SB() sono sempre le foglie dell'albero, in quanto si occupano di leggere le tabelle direttamente dallo storage fisico. SJ() e BJ() compongono invece i nodi dell'albero e possono creare un *left-depth tree*. Questo perché, per ragioni di ottimizzazione e pipelining, non si effettua mai un Join tra altri due Join, bensì uno dei due figli deve essere una operazione di scansione di una tabella fisica. Infine GB() rappresenta l'ultimo task da eseguire, la radice dell'albero. Per ragioni di chiarezza nella Figura 4.2 sono stati omessi alcuni parametri che caratterizzano i tipi di task, questi stessi parametri sono rappresentati nella Tabella 4.1:

- *pred*: è un parametro opzionale che specifica un predicato di filtro o join.

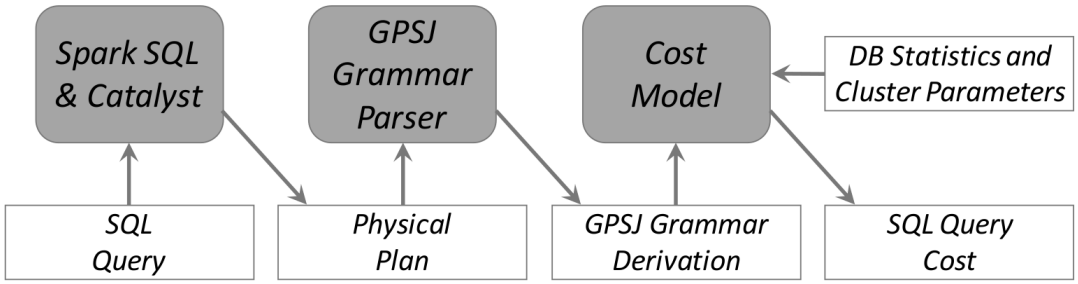


FIGURA 4.1: Una rappresentazione del processo per il nostro modello di costo. I blocchi in grigio sono moduli di sistema, mentre quelli in bianco sono i dati in input/output.

```
<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 4.2: Rappresentazione di Backus-Naur della grammatica GPSJ.

- *cols*: rappresenta il sottoinsieme di colonne che vanno restituite.
- *groups*: identifica un insieme di operazioni di proiezione.
- *pipe*: è un booleano che rappresenta se il task type deve effettuare una scrittura o se lavora in pipelining con il task successivo. La possibilità di pipe vi è solo nei task seguiti da un broadcast join in quanto gli altri nodi sono foglie(SC() e SB()) oppure richiedono lo shuffle(SJ() e GB()).

Oltre alla GB(), anche le operazioni di SC() e SB() potrebbero avere il predicato di raggruppamento *groups* in quanto, come già visto, Catalyst effettua il push-down delle proiezioni.

4.1 Struttura delle Query GPSJ

Il nostro modello copre un gran numero di query che sono composte da tre operazioni SQL base: selezione, join e proiezione generale. La combinazione di queste tre operazioni compone una query GPSJ (*Generalized Projection / Selection / Join*), che abbiamo studiato nel [4]. Questa classe di query è la più utilizzata in ambito OLAP (*On-Line Analytical Processing*). Una query di tipo

TABELLA 4.1: Struttura e caratteristiche dei *TaskType*.

Task Type	Addittional 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

GPSJ è una proiezione generale π su una selezione σ su di uno o più join χ. Può quindi essere espressa in algebra relazionale nel seguente modo:

$$q = \pi_{P,M}\sigma_{Pred}\chi$$

dove χ rappresenta un numero indefinito di join:

$$\chi = T_1 \bowtie ... \bowtie T_n$$

Una proiezione generale $\pi_{P,M}(R)$ è un'estensione della proiezione di eliminazione dei duplicati. In tale proiezione P costituisce il pattern di aggregazione su una relazione R; M denota un insieme di operatori di aggregazione applicati all'attributo R. A livello di SQL P rappresenta gli attributi sui quali si fa *group by*, e P e M insieme costituiscono le clausole della selezione. Naturalmente non tutte le query GPSJ presentano tutti e tre gli operatori, il nostro modello copre sia le query GPSJ complete, sia semplici query di selezione o selezione e join.

4.2 Cluster abstraction e parametri del modello di costo

In questa sezione ci occupiamo di definire i parametri necessari per il calcolo del modello di costo. I parametri sono di due tipi: parametri del cluster e parametri di Spark. I primi riguardano la struttura fisica del cluster sul quale stiamo lavorando, i secondi vengono invece scelti a run-time nei config di Spark. Innanzi tutto definiamo la struttura di un cluster. Come si può osservare nella Figura 4.3, un cluster è composto da $\#N$ nodi pari distribuiti su $\#R$ Rack. Ogni nodo a suo volta monta $\#C$ cores. Per semplicità nel nostro modello abbiamo assunto che tutti i rack ed i nodi siano uguali a livello di funzionalità hardware e di configurazione. All'interno del cluster i dati sono memorizzati sul file system HDFS e sono suddivisi in blocchi. Ogni blocco ha un *Redundancy Factor* rf (fattore di ridondanza), che di default in Spark è $rf = 3$. Questo

valore consiste nel numero di copie che vengono tenute per salvaguardarsi dai fallimenti o dai guasti. La scelta di tale valore è molto importante perché se scelto troppo basso si ha un alto rischio di perdita dei dati, se scelto troppo altro influisce sulle performance inutilmente. Un altro elemento che influisce notevolmente sul costo di una funzione Spark, e di cui è fondamentale tenere conto per effettuare un calcolo dei costi preciso, è il throughput generato dal disco. Per modellare tale throughput noi usiamo due funzioni: $\delta_r(\#Proc)$, per il throughput in lettura, e $\delta_w(\#Proc)$, per quello in scrittura. Ciascuna delle due funzioni restituisce il throughput, in MB/s, che gli viene richiesto per ogni processo. Tali valori si possono ottenere eseguendo specifici test sul cluster in questione.

Un aspetto fondamentale riguardante i cluster è che sono composti da una rete di macchine. Quindi è fondamentale tenere conto anche dell'influsso che ha la rete sui costi di calcolo di una funzione. Nel nostro modello abbiamo presupposto un tipo di rete point-to-point con un limite di banda per ogni connessione. Analogamente al comportamento del disco, anche il throughput di rete dipende dal numero di processi che stanno trasmettendo simultaneamente tra coppie di nodi. Inoltre le velocità di comunicazione in rete sono differenti rispetto alla posizione dei nodi che stanno comunicando. I tipi di comunicazione possono essere intra-rack($IntraRSpeed$) ed inter-rack($ExtraRSpeed$). Nel nostro modello abbiamo assunto che la velocità di comunicazione all'interno del rack sia notevolmente superiore rispetto a quella di comunicazione con nodi in altri rack, come è solitamente. Nel modello abbiamo tenuto conto anche della riduzione del throughput dovuta alla compressione dei dati ($nCmp$). Tale compressione viene attuata automaticamente da Spark ed il suo valore percentuale può essere modificato in Spark dalle opzioni di configurazione. In conclusione le formule per il calcolo del throughput di rete in base al numero di processi sono:

$$\rho_i(\#Proc) = \frac{IntraRSpeed}{\#Proc \cdot nCmp} \tag{4.1}$$

$$\rho_e(\#Proc) = \frac{ExtraRSpeed}{\#Proc \cdot nCmp} \tag{4.2}$$

Come accennato precedentemente, ogni applicazione Spark in esecuzione sul cluster ha il suo insieme di parametri e risorse personali. È importantissimo specificare che nel nostro modello di costo abbiamo assunto che le risorse, una volta assegnate, non possono più essere modificate durante l'esecuzione. Le due risorse principali da definire sono il numero di executor ($\#E$) ed il numero di cores in ogni executor ($\#EC$).

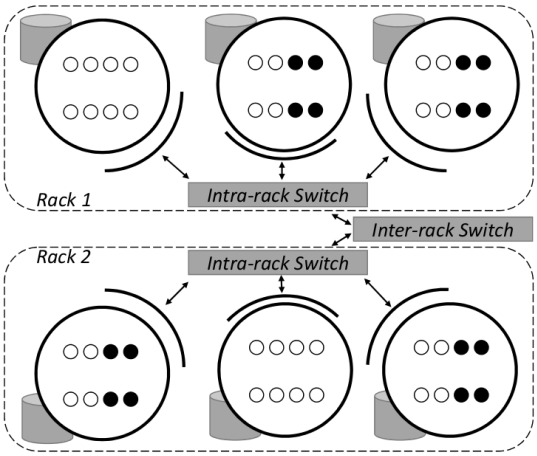


FIGURA 4.3

Come visto nella Sezione 3.1, Spark riserva un numero ben definito di bucket per effettuare le operazioni di shuffle. Tale numero influenza notevolmente il tempo di esecuzione delle operazioni di Shuffle, quindi nel nostro modello teniamo conto anche di tale valore ($\#SB$) che di default è pari a 200. Un'altra assunzione che abbiamo fatto durante lo sviluppo del modello è che ogni bucket di shuffle coincida esattamente con la memoria dell'executor. In questa maniera i dati non devono mai passare per il disco locale, che rallenterebbe notevolmente l'esecuzione della funzione.

Dato che il numero di partizioni RDD è solitamente maggiore del numero di cores disponibili per il calcolo, i task vengono suddivisi nei cores a disposizione in diverse *waves* dal gestore delle risorse. Ogni *wave* è caratterizzata dalla possibilità di eseguire in parallelo un insieme di task dello stesso tipo, uno in ogni core disponibile nei vari executors dell'applicazione. Nel nostro modello abbiamo considerato che tutte le esecuzioni all'interno della stessa wave si comportano più o meno alla stessa maniera. In questo modo il tempo complessivo necessario per l'esecuzione dell'applicazione Spark, è equivalente al tempo necessario per l'esecuzione di un singolo task su un singolo core.

4.3 Basic Brick

Non è nostro interesse modellare Azioni e Trasformazioni in questo modello di costo. Fare ciò comporterebbe un inutile complessità per il modello, che invece ha il solo interesse ad analizzare i costi di accesso al disco e di rete. Per questa ragione il più piccolo elemento per il calcolo del costo nel nostro modello è, per l'appunto, il *Basic Brick*. Questi bricks vengono usati all'interno dei vari task

TABELLA 4.2: Cost model parameters and basic functions. Horizontal lines split those related to cluster, application and data respectively.

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 .

type per calcolare il costo di escuzione delle query GPSJ. Il livello di astrazione è addirittura più basso rispetto a Azioni e Trasformazioni: vengono modellate le operazioni che vengono eseguite su ogni partizione RDD (i.e. Read, Write, ...). Ai bricks non interessa il linguaggio che viene usato per lanciare tali operazioni (che sia SQL o altro). Questi hanno bisogno di conoscere solo i parametri di Spark e di Cluster (Tabella 4.2) per riuscire a svolgere i loro calcoli dei costi. Nei successivi paragrafi analizzeremo la struttura di ogni Basic Brick e la loro funzione di costo.

4.3.1 Read

Il tempo di lettura in un cluster varia in base a dove si trova la partizione RDD da leggere. Nel caso di un cluster Spark il *data locality principle* ci assicura che ogni nodo cercherà sempre di leggere i dati dalla partizione più "vicina". Il tempo di lettura dunque varia se l'executor legge la partizione dal disco locale, da un nodo all'interno dello stesso rack o da un nodo esterno al rack. $Read(Size, X)$ calcola il tempo di lettura di una partizione RDD di una determinata dimensione in base alla sua posizione. La sua posizione è definita dal parametro $X \in \{L, R, C\}$, dove L sta per Local, R sta per Rack e C sta Cluster.

Dunque se i dati sono letti da disco non vi è alcuna trasmissione di dati in rete, ma se i dati vengono letti da un altro nodo allora è necessario considerare anche il tempo necessario per la trasmissione di tali dati. Dato però che la lettura su disco e la trasmissione in rete di tali dati all'executor che li ha richiesti avviene in pipeline, il tempo complessivo impiegato nella lettura di una partizione è il massimo tra le due componenti di lettura e trasmissione in rete:

$$Read(Size, X) = \text{MAX}(ReadT_X; TransT_X)$$

Vediamo ora di definire i tempi di $ReadT_X$ e $TransT_X$ per ogni caso (L, R, C).

Se la lettura avviene da una partizione in locale, allora non è necessaria la trasmissione in rete, quindi $TransT_L = 0$.

$$ReadT_L = \frac{Size}{\delta_r(\#EC)}$$

$ReadT_L$ è il tempo necessario ad un core per leggere una Partizione RDD di dimensione $Size$ da un disco locale il cui throughput per-process è $\delta_r(\#EC)$.

Quando si ha una rack wave, ogni core di nodi executor riceve i dati dai nodi non executor presenti all'interno dello stesso rack ($\#RN - \#RE$ nodi). Durante questa fase non viene mai acceduto il disco locale in quanto, se vi si leggessero dei dati, sarebbe una wave locale e non di rack. Dato che il numero di processi che possono eseguire su core in parallelo è $\#RE \cdot \#EC$ e supponendo che una partizione viene letta per intero da un nodo, possiamo dire che ogni nodo non executor serve in media $\lceil \frac{\#RE \cdot \#EC}{(\#RN - \#RE)} \rceil$ richieste. Quindi il tempo di lettura è:

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

Inoltre, al contrario delle letture locali, i dati vengono poi trasmessi in rete. Quindi ognuno dei core di un executor node ($\#EC$) riceve una partizione RDD da uno dei nodi non-executor ($\#RN - \#RE$) Supponendo dunque una equi distribuzione delle partizioni, possiamo definire il numero di core che condividono la stessa rete da nodo a nodo come $\lceil \frac{\#EC}{\#RN - \#RE} \rceil$. Quindi il tempo necessario a trasmettere i dati in ogni connessione di rete è:

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

Le wave di tipo cluster so possono modellare in maniera analoga a quelle di tipo rack, con la differenza che ogni nodo non executor può ricevere una richiesta da ciascuno dei core degli executor che risiedono in altri rack $((\#R - 1) \cdot \#RE \cdot$

$\#EC$). Queste richieste sono però distribuite sui $(\#R-1)\cdot(\#RN-\#RE)$ nodi non-executor che si trovano in altri rack. Il throughput generato dal disco su ogni nodo non-executor è quindi $\delta_r(\lceil\frac{(\#R-1)\cdot\#RE\cdot\#EC}{(\#R-1)\cdot(\#RN-\#RE)}\rceil)$, perciò il tempo di accesso al disco è:

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

Anche in questo caso dobbiamo tenere conto dei costi di trasmissione delle partizioni RDD in rete. Durante una cluster wave abbiamo detto che ognuno dei $\#EC$ cluster richiede una partizione ad uno dei $(\#R-1)\cdot(\#RN-\#RE)$ nodi non-executor. Come nel caso dei rack alcuni di questi core appartengono allo stesso nodo, possiamo quindi definire tale valore come $\lceil\frac{\#EC}{(\#R-1)\cdot(\#RN-\#RE)}\rceil$. Analogamente al caso precedente, il tempo necessario per trasmettere i dati attraverso ognuna delle connessioni di rete è:

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

4.3.2 Write

Come già discusso nella Sezione 3.1, una partizione RDD non può essere modificata. Questo implica che, una volta letta e elaborata in memoria centrale, viene riscritta la nuova partizione su disco locale. La funzione $Write(Size)$ ha il compito di calcolare quanto tempo è necessario ad ogni executor per scrivere sul disco $Size$ MBs di dati. Naturalmente il throughput di disco viene influenzato dal numero di executor in ogni core, perciò:

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

4.3.3 Shuffle Read

Quando Spark effettua un'operazione di shuffle read quello che deve fare è leggere i $\#SB$ shuffle bucket che sono stati precedentemente scritti nella fase di shuffle write. Per ottimizzare questa fase Spark crea tanti task quanti sono i bucket ($\#SB$) e distribuisce i bucket sui vari executor, cioè ogni executor memorizza una porzione di ogni bucket. La funzione $SRead(Size)$ modella il tempo necessario per leggere ogni singolo bucket di dimensione $Size$ MBs. La lettura dei dati presenti all'interno del bucket viene effettuata in pipeline. Quindi, secondo il Volcano model, il tempo di caricamento dei dati può essere calcolato

come il massimo dei tempi necessari per portare a termine le due operazioni.

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

È importante sottolineare che le operazioni di shuffle join riguardano esclusivamente i nodi executor. Abbiamo inoltre presupposto che ogni executor si comporti allo stesso modo e che non sia possibile applicare il principio della *datalocality*. Quest'ultimo non è possibile per il fatto che, in questo caso, ogni singolo bucket è distribuito su tutti gli executor e non vi è alcuna replicazione dei dati. Dato che il bucket è diviso in maniera eguale su tutti gli executor, possiamo definire la dimensione della porzione di bucket memorizzata in ogni executor come $ExecBucketSize = Size/\#E$ MBs. Durante l'esecuzione di questa operazione ogni core richiede in parallelo a tutti gli executor la porzione di bucket in loro possesso. Quindi ogni executor deve soddisfare $\#E \cdot \#EC$ richieste ($\#EC$ dai core locali, $(\#E-1)\cdot\#EC$ da quelli remoti). Dato il parallelismo delle richieste, il tempo di lettura complessivo è:

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

Per quanto riguarda il tempo di condivisione dei dati in rete, ricordiamo nuovamente che i vari executor sono simmetrici e che il numero totale di processi che condividono la stessa connessione da nodo a nodo sono $\#EC$. Perciò il numero di processi in invio ed in ricezione in ogni connessione è lo stesso, dunque anche il relativo tempo di trasmissione. Dato però che le velocità di trasmissione dei dati all'interno dello stesso rack è solitamente differente da quella di trasmissione tra rack differenti, è importante capire quando lo shuffle avviene tra nodi interni al rack e quando invece appartengono a rack differenti. La probabilità che v executors siano allocati all'interno dello stesso rack è definita dalla formula:

$$P_{SR}(v) = \begin{cases} \frac{\binom{\#RN}{v}}{\binom{\#N}{v}} \cdot \#R, & \text{if } v \leq \#RN \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

la probabilità che gli executor siano allocati tutti all'interno dello stesso rack, nel caso in cui il numero di executor (v) sia maggiore del numero di nodi in ogni rack ($\#RN$), è pari a 0. Altrimenti la frazione ci definisce la probabilità. Detto ciò, possiamo definire il tempo di trasmissione come segue:

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

4.3.4 Broadcast

L'operazione di broadcast è principalmente composta da due fasi: una prima fase prevede il caricamento di tutte le partizioni RDD (ognuna di dimensione $Size$) all'interno dell'application driver; la seconda fase consiste ndella distribuzione dell'intero RDD a tutti gli executors perché possano effettuarci le proprie elaborazioni. Per come è strutturato Spark, l'intero RDD deve essere caricato nell'application driver prima che quest'ultimo re-invii l'RDD ai vari executors. Per questo motivo il tempo totale impiegato per completare un'operazione di broadcast consiste nella somma dei tempi necessari al completamento di questi due passaggi, senza possibilità di parallelizzarli.

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

Come nei casi precedenti, sono lette $\#E \cdot \#EC$ partizioni in parallelo ed ogni connessione di rete da nodo a nodo è condivisa da $\#EC$ processi. Analogamente allo shuffle read, gli executor si potrebbero trovare all'interno dello stesso rack o in rack differenti. La probabilità che gli executor si trovino all'interno dello stesso rack è sempre $Psr(v)$. Dato che in questo caso oltre agli executor dobbiamo considerare anche l'application driver, allora $v = \#E + 1$. Quindi:

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

Per quanto invece riguarda il secondo passaggio, per comodità del modello di costo, abbiamo considerato il costo della distribuzione dell'intero RDD a tranche di $\#E \cdot \#EC$ partizioni. Inoltre, dato che l'application driver invia i dati ad ogni nodo, per ogni connessione di rete vi è un solo processo attivo. In conclusione:

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

Come in casi precedenti, $DistributeT$ non rappresenta il tempo necessario per distribuire l'intero RDD, bensì quello necessario a distribuire una sola wave. L'intero RDD viene infatti distribuito nel suo complesso in diverse wave, tutte eseguite in maniera parallela.

4.4 Task Type

In questa sezione vediamo come è possibile usare i basic brick, visti nella Sezione 4.3, per calcolare il costo complessivo di esecuzione della query. Il tempo di

esecuzione consiste nella somma dei tempi necessari per eseguire ai singoli nodi del piano di esecuzione (Sezione 3.2). Ad ognuno di questi nodi corrisponde un task type. Noi abbiamo associato ad ciascuno dei task type la funzione che restituisce il costo di esecuzione di tale nodo e la struttura della tabella relazione prodotta in output dal nodo. Navigare l'albero in depth-first ci garantisce che ogni nodo ha sempre le tabelle di input a disposizione per i propri calcoli, in tale maniera è sempre possibile computare sia il tempo di esecuzione che la tabella generata. Una volta che sono state associate tali funzioni ai task type, è sufficiente sommare i tempi di esecuzione di ciascun nodo per ottenere il costo complessivo per l'esecuzione della query.

4.4.1 Statistiche e predicati di selettività

Come per i basic brick, anche in questo caso abbiamo bisogno di una serie di dati fondamentali nel calcolo di tali funzioni. Innanzi tutto abbiamo bisogno di memorizzare delle statistiche riguardanti il database, necessari per stimare i predicati di selettività e la dimensione delle tabelle. In questo modello di costo abbiamo assunto uniformità ed indipendenza dei valori degli attributi e il join containment [3].

Vediamo ora quali sono i dati che memorizziamo riguardanti la struttura del database. Una tabella t è composta da un insieme di attributi $t.Attr$. Di ogni tabella memorizziamo la sua cardinalità $t.Card$, la sua dimensione $t.Size$ e se è salvata in un formato di file compresso o meno. Per gestire tale compressione abbiamo memorizzato il fattore di compressione medio $fCmp$. Teniamo anche memoria della dimensione media della partizione HDFS che memorizza la tabella $t.PSize$ Nonostante la dimensione di una partizione HDFS sia un parametro (tipicamente impostato a 128 MB), la sua dimensione effettiva può variare notevolmente; può essere notevolmente più piccola se la tabella è a sua volta piuttosto piccola o se è stata creata senza essere compressa tramite un comando Spark.

Per ogni attributo $a \in t.Attr$ memorizziamo il numero di valori distinti $a.Card$ e la sua lunghezza media in bytes $a.Len$. Basandosi sui dati precedenti e considerando il ben conosciuti lavori di [9] e [8], è possibile stimare il predicato di selettività congiunta $Sel(t, pred)$, la cardinalità $JCard(t_1, t_2, pred)$ e la dimensione $JSize(t_1, t_2, pred)$ di un equi-join $t_1 \bowtie_{pred} t_2$. Dato che un predicato $pred$ può contenere più condizioni, la selettività di tale predicato sulla tabella t è il prodotto delle selettività delle singole condizioni. Per ogni condizione, posto val il valore con cui va confrontato l'attributo, op l'operatore che viene applicato e $a.highest$ e $a.lowest$ i valori più grandi e piccoli presenti nell'attributo,

la formula per calcolare la selettività è la seguente.

$$\begin{cases} \frac{1}{a.Card}, & \text{if } op = \\ \frac{a.highest - val}{a.highest - a.lowest}, & \text{if } op > \text{or } \geq \\ \frac{val - a.lowest}{a.highest - a.lowest}, & \text{if } op < \text{or } \leq \end{cases}$$

Il join tra due tabelle più diffuso è basato su una sola chiave, ma in certi casi potrebbe essere basato anche su chiavi multiple. La cardinalità della tabella risultante nel primo caso può essere stimato in maniera piuttosto semplice, ma nel caso di chiavi multiple è necessario fare un'approssimazione che potrebbe portare a stime non sempre vicine al dato reale. La formula per calcolare la cardinalità del join è:

$$JCard(t_1, t_2, pred) = \frac{t_1.card * t_2.card}{\text{MAX}(leftKeysCard, rightKeysCard)}$$

Nel caso in cui il join sia a chiave singola, $leftKeysCard$ è la cardinalità dell'attributo chiave della prima tabella e $rightKeysCard$ corrisponde a quella dell'attributo chiave della seconda tabella. Per il join a chiave multipla abbiamo adottato la scelta di calcolare i due valori come prodotto delle cardinalità degli attributi chiave rispettivamente della prima e della seconda tabella, quindi:

$$leftKeysCard = \prod_{a \in t_1.keys} a.Card$$

$$rightKeysCard = \prod_{a \in t_2.keys} a.Card$$

Conoscendo la cardinalità, siamo in grado di calcolare anche la dimensione della tabella di join in maniera piuttosto immediata:

$$JSize(t_1, t_2, pred) = JCard(t_1, t_2, pred) \cdot (count(t_1.Attr) + count(t_2.Attr))$$

Siamo anche in grado di stimare la riduzione percentuale della lunghezza determinata da una proiezione $Proj(t, cols)$ sugli attributi $cols \subseteq t.Attr$:

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

Un'altra statistica importante è la riduzione della cardinalità indotta da una proiezione. Tale valore lo si può ottenere sfruttando la formula di Cardenas [2]

nella seguente maniera:

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

4.4.2 Scan SC()

Il compito di un task di Scan è quello di accedere una tabella t memorizzata nel HDFS. La funzione $SC(t, pred, cols, groups, pipe)$ calcola il tempo necessario per portare a termine tale task. Le operazioni di base che vengono svolte da questo task type sono:

(a) **Fetching**: il primo passaggio consta nell'andare a recuperare la partizione RDD che memorizza la tabella t in memoria. Effettuale tale recupero richiede l'accesso al HDFS per recuperare la partizione RDD e mandarne il contenuto agli executor che hanno il compito di processarla. Non vi è necessità di trasmettere i dati in rete a meno che le partizioni non siano memorizzate in locale. Poichè Spark adotta il Volcano model, il tempo complessivo di scansione è il massimo tra il tempo di accesso e trasmissione.

(b) **Filtering (optional)**: in questa seconda fase il task type ha il compito di eseguire le operazioni di filtraggio che gli sono state specificate all'interno del predicato *pred*. Dal momento che Catalyst applica il push down della selezione, questa operazione viene oprata a termine non appena le tuple non sono più necessarie per successive computazioni. Quando è supportato dal formato di file utilizzato (per esempio Parquet [1]), Spark può effettuare il push down del filtraggio addirittura all'interno del file sorgente. In questo caso le tuple non necessarie non vengono nemmeno lette.

(c) **Projecting (optional)**: a questo punto il task può procedere ad una cancellazione delle colonne inutili dalla tabella (le colonne di t che non sono incluse nel parametro *cols*). Dato che Catalyst applica il push down anche della proiezione, tale cancellazione viene effettuata non appena la colonna non è più necessaria per calcoli successivi. Quando supportato dal formato del file su disco (per esempio Parquet), Spark può effettuare tale push down nel file sorgente. In questo caso queste colonne non vengono neppure lette.

(d) **Aggregating (optional)**: quando Catalyst effettua il push down della proiezione, il task effettua una prima aggregazione delle tuple allo scopo

di ridurre la quantità di dati che devono essere gestiti nelle successive operazioni.

(e) **Writing (optional)**: in quest'ultima fase il task scrive il risultato su disco per successive elaborazioni o per salvare il risultato finale. Tale scrittura non viene effettuata nel caso in cui questo task type è seguito da un broadcast join, che viene eseguito in pipeline ($pipe = T$)

Nel nostro modello di costo solo le operazioni (a) e (d) devono essere modellate direttamente, mentre le operazioni (b) e (c) influenzano le performance solo per il fatto che riducono la quantità di dati che vengono scritti su disco.

Vediamo ora il nostro modello. Il numero di partizioni che compongono t è

$$\#TableP = \frac{t.Size \cdot fcmp}{t.PSize} \tag{4.4}$$

se la tabella è memorizzata in un formato compresso $0 < fcmp < 1$, $fcmp = 1$ altrimenti. Ogni partizione RDD considera la quantità di bytes da leggere da disco come

$$RSize = t.PSize \tag{4.5}$$

Ma nel caso in cui venga applicato il push down di filtri e predicati alla selezione, la quantità di bytes da leggere da disco si riduce a

$$RSize = t.PSize \cdot Sel(pred) \cdot Proj(t, cols) \tag{4.6}$$

Il recupero dei dati da disco viene effettuato in parallelo sulle varie partizioni. Il numero di *waves* è

$$\#waves = \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \tag{4.7}$$

Abbiamo distingo 3 tipi distinti di waves in base alla posizione delle partizioni RDD da leggere rispetto all'executor che le deve elaborare:

- **L** - local
- **R** - rack
- **C** - cluster

Data quindi una partizione RDD p, la probabilità che un executor debba recuperare p da disco locale (P_L), all'interno del suo rack (P_R) o altrove (P_C), può essere calcolata con le seguenti formule:

$$P_L = 1 - \frac{\binom{\#N-rf}{\#E}}{\binom{\#N}{\#E}} \tag{4.8}$$

$$P_C = \sum_{x=1}^{min(\#R,rf)} \sum_{y=1}^{min(\#R,\#E)} P_{Part}(x) \cdot P_{Exe}(y) \cdot \frac{\binom{\#R-x}{y}}{\binom{\#R}{y}} \tag{4.9}$$

$$P_R = 1 - P_L - P_C \tag{4.10}$$

Nella Formula 4.8, la frazione definisce il rapporto tra il numero di executor allocati su un nodo del cluster che non ha in memoria una delle rf copie di p, rispetto al numero di executor alloracti sull'intero cluster.

Nella formula 4.9, $x \in \{1, \cdots, min(\#R, rf)\}$è il numero di rack che hanno almeno una delle rf copie di p in uno qualuqnue dei loro nodi, mentre $y \in \{1, \cdots, min(\#R, \#E)\}$ è il numero di racks che hanno almeno un executor allocato. La frazione restituisce il rapporto tra l'allocazione di y racks su $\#R - x$ racks (i rack che non hanno una copia di p nei propri nodi), e il numero totale di allocazioni dei y racks. Tale rapporto va pesato con la probabilità che esattamente x racks ospitino una delle rf copie di p ($P_{Part}(x)$) e la probabilità che esattamente y rack abbiano almeno un executor allocato ($P_{Exe}(y)$):

$$P_{Part}(x) = \binom{\#R}{x} \cdot \sum_{j=0}^{x} (-1)^j \binom{x}{j} \frac{\binom{\#RN \cdot (x-j)}{rf}}{\binom{\#N}{rf}} \tag{4.11}$$

$$P_{Exe}(y) = \binom{\#R}{y} \cdot \sum_{j=0}^{y} (-1)^j \binom{y}{j} \frac{\binom{\#RN(y-j)}{\#E}}{\binom{\#N}{\#E}} \tag{4.12}$$

Nella Formula 4.11, abbiamo usato il principio di inclusione-esclusione per calcolare la probabilità che i x rack specificati, e solo loro, ospitino almeno una delle rf copie di p. Tale probabilità viene dunque moltiplicata per $\binom{\#R}{x}$ in quanto ciascuno dei x racks scelti ha la stessa probabilità. La Formula 4.12 è stata costruita in maniera simmetrica.

Una volta che la partizione RDD è stata letta, vengono portate a termine le operazioni (b), (c) e (d) sulle tuple. A questo punto vengono riscritti su disco $WSize$ bytes:

$$\begin{aligned} WSize =& t.PSize \cdot Sel(pred) \cdot Proj(t, cols) \cdot \\ & Group(t.Card \cdot Sel(pred), \prod_{a \in groups} a.Card) \end{aligned}$$

Se non viene effettuato alcun tipo di raggruppamento, la lettura e la scrittura dei dati possono avvenire in pipeline, altrimenti la scrittura può iniziare solo una volta che tutte le tuple sono state lette e raggruppate. Nel primo caso il tempo di esecuzione di ogni task è stimato come il massimo tra i tempi di

TABELLA 4.3: Stima delle caratteristiche delle tabelle in output t' dei vari task types.

Task type	$t'.Attr$	$t'.Card$	$t'.Size$	$t'.Part$
SC()	$cols$	$t.Card \cdot Sel(t, pred) \cdot Group(t.Card \cdot Sel(t, pred), \prod_{a \in groups} a.Card$	$WSize \cdot \#TableP$	$\#TableP$
SB()	$cols$	$t.Card \cdot Sel(t, pred)$	$WSize \cdot \#TableP$	$\#TableP$
SJ()	$cols$	$JCard(t_1, t_2, pred) \cdot Group(JCard(t_1, t_2, pred), \prod_{a \in groups} a.Card)$	$WSize \cdot \#SB$	$\#SB$
BJ()	$cols$	$JCard(t_1, t_2, pred) \cdot Group(JCard(t_1, t_2, pred), \prod_{a \in groups} a.Card)$	$WSize \cdot t_2.Part$	$t_2.Part$
GB()	$cols$	$t.Card \cdot Group(t.Card, \prod_{a \in groups} a.Card)$	$WSize \cdot \#SB$	$\#SB$

lettura e di scrittura: $SC(t, pred, cols, groups, pipe) =$

$$\begin{cases} \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \cdot \sum_{X \in \{L,R,C\}} P_X \cdot \\ \cdot \text{MAX}(Read(RSize, X), Write(WSize)), & \text{if } pipe \\ \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \cdot \sum_{X \in \{L,R,C\}} P_X \cdot Read(RSize, X), & \text{otherwise} \end{cases} \tag{4.13}$$

Nel secondo caso, i due tempi vanno sommati: $SC(t, pred, cols, groups, pipe) =$

$$\begin{cases} \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \cdot \sum_{X \in \{L,R,C\}} P_X \cdot (Read(RSize, X) + Write(WSize)), & \text{if } pipe \\ \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \cdot \sum_{X \in \{L,R,C\}} P_X \cdot Read(RSize, X), & \text{otherwise} \end{cases} \tag{4.14}$$

In entrambe le Funzioni 4.13 e 4.13, nel caso in cui non vi sia *pipe*, non viene effettuata alcuna scrittura su disco, quindi il relativo tempo di scrittura è pari a 0 e non va considerato.

La Tabella 4.3 mostra le caratteristiche della tabella che viene restituita dal termine di questo task type t'.

4.4.3 Scan & Broadcast SB()

Un'operazione di Scan & Broadcast accede alla tabella t memorizzata nel HDFS e la invia all'Application Driver. Quest'ultimo colleziona tutte le partizioni RDD e invia in broadcast l'intera tabella a tutti gli executors. La funzione $SB(t, pred, cols)$ restituisce il tempo necessario per portare a termine il task.

La raccolta dei dati dal file system è la stessa del task type SC(), facciamo riferimento alle equazioni da 4.4 a 4.10 per modellare la lettura dei dati in questo task type. La dimensione di una partizione RDD da inviare in broadcast può essere ridotta rispetto ai dati letti nel caso in cui vengano applicati predicati di filtro o proiezione:

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

Una nota da fare è che, a prescindere dal fatto che i dati vengano filtrati e proiettati direttamente mentre sono letti da disco o successivamente in memoria centrale, il costo resta lo stesso. Infatti il loro impatto sul tempo di esecuzione del task non è diretto. Dal momento che la lettura dei dati lavora in parallelo con il broadcast di questi ultimi, il tempo di esecuzione totale del task è:

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

La Tabella 4.3 riporta le caratteristiche della tabella t' restituita dal task al termine dell'elaborazione.

4.4.4 Shuffle Join SJ()

Un task di tipo Shuffle Join ha il compito di portare a termine il join tra due tabelle t_1 e t_2 le cui partizioni sono già state precedentemente mappate tramite una funzione di hash in $\#SB$ buckets. Le partizioni RDD in input sono memorizzate nel disco locale dei vari executor prima di procedere con l'esecuzione del task. La funzione $SJ(t_1, t_2, pred, cols, groups, pipe)$ restituisce il tempo necessario per portare a termine questo task. Una wave di un'operazione di shuffle join si compone delle seguenti operazioni principali:

(a) **Shuffle Read**: in questa prima fase vengono caricati i bucket corrispondenti dalle tabelle t_1 e t_2. Una porzione di ogni bucket viene poi memorizzata su ogni executor.

(b) **Join**: una volta che l'executor ha a sua disposizione i relativi bucket da t_1 e t_2, può applicare il predicato *pred* e effettuare l'unione delle tuple.

(c) **Project (optional)**: in questa fase, opzionalmente eseguita solo nel caso in cui sia esplicitamente specificato un predicato di proiezione *cols*, vengono cancellate tutte le colonne non più necessarie per le future elaborazioni o per il risultato.

(d) **Aggregating (optional)**: può essere richiesto da Catalyst un push-down dell'aggregazione per ridurre la quantità di dati da gestire nelle successive elaborazioni. In tale caso è in questo momento che il task esegue una prima aggregazione sulla base del predicato *group*.

(e) **Writing (optional)**: quest'ultima operazione prevede la scrittura su disco delle tuple restituite dalle precedenti operazioni. Come nel caso del

task type Scan, la scrittura è opzionale e non è necessaria nel caso in cui questo task sia seguito in pipeline da quello di broadcast join.

Un join non può mai cominciare prima che tutte le tuple dei relativi bucket siano caricate. Conseguentemente, il tempo necessario per completare una wave è la somma dei tempi necessari per la lettura in shuffle delle tabelle e la scrittura su disco, se necessaria, delle tuple processate. La quantità di dati che è necessario leggere ad ogni wave è

$$RSize = \frac{t_1.Size + t_2.Size}{\#SB}$$

La quantità di dati che vanno riscritti su disco al completamento del join da ogni core ad ogni wave è

$$WSize = \frac{JSize(t_1, t_2, pred) \cdot Proj(cols)}{\#SB} \cdot \\ \cdot Group(JCard(t_1, t_2, pred), \prod_{a \in groups} a.Card)$$

In conclusione, il tempo complessivamente necessario per il completamento di un task di shuffle join è dato dalla seguente formula:

$SJ(t_1, t_2, pred, cols, groups, pipe) =$

$$\begin{cases} \lceil \frac{\#SB}{\#E \cdot \#EC} \rceil \cdot (SRead(RSize) + Write(WSize)), & \text{if } pipe \\ \lceil \frac{\#SB}{\#E \cdot \#EC} \rceil \cdot SRead(RSize), & \text{otherwise} \end{cases} \tag{4.15}$$

Nella funzione 4.15, nel caso in cui non vi sia *pipe*, non viene effettuata alcuna scrittura su disco, quindi il relativo tempo di scrittura è pari a 0 e non va considerato.

La Tabella 4.3 mostra le caratteristiche della tabella che viene restituita dal termine di questo task type t'.

4.4.5 Broadcast Join BJ()

Un task di broadcast join si occupa di portare a termine il join tra ude tabelle t_1 e t_2 nel caso in cui una delle due tabelle è sufficientemente piccola da poter essere inviata in broadcast e tenuta per intero nella memoria dei vari executor. Nel modello di costo abbiamo posto per convenzione tale tabella essere t_1. La funzione $BJ(t_1, t_2, pred, cols, groups, pipe)$ restituisce il tempo necessario a portare a termine questo task. Ai fini del nostro modello, l'unica operazione rilevante in termini di costo è la scrittura dei dati su disco. Questo è dovuto

al fatto che il costo di caricamento delle due tabelle è a carico dei nodi figli. Come già specificato nella grammatica (Figura 4.2), t_1 viene letta con un task di tipo SB(), mentre t_2 può essere il risultato di una precedente operazione (per esempio uno SJ() o a sua volta di un altro BJ()), oppure può essere il caricamento di una tabella tramite un task di SC(). Analogamente allo shuffle join, filtro, proiezione e aggregazione possono essere opzionalmente applicati ed eseguiti in memoria centrale.

Una caratteristica fondamentale del broadcast join è che viene sempre eseguito in pipeline con un'altra operazione, sia essa di scan, shuffle join o broadcast join. Questo implica che il numero di waves necessarie dipende esclusivamente dal numero di partizioni di t_2. La quantità di dati che ogni core deve scrivere durante ogni wave è:

$$WSize = \frac{JSize(t_1, t_2, pred) \cdot Proj(cols)}{t_2.Part} \cdot \\ \cdot Group(JCard(t_1, t_2, pred), \prod_{a \in groups} a.Card)$$

Quindi il tempo necessario per completare questo task è ottenuto sommando il tempo necessario per la scrittura in ogni wave:

$$BJ(t_1, t_2, pred, cols, groups) = \lceil \frac{t_2.Part}{\#E \cdot \#EC} \rceil \cdot Write(WSize)$$

La Tabella 4.3 mostra le caratteristiche della tabella che viene restituita dal termine di questo task type t'.

4.4.6 Group By GB()

Il task di group by si occupa di portare a termine l'ultima aggregazione di una query GPSJ. Le tuple della tabella in input t sono già state precedentemente suddivise in $\#SB$ buckets. Le partizioni RDD in input vengono memorizzate nel disco locale di ogni executor. È la funzione $GB(t, pred, cols, groups)$ che restituisce il tempo necessario per completare il task. Ad ogni wave le operazioni portate a termine dal task sono:

(a) **Shuffle Read**: in questa prima fase vengono letti i bucket dalla tabella t. Una porzione di ogni bucket è memorizzata su ogni executor.

(b) **Aggregation**: in questa fase, che è il cuore del task, vengono raggruppate le tuple secondo quanto specificato negli attributi $groups$ e vengono calcolati i valori degli attributi generati dal raggruppamento.

(c) **Write**: in quest'ultima fase le tuple vengono scritte nuovamente su disco.

Un raggruppamento non può mai cominciare prima che tutte le tuple del relativo bucket siano caricate. Perciò il tempo necessario per portare a termine una wave è la somma del tempo necessario per la lettura in shuffle e la successiva scrittura delle tuple calcolate. La quantità di dati che devono essere letti da ogni core ad ogni wave è:

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

La quantità di dati che vanno scritti su disco al termine della computazione da ogni core ad ogni wave è:

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

In quest'ultima formula $hSel$ è una costante che rappresenta il fattore di selettività per le clausole di $having$ (di default $hSel = 0.33$); nel caso in cui non venga specificato alcun predicato $pred$, la costante $hSel$ viene considerata avere un valore pari ad 1 (non influisce calcolo del tempo complessivo di svolgimento del task). Nonostante una tale stima sia spesso piuttosto differente dall'effettivo valore di selettività, nelle soluzioni commerciali questa è una soluzione standard. Stime più dettagliate ed elaborate implicano assunzioni che non sarebbero coerenti con il nostro framework [5].

Il templo complessivo di svolgimento del task può essere calcolato come la somma dei tempi necessari per lo svolgimento di del raggruppamento in ogni singola wave:

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

La Tabella 4.3 mostra le caratteristiche della tabella che viene restituita dal termine di questo task type t'.

Capitolo 5

Un'applicazione per il controllo dei costi

L'applicazione che ho sviluppato permette di stimare il tempo necessario ad eseguire una query SQL su un determinato cluster. Tale stima è l'applicazione pratica del modello visto nel Capitolo 4. Il codice è scritto in HTML, PHP, CSS e JavaScript, perciò per lanciare l'applicazione è sufficiente un server web (per esempio Apache) con php-5.4 o superiore ed un client con JavaScript abilitato. Lato server viene usato php per generare le pagine, scambiare dati con il database e calcolare le funzioni di costo. Lato client viene usato JavaScript per visualizzare la rappresentazione grafica dell'albero di esecuzione e per alcuni estetismi. Nelle seguenti sezioni vediamo nel dettaglio come è strutturato il sistema e come vengono usate le varie parti.

L'applicazione permette ai visitatori di registrarsi e autenticarsi. Una volta effettuato l'accesso ciascun utente ha completo accesso a tutte le funzionalità del sistema:

- inserimento e cancellazione dei database;
- inserimento e cancellazione dei cluster;
- esecuzione della stima del tempo di esecuzione di una query specificando il cluster ed il database. Nella pagina principale è infatti possibile inserire il piano di esecuzione generato da Spark e caricarlo. Una volta caricato, il server restituisce al client una pagina con la grammatica generata da tale piano, una sua rappresentazione grafica, e la possibilità di inserire i parametri per il calcolo dei costi di esecuzione.

5.1 La struttura dell'applicazione

L'applicazione che ho realizzato è composta da tre parti principali:

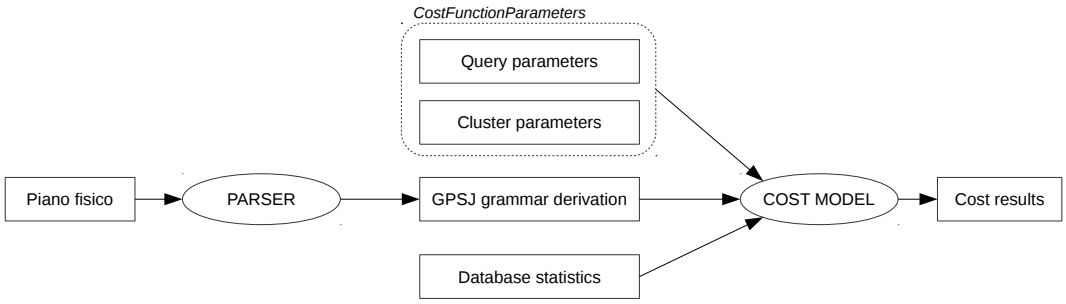


FIGURA 5.1: Struttura del modello di costo. Gli ovali rappresentano le operazioni del modello stesso, i rettangoli sono invece oggetti. Tali oggetti possono essere sia parametri presi in input, sia oggetti generati da un passaggio precedente.

- **Inserimento dei parametri**: consiste in una serie di sezioni che consentono all'utente di inserire i parametri sui quali eseguire la stima del tempo di esecuzione. Questi sono i parametri di cluster, la struttura del database ed i parametri di query.
- **Parsing del piano fisico**: in questa fase viene letto in input il piano fisico generato da Spark SQL e viene generata quella che abbiamo definito *GPSJ grammar derivation*. È un passaggio fondamentale, infatti è in questa fase che vengono estrapolati i dati della query e generato l'albero di esecuzione.
- **Calcolo del costo**: successivamente, vengono presi la GPSJ grammar derivation e i parametri con i quali eseguire il piano di costo, e viene calcolato il tempo necessario per l'esecuzione della query. È qui che viene effettivamente applicato il modello di costo visto nel Capitolo 4.

Vedremo nelle sezioni successive queste parti in dettaglio. È importante notare che le parti sono indipendenti l'una dall'altra e vengono messe in collegamento dall'interfaccia principale.

5.2 Parsing del piano fisico

Per applicare il modello di costo sul piano fisico inserito dall'utente, è necessario essere in grado di costruire, a partire da questo, il grafo aciclico di esecuzione dei job in spark. Tale operazione consiste in una prima fase di parsing del testo, che genera una lista di operazioni ed una fase successiva nella quale tale lista viene analizzata e ristrutturata per costruire l'albero di esecuzione. Ho supposto la possibilità che in futuro nuove versioni di Spark generino differenti piani di esecuzione. Quindi la prima parte, quella del parsing del testo, è intercambiabile in base alla versione di

TABELLA 5.1: Struttura delle varie operazioni del piano di esecuzione generato da Spark. Gli * indicano parti della riga di cui non ci interessa il contenuto. Dove abbiamo una lista di valori (per esempio la lista di attributi), la virgola funge da separatore.

Nome	Struttura
HiveTableScan	[*attributes*](*database*, *tabella*, *)
Filter	(*condizione*)
Project, TungstenProject	[*attributes*]
SortMergeJoin	[$attributi_1$],[$attributi_2$]
BroadcastHashJoin	[$attributi_1$],[$attributi_2$],*condizione*
TungstenAggregate	(key=*attributi*, functions=[*], output=[*attributi*])
TungstenSort	ingorata
ConvertToUnsafe	ignorata
ConvertToSafe	ignorata
TungstenExchange	*

Spark; infatti il linguaggio usato nel piano fisico è strettamente legato alla versione di Spark.

5.2.1 Struttura del piano fisico e primo livello di parsing

In questa sezione affronto la prima parte della generazione della GPSJ grammar derivation. Nonostante abbia garantito elasticità nel parsing in base alla versione di Spark, per ora ho implementato e gestito il solo piano fisico generato da Spark alla versione 1.5.0, quindi vediamo come viene effettuata la conversione di quest'ultimo.

Il piano di esecuzione che Spark genera è un file di testo nel quale in ogni riga abbiamo il nome dell'operazione da svolgere ed i suoi parametri. Non è stato banale effettuare l'analisi delle varie righe, infatti queste ultime hanno strutture differenti e sono ricche di eccezioni. Nella Tabella 5.1 ho schematizzato la struttura dei parametri delle varie operazioni; il tipo di operazione eseguito viene anteposto ai suoi parametri all'inizio della riga. Il piano di esecuzione riporta un gran numero di operazioni e di parametri per ciascuna di esse, ma non tutti ci interessano. Lo scopo ultimo del parsing è costruire la GPSJ grammar derivation per il calcolo del costo. Nel nostro modello abbiamo ignorato alcune condizioni e ne abbiamo approssimate altre, quindi gli unici parametri che ci interessano sono quelli usati all'interno del modello. Infatti alcune operazioni vengono completamente ignorate (Per esempio ConvertToSafe, ConvertToUnsafe, TungstenSort) in quanto denotano la presenza di un'operazione che nel nostro modello abbiamo stimato non influire quasi affatto sul tempo di esecuzione.

Le righe del piano generato da Spark cercano di simulare un albero delle operazioni. La struttura di tale visualizzazione è una navigazione dell'albero in depth-first, perciò se abbiamo due join a cascata le righe sono ordinate come mostrato in Figura 5.2. Questa struttura ha influito notevolmente sulle scelte adottate nel secondo

```
1       join(AB,C)
2         join(A,B)
3           scan(A)
4           scan(B)
5         scan(C)
```

FIGURA 5.2: Semplificazione della struttura del piano fisico generato da Spark. Si può osservare chiaramente la struttura depth-first.

livello di parsing.

5.2.2 Secondo livello di parsing

Ottenuta una lista di operazioni più strutturate, siamo in grado di passare alla seconda fase del parsing: strutturare queste operazioni in nodi e costruirci l'albero di esecuzione. Nel piano fisico abbiamo due tipi principali di operazioni, che ho definito come:

- Operazioni nodo: queste sono tutte quelle operazioni principali che ci consentono di identificare uno stage. Un esempio sono le operazioni di HiveTableScan, BroadcastHashJoin o SortMergeJoin. Definiscono senza ambiguità dove finisce uno stage e dove inizia quello successivo (sono sempre le prime operazioni di ciascuno stage).

- Operazioni NEL nodo: questo tipo di operazioni rappresentano il push down di aggregazione, proiezione e filtraggio. Non vanno a formare uno stage autonomo, ma vanno a "decorare" i nodi operazione già presenti. Come già detto la struttura del piano è un albero navigato in depth first, perciò queste operazioni vengono eseguite dopo l'operazione nodo; pertanto vanno memorizzate ed applicate alla prima operazione nodo successiva (Figura 5.3).

L'operazione $TungstenAggregate$ però costituisce un'eccezione a queste regole. L'aggregazione può subire il push down, dunque abbiamo due situazioni: il caso nel quale è stato operato il push down, quindi rappresenta una $operazionenelnodo$ e lavora all'interno di un altro nodo; il caso nel quale funge da $operazionenodo$, e in questo caso è essa stessa a creare un nuovo nodo sul quale svolgere l'operazione. In fase di parsing questa distinzione la possiamo operare esclusivamente analizzando la sua posizione rispetto al $TungstenExchange$. L'operazione di TungstenExchange rappresenta all'interno di Spark il fatto che viene effettuato lo shuffle dei dati. Purtroppo non ho potuto usare questa operazione, anzi che i le operazioni nodo, per ricreare l'albero di esecuzione per il fatto che alcune operazioni non effettuano shuffle (per esempio BroadcastHashJoin). Resta il fatto che un'operazione di

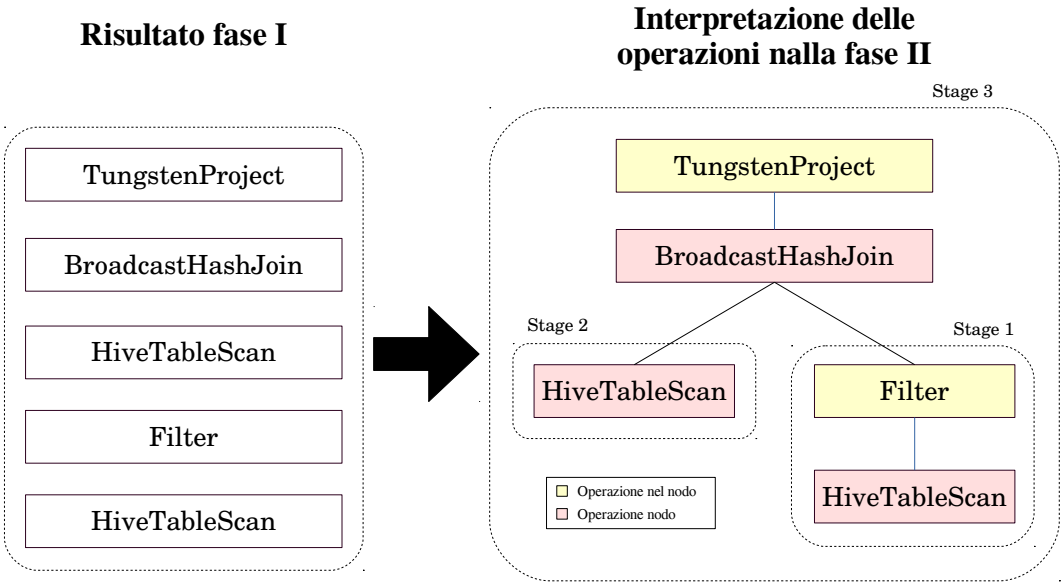


FIGURA 5.3: Interpretazione grafica del parsing. A sinistra viene visualizzata la lista di operazioni già interpretate dal primo livello. A destra viene mostrato il modo in cui le operazioni nel nodo vengono incluse all'interno delle operazioni nodo.

TABELLA 5.2: Traduzione delle operazioni del piano fisico Spark nella grammatica adottata.

Nome in Spark	Nome nella grammatica	Tipo di operazione
HiveTableScan seguito da TungstenExchange	Scan	nodo
HiveTableScan non seguito da TungstenExchange	Scan_&_Broadcast	nodo
Filter	filter	operazione nel nodo
Project, TungstenProject	project	operazione nel nodo
SortMergeJoin	Shuffle_Join	nodo
BroadcastHashJoin	Broadcast_Join	nodo
TungstenAggregate preceduto da TungstenExchange	Group_By	nodo
TungstenAggregate non preceduto da TungstenExchange	aggregate	operazione nel nodo

raggruppamento, se non preceduta da shuffle ($TungstenExchange$), viene applicata al nodo attualmente attivo. Se invece è preceduta da un'operazione di shuffle, istanzia un nuovo nodo nell'albero: il nodo caratterizzato dall'operazione di *Group by*. L'operazione TungstenExchange, rappresentando uno shuffle, ci torna utile anche per identificare se un $HiveTableScan$ rappresenta una operazione di $Scan$ o di $Scan\&Broadcast$: se seguita da shuffle si tratta di una Scan, altrimenti si tratta di una Scan & Broadcast.

Una volta analizzato il piano d'esecuzione, bisogna ricostruire la GPSJ grammar derivation. Il primo passo consiste nel tradurre le varie operazioni nella grammatica che abbiamo adottato. Nella Tablella 5.2 mostro come vengono tradotte tali operazioni e a cosa corrispondono.

```
1  SELECT * FROM lineitem, supplier WHERE l_suppkey=s_suppkey AND l_quantity
       <=25;
```

FIGURA 5.4: Un esempio di query.

```
1  [== Physical Plan ==]
2  [TungstenProject [o_orderkey#87L,o_custkey#88L,o_orderstatus#89,o_totalprice
       #90,o_orderdate#91,o_orderpriority#92,o_clerk#93,o_shippriority#94,
       o_comment#95,l_orderkey#96L,l_partkey#97L,l_suppkey#98L,l_linenumber#99,
       l_quantity#100,l_extendedprice#101,l_discount#102,l_tax#103,l_returnflag
       #104,l_linestatus#105,l_shipdate#106,l_commitdate#107,l_receiptdate#108,
       l_shipinstruct#109,l_shipmode#110,l_comment#111]]
3  [ SortMergeJoin [o_orderkey#87L], [l_orderkey#96L]]
4  [  TungstenSort [o_orderkey#87L ASC], false, 0]
5  [   TungstenExchange hashpartitioning(o_orderkey#87L)]
6  [    ConvertToUnsafe]
7  [     HiveTableScan [o_orderkey#87L,o_custkey#88L,o_orderstatus#89,o_totalprice
       #90,o_orderdate#91,o_orderpriority#92,o_clerk#93,o_shippriority#94,
       o_comment#95], (MetastoreRelation tpch_1tb, orders, None)]
8  [  TungstenSort [l_orderkey#96L ASC], false, 0]
9  [   TungstenExchange hashpartitioning(l_orderkey#96L)]
10 [    ConvertToUnsafe]
11 [     Filter (l_quantity#100 <= 25.0)]
12 [      HiveTableScan [l_orderkey#96L,l_partkey#97L,l_suppkey#98L,l_linenumber
       #99,l_quantity#100,l_extendedprice#101,l_discount#102,l_tax#103,
       l_returnflag#104,l_linestatus#105,l_shipdate#106,l_commitdate#107,
       l_receiptdate#108,l_shipinstruct#109,l_shipmode#110,l_comment#111], (
       MetastoreRelation tpch_1tb, lineitem, None)]
```

FIGURA 5.5: Piano fisico generato da Spark 1.5.0 per la query di esempio.

Vediamo un esempio: la query in Figura 5.4 genera in Spark il piano d'esecuzione riportato in Figura 5.5. Tale piano viene analizzato riga per riga. Vengono estratti da ogni riga i parametri che ci interessa memorizzare (per esempio attributi di proiezione, chiavi di join, ecc). Infine si analizzano tali operazioni per costruire i nodi dell'albero rispettando la grammatica. Il risultato consiste nell'albero rappresentato in Figura 5.6.

5.2.3 La grammatica in output

In questa sezione ci tengo a formalizzare la struttura ed il linguaggio della GPSJ grammar derivation. Innanzi tutto abbiamo una struttura ad albero ed ogni nodo è sempre caratterizzato dal nome dell'operazione. Tale operazione può avere uno dei seguenti valori:

- *Scan*: il nodo rappresenta un'operazione di scan, a livello di modello corrisponde al task SC(). Può memorizzare la lista degli attributi su cui fa proiezione ($project$), un eventuale filtro ($filter$) e un opzionale predicato di raggruppamento ($aggregate$).

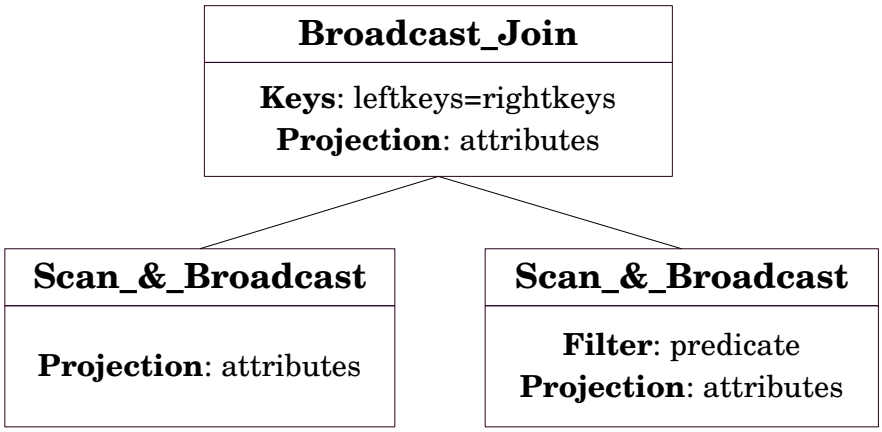


FIGURA 5.6: La rappresentazione grafica dell'albero di esecuzione ricostruito dopo il parsing.

- *Scan & Broadcast*: è gemella all'operazione di scan e corrisponde al task SB(), la differenza consiste nel fatto che non effettua uno shuffeling dei dati ma un broadcast.

- *Shuffle Join*: un nodo di Shuffle Join ha sempre un parametro che memorizza la lista di attributi sui quali fare proiezione dopo il join ($project$) e il parametro $join\_keys$, che tiene memoria di quali sono le chiavi sulle quali fare join. Quest'ultimo tiene memoria di due liste: la lista delle chiavi della tabella di sinistra e la lista delle corrispondenti chiavi nella tabella di destra. Anche questo nodo può opzionalmente memorizzare un predicato di raggruppamento ($aggregate$) ed uno di filtro ($filter$).

- *Broadcast Join*: questo nodo è gemello del precedente e rappresenta nel modello di costo il task BJ(). Come per lo Shuffle Join, non vi è scrittura ma solamente broadcast dei dati.

- *Group By*: questo nodo, se presente, è sempre uno solo all'interno dell'albero e costituisce sempre la radice. Corrisponde al task GB() ed ha sempre il parametro $aggregate$, che indica l'operazione di raggruppamento da applicare, ed il parametro $project$, che indica su quali attributi effettuare la proiezione.

Ognuno dei nodi precedentemente elencati è composto anche dal parametro $pipe$ che indica se si sta eseguendo il nodo in pipelining o se è necessario scrivere i dati su disco, perciò le operazioni che abbiamo visto distinguersi per la presenza del broadcast avranno tale valore impostato su $true$, mentre tutti gli altri hanno il valore impostato a $false$. Inoltre è importante ricordare che i nodi $Scan$ e $Scan\&Broadcast$ sono sempre foglie dell'albero di esecuzione, mentre gli altri

```
{
 "project": ["l_orderkey#224","l_partkey#225","l_suppkey#226","l_linenumber#227
     ","l_quantity#228","l_extendedprice#229","l_discount#230","l_tax#231","
     l_returnflag#232","l_linestatus#233","l_shipdate#234","l_commitdate#235","
     l_receiptdate#236","l_shipinstruct#237","l_shipmode#238","l_comment#239","
     s_suppkey#240","s_name#241","s_address#242","s_nationkey#243","s_phone#244
     ","s_acctbal#245","s_comment#246"],
 "operation": "Broadcast_join",
 "join_keys": [
  [
    "l_suppkey#226"
  ],
  [
    "s_suppkey#240"
  ]
 ],
 "id": 1,
 "pipe": true,
 "children": [
  {
    "filter": "l_quantity#228<=25.0",
    "operation": "Scan_&_Broadcast",
    "HDFS": {
      "t_name": "lineitem","0": "MetastoreRelationtpch","2": "None"
    },
    "columns": ["l_orderkey#224","l_partkey#225","l_suppkey#226","l_linenumber#
        227","l_quantity#228","l_extendedprice#229","l_discount#230","l_tax#231
        ","l_returnflag#232","l_linestatus#233","l_shipdate#234","l_commitdate#
        235","l_receiptdate#236","l_shipinstruct#237","l_shipmode#238","
        l_comment#239"],
    "id": 2,
    "pipe": true
  },
  {
    "operation": "Scan_&_Broadcast",
    "HDFS": {
      "t_name": "supplier","0": "MetastoreRelationtpch","2": "None"
    },
    "columns": [
      "s_suppkey#240","s_name#241","s_address#242","s_nationkey#243","s_phone#2
          44","s_acctbal#245","s_comment#246"
    ],
    "id": 3,
    "pipe": true
  }
 ]
}
```


FIGURA 5.7: File json generato dalla query di esempio dopo il parsing.

compongono sempre dei nodi intermedi o finali. Quindi questi ultimi hanno anche un parametro *children* che memorizza la lista dei figli del nodo. I due tipi di join citati in precedenza hanno sempre due figli, mentre il group by ne ha uno solo. Per quanto riguarda la struttura dell'albero nel dettaglio e le possibili combinazioni, rimando al capitolo 4 ed in particolare alla Figura 4.2.

Per la memorizzazione di tale grammatica ho deciso di utilizzare il formato Json. Nella Figura 5.7 si può vedere quale è la struttura dell'albero generato dall'esempio visto in precedenza.

5.3 Lo schema delle classi del modello di costo

Nel Capitolo 4 abbiamo visto quali sono i tipi di task che può eseguire Spark ed abbiamo calcolato per ognuno di essi il tempo necessario all'esecuzione. Nella sezione precedente ho mostrato quali sono i nodi di cui si compone la GPSJ grammar derivation ed ho associato ad ognuno di essi il task che caratterizza ciascun nodo. Queste informazioni, insieme ai parametri, ci permettono di calcolare il tempo complessivamente necessario ad eseguire la query SQL. Nella Figura 5.8 si può osservare qual'è la struttura delle più importanti classi che intervengono in questa fase. Il modello di costo consiste in un oggetto($CostModel$) che memorizza un riferimento a tutti i parametri necessari per il calcolo (database, query, parametri del cluster e dell'applicazione) e fornisce un metodo per eseguire le funzioni e calcolare il tempo di esecuzione. Durante l'esecuzione delle funzioni del modello, vengono memorizzati anche alcuni risultati intermedi, per esempio sono molto importanti il tempo di lettura, il tempo di scrittura ed il tempo di rete di ciascun nodo. Inoltre vengono memorizzati sia il tempo di esecuzione in pipeline (tempo trascorso dall'inizio dell'esecuzione al completamento dell'operazione) che il tempo di esecuzione complessivo (secondi complessivi di lavoro del cluster). Tali valutazioni sono molto importanti per riuscire a capire quali sono le operazioni che richiedono maggior tempo ad eseguire. Questa informazione permette ad un ottimizzatore di migliorare i tempi di esecuzione delle query allocando più risorse dove necessario. Tutti i risultati parziali, tra cui quelli citati in precedenza, vengono memorizzati all'interno di ciascun nodo dell'albero. Per una maggiore trasparenza e semplicità di recupero di tali informazioni, il modello di costo restituisce anche la lista di tutti i nodi operazione interni all'albero di esecuzione.

Vista la struttura generale, entriamo ora nel dettaglio analizzando come ho implementato il modello di costo.

5.3.1 I Basic Brick

I basic brick sono gli oggetti corrispondenti ai basic brick del modello di costo. Vi sono quindi quattro tipi differenti di mattoni base:

- *Read*
- *SRead*
- *Write*
- *Broadcast*

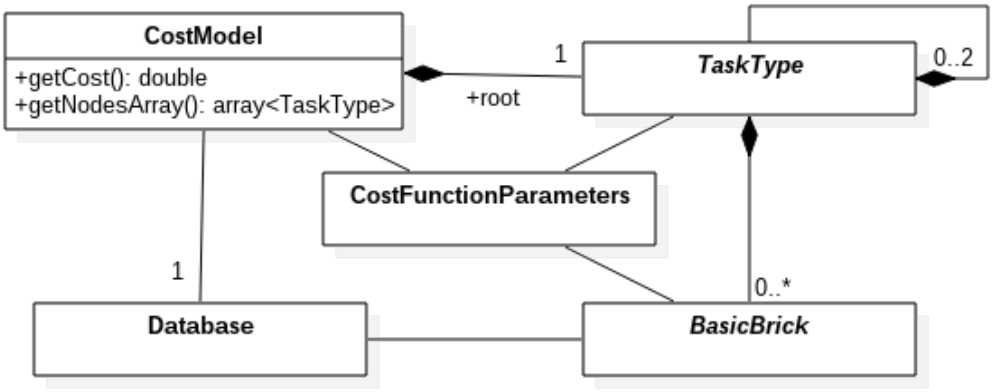


FIGURA 5.8: Diagramma delle principali classi all'interno dell'applicazione e della loro interazione. `CostModel` è la classe principale che si occupa di calcolare la funzione. Questa tiene un riferimento ai parametri ed uno al database usato. Dopo di che sfrutta i `TaskType` per svolgere le stime del modello di costo.

Le funzioni di calcolo dei tempi di esecuzione all'interno di ogni BasicBrick hanno bisogno solo di conoscere i parametri di cluster ed al massimo qualche altro parametro come la dimensione dei dati che si vanno ad elaborare, che gli vengono passati in fase di creazione. Non hanno alcuna necessità di sapere su quale base di dati si sta lavorando nello specifico, il che fornisce loro un buon grado di flessibilità all'interno del modello. La struttura e l'organizzazione dei BasicBrick la possiamo vedere nel diagramma in Figura 5.9. Questi oggetti sono accomunati dalle funzioni di inserimento e lettura del riferimento ai parametri e dalla funzione che restituisce il costo di esecuzione di tale operazione. Perciò ho implementato una classe astratta che si occupa di gestire questi aspetti base; ogni BasicBrick estende, direttamente o indirettamente, tale classe.

5.3.2 I Task Type

Ciascun nodo dell'albero di esecuzione corrisponde ad un tipo di task in Spark. Nel modello di costo (Capitolo 4) abbiamo definito le funzioni necessarie per la stima del tempo di esecuzione di ciascun task, perciò ho ritenuto logico creare un oggetto per ogni tipo di task e usare questi oggetti per ricostruire la struttura ad albero del piano di esecuzione. Alla base di questi oggetti vi è il TaskType: una classe astratta che specifica ed implementa gli aspetti fondamentali ed i metodi base di ciascun task. Questa classe gestisce la memorizzazione ed il recupero dei parametri (quali il database i parametri del cluster e di query). Inoltre vi sono implementate alcune funzioni necessarie in quasi tutti i tipi di task (per esempio la funzione $proj$

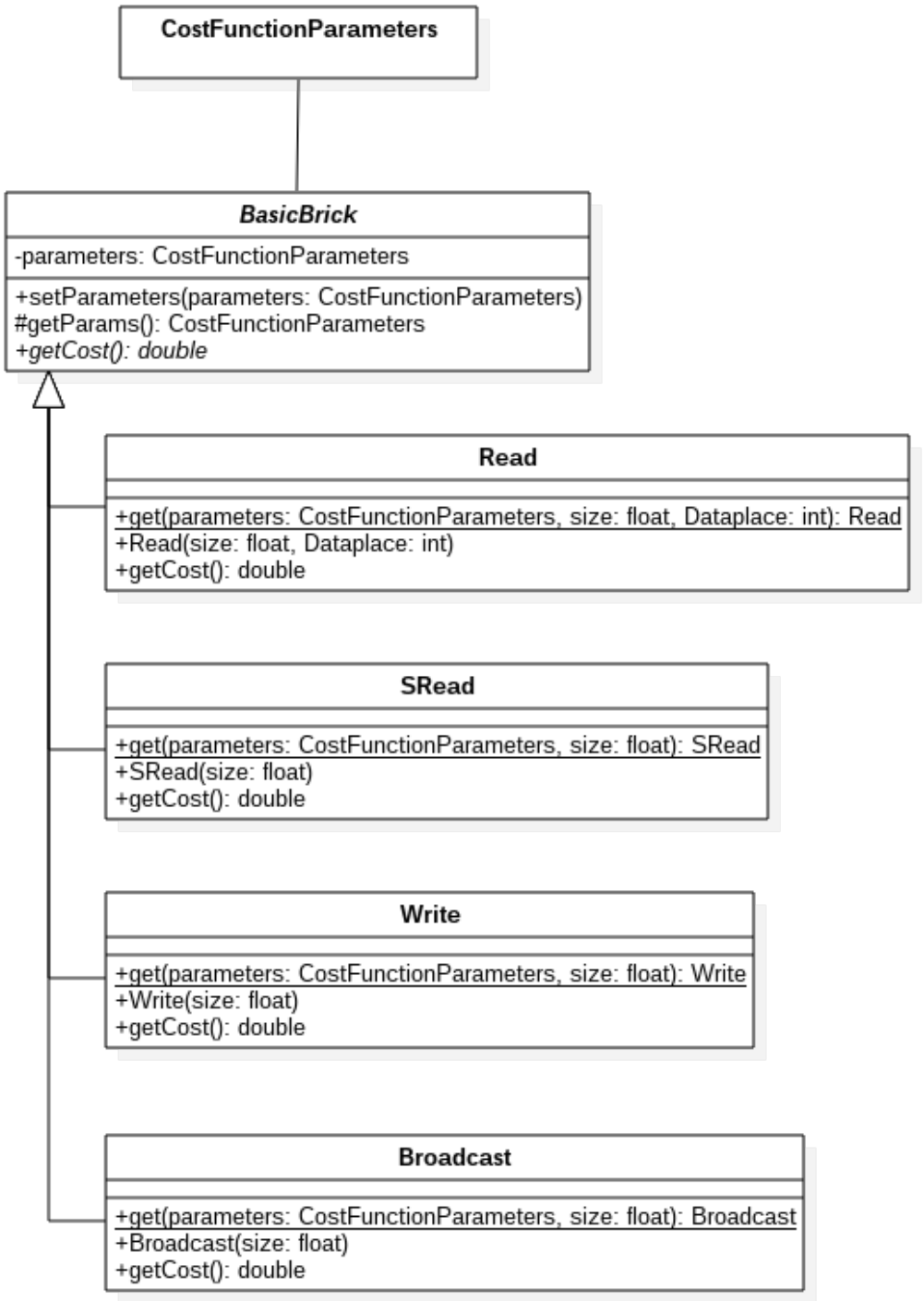


FIGURA 5.9: Diagramma delle classi dei `BasicBrick`. Si può notare qual'è lo scopo della classe astratta dalla quale i singoli tipi estendono.

o *group*) e funzioni che permettono di ottenere statistiche più dettagliate sui tempi di esecuzione della query.

La gestione ad oggetti dei nodi permette di memorizzare in fase di calcolo i risultati parziali e sia riusarli, risparmiando risorse computazionali, sia restituirli successivamente con le apposite chiamate a funzioni. Ciascun tipo di task può decidere di memorizzare e fornire l'accesso ai più variegati risultati parziali, ma alcuni di questi sono comuni a tutti e forzatamente memorizzati nella classe astratta. In particolare sto parlando dei costi di lettura, scrittura e trasmissione in rete e se ciascuno di essi ha influito o meno sul tempo computazionale totale (nel caso di pipelining l'operazione che richiede meno tempo viene parallelizzata e non influisce sul tempo totale in maniera attiva). I metodi standard $getReadT()$, $getWriteT()$ e $getNetworkT()$ memorizzano dei valori double, ma questo non toglie che il dato possa essere sovrascritto per memorizzare altri formati di dato. Un esempio di override lo vediamo nel TaskType SC() che, eseguendo tre computazioni (in locale, rack e cluster), sovrascrive i singoli tempi con degli array di 3 valori. Ritengo importante fare un accenno al fatto che di default tali metodi restituiscono un costo di esecuzione pari a 0, perciò i tempi vanno inizializzati solo nel caso in cui un task effettui operazioni che influiscono su quel tempo.

Ciascun task è strutturato in maniera tale da eseguire tutti i calcoli una sola volta, nelle chiamate successive riusa i valori memorizzati. Vi è un notevole vantaggio nelle performance computazionali, ma costituisce un ostacolo nel momento in cui si vogliano testare delle variazioni dei parametri. Perciò la classe TaskType è dotata anche di un apposito metodo che forza la ricomputazione del modello di costo sul nodo (e conseguentemente anche dei suoi figli a cascata) $askRecomputation()$. L'uso di tale funzione permette di sfruttare la struttura dati già creata senza dover ricreare un nuovo task da zero ad ogni variazione dei parametri.

5.4 I parametri

All'interno dell'applicazione hanno un ruolo fondamentale i parametri. Il tempo di esecuzione di ciascuna query SQL è strettamente dipendente dalle statistiche ottenute dal database e dai valori parametri di configurazione del cluster. Per maggiori approfondimenti rimando alla Sezione 4.2, dove sono stati trattati dettagliatamente, in questa sezione affronto le tecniche e le strutture adottate per implementare la loro memorizzazione. Ho supposto che in futuro, con eventuali raffinamenti della funzione di costo, possano essere necessari altri parametri oltre a quelli già memorizzati. Per permettere un facile evoluzione delle funzioni del modello, senza dover modificare tutto il sistema, ho creato due strutture fondamentali: il $Database$, che wrappa la struttura della base di dati sulla quale viene eseguita la query, ed

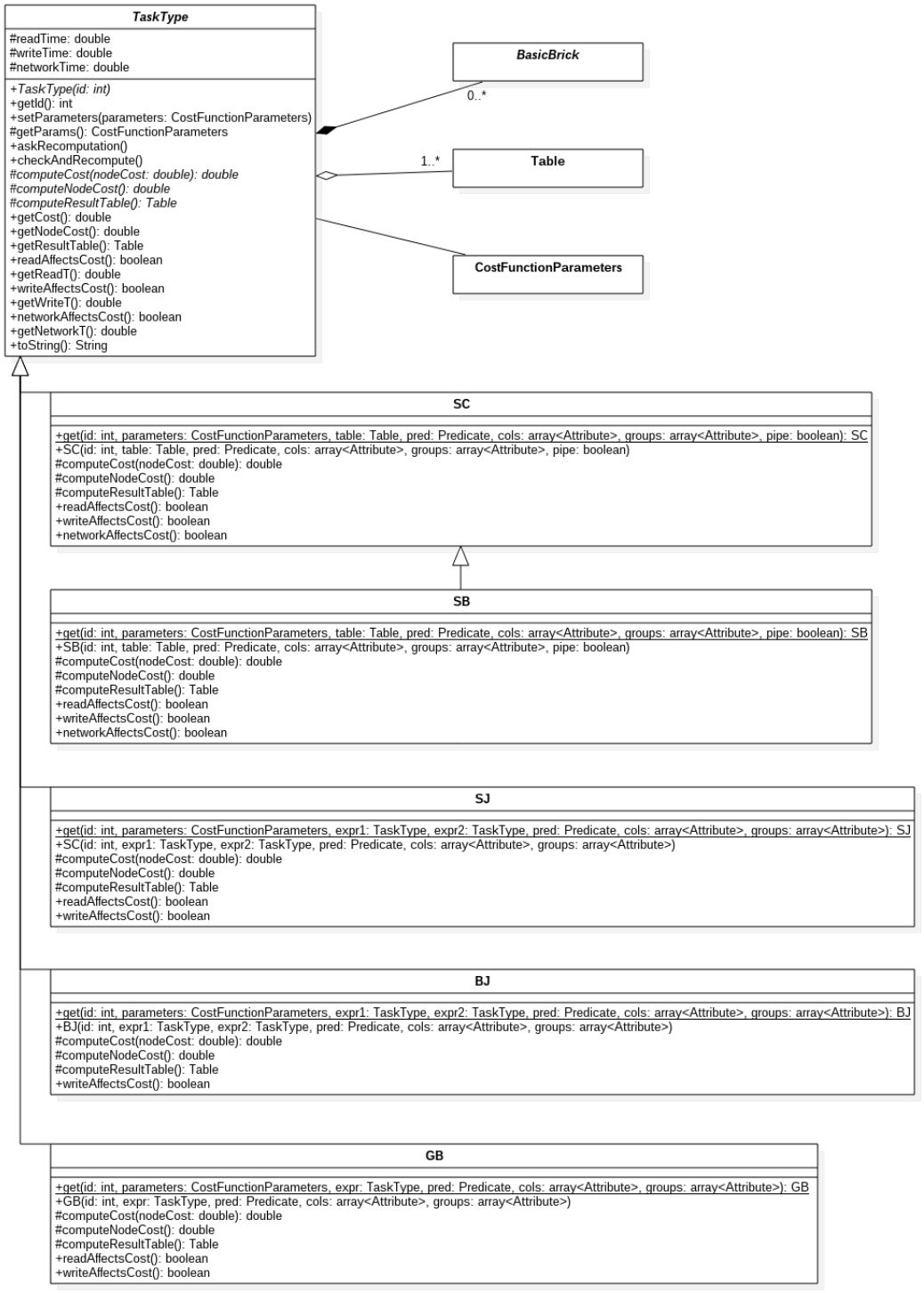


FIGURA 5.10: Diagramma delle classi della struttura adottata per i `TaskType`. Si può osservare il modo in cui vengono gestiti i parametri e come vengono usati i `BasicBrick`. SB() non estende direttamente `TaskType`, ma SC() in maniera tale da poter sfruttare alcune funzioni già implementate senza duplicazioni di codice.

la $CostFunctionParameters$. Il primo è un semplice oggetto definito da un nome che tiene memoria delle tabelle che caratterizzano il database e ne permette la consultazione. La seconda struttura dati tiene memoria di tutta quella serie di valori non strutturati o semistrutturati (array) che insieme costituiscono i parametri di cluster e di query. Nel caso in cui in futuro si ponga la necessita di fruire di altri parametri, sarà sufficiente aggiungerne la gestione in queste strutture dati senza dover modificare in alcun modo l'infrastruttura esterna e l'uso dei vari $TaskType$, del $CostModel$ o dei $BasicBrick$.

Ritengo degno di nota un cenno all'implementazione della classe *CostFunctionParameters*. Inizialmente avevo pensato di creare una classe che semplicemente memorizzasse delle variabili pubbliche accessibili da chiunque all'interno del progetto. Però mi sono reso conto che tale soluzione non era né elastica né adatta per la forma dei parametri in questione. Per esempio alcune variabili sono derivate da altre, è quindi un inutile spreco di memoria memorizzare tali variabili in fase di creazione dell'oggetto, inoltre nel caso di una eventuale modifica del parametro sarebbe necessario modificare a cascata tutti i parametri da lui derivati. Ho quindi adottato un'altra soluzione: l'oggetto è composto da variabili private e metodi pubblici. Le variabili memorizzano i parametri fondamentali, le funzioni rendono accessibili all'esterno i valori stessi di questi parametri e di quelli derivati. Sfruttando le chiamate a funzione la struttura resta dinamica alle varie modifiche. Un altro vantaggio dovuto a questa impostazione è quello di poter usare delle callback. Le callback sono funzioni esterne che è possibile usare all'interno del $CostFunctionParameters$. Questo permette di implementare esternamente alcuni comportamenti non statici. In particolare sono state usate per il calcolo dei throughput di disco e di rete in base al numero di processi. In questa prima versione del modello abbiamo deciso di adottare la seguente struttura: le funzioni riguardanti il throughput di rete restituiscono il valore stimandolo dalla velocità di rete (Sezione 4.2); il throughput di disco invece si basa su delle valutazioni concrete effettuate tramite sperimentazioni sul cluster. Il fatto di usare delle callback separa queste scelte dall'implementazione del contenitore dei parametri, il $CostFunctionParameters$. In futuro sarà possibile cambiare il modo in cui si stimano questi valori con una estrema facilità. Un problema nato dall'implementazione che è stata fatta del throuhput di disco è che il numero di valutazioni effettuate dall'utente è discreto. Se il modello richiede il throughput per un numero di processi non valutato discretamente, si considera il valore del throughput valutato per il maggior numero di processi.

Il modello di costo ($CostModel$) richiede che gli vengano passati il database, i parametri di cluster e di query ed la grammatica generata dalla query in fase di creazione. Una volta in possesso di tali parametri, procede in maniera autonoma ad

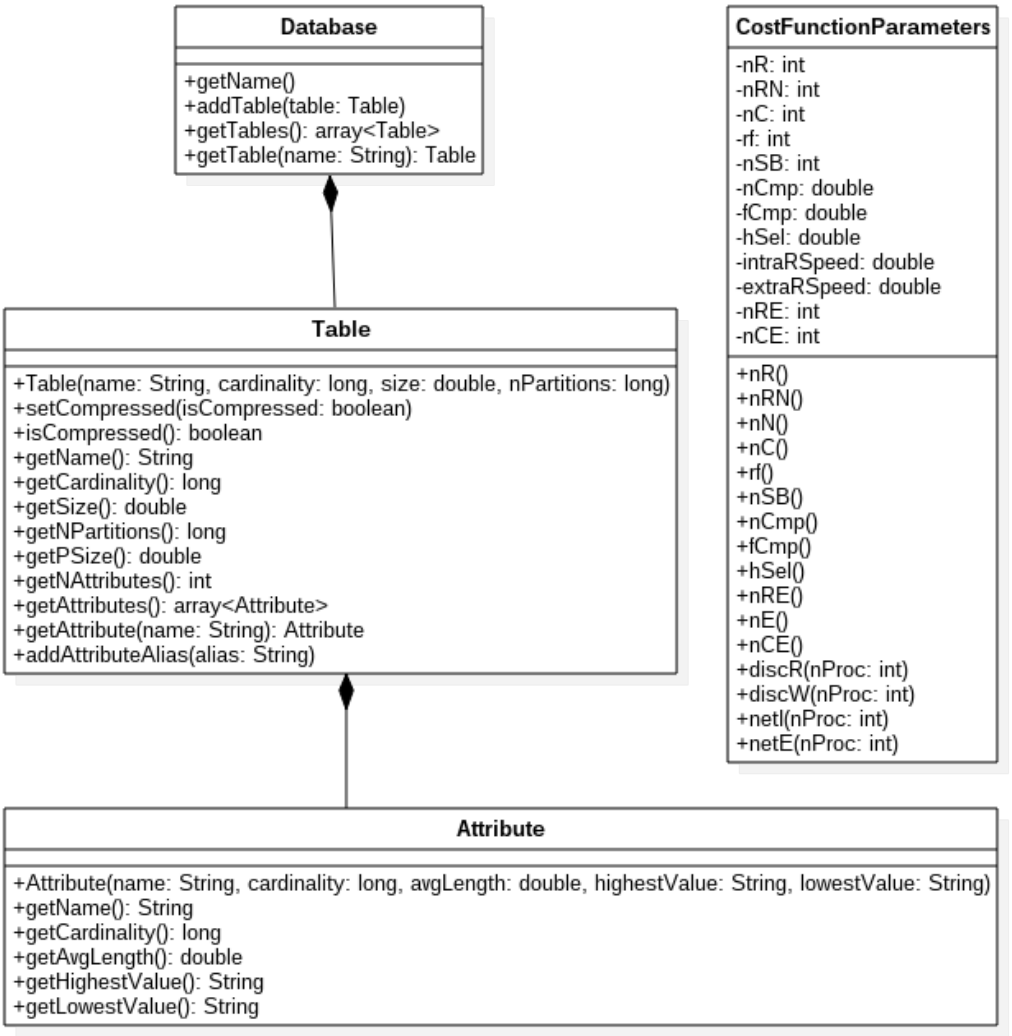


FIGURA 5.11: Diagramma delle classi che si occupano della gestione dei parametri di cluster, query e database. Sono mostrati solo le operazioni pubbliche per semplicità e chiarezza. Fa eccezione `CostFunctionParameters` per evidenziare la relazione che intercorre tra i parametri memorizzati e quelli messi a disposizione tramite le operazioni pubbliche.

assegnare a ciascun task i parametri a lui necessari e calcolare il tempo necessario per l'esecuzione.

5.5 Il database

Tutte le strutture memorizzate hanno un riferimento diretto o indiretto all'utente che le gestisce. Quindi una tabella fondamentale è quella che si occupa di raccogliere gli utenti che si registrano sulla piattaforma. Ciascun utente può salvare la

configurazione di uno o più cluster e la struttura dei vari database sui quali esegue le query. Il database è stato modellato in tabelle ed attributi. Ciascuna tabella memorizza i suoi nome, cardinalità, dimensione, numero di partizioni e se è memorizzata in formato compresso o meno. Gli attributi sono caratterizzati da nome, lunghezza media, valore più grande e più piccolo e dalla cardinalità. Quest'ultimo valore rappresenta il numero di chiavi distinte. La tabella $Cluster$ modella un cluster, per questa ragione tiene in memoria tutti i parametri fondamentali (non derivati) che lo caratterizzano. Il cluster è anche definito da due liste di dati: i valori del throughput di disco in scrittura ed in lettura. Della memorizzazione di queste liste se ne occupa la tabella $Throughput$.

La struttura del database è molto semplice. L'unica nota da fare riguarda il formato di memorizzazione dei dati. All'interno del del database i dati sono memorizzati in bytes (sia le dimensioni delle tabelle sia i valori di throughput o le velocità di condivisione dei dati in rete). Viste le dimensioni delle tabelle che si memorizzano, potrebbe essere eccessivo usare una simile unità di misura (potrebbero bastare MB o GB), ma i bytes ci assicurano una maggiore precisione e la completa assenza di perdita di precisione.

In Figura 5.12 è mostrato lo schema del database.

5.6 La rappresentazione grafica dell'albero

Per fornire all'utente una migliore cognizione della struttura dell'albero di esecuzione, di quali nodi è composto e del tipo di operazioni che vengono eseguite, ho inserito nell'applicazione una rappresentazione grafica della GPSJ grammar derivation. All'interno di tale grafica vediamo i nodi che compongono l'albero. Ogni nodo è caratterizzato dal nome dell'operazione che viene eseguita e dall'id univoco del nodo all'interno dell'albero. Visto che all'interno di un albero vi possono essere più task dello stesso tipo, l'operazione svolta non li identifica univocamente. Tramite l'id è possibile riconoscere di quale nodo si sta parlando sia nella GPSJ grammar derivation, che nella sua rappresentazione grafica che nella tabella dei costi. I task sono collegati ai task precedenti da degli archi. Per riconoscere in maniera immediata se lo scambio di dati tra i due task presenta uno shuffle o un broadcast, ho usato due colori diversi, rispettivamente nero e blu. Un esempio di una rappresentazione grafica generata con questo strumento è quello nella Figura 5.13.

Per generare quest'albero all'interno della pagina web è stato usato *d3.js*: un framework che permette di generare delle rappresentazione grafiche partendo da un dataset, prima ne associa i dati ad un DOM (*Document Object Model*) quindi fornisce degli strumenti all'utente per rappresentare tali oggetti come meglio preferisce

FIGURA 5.12: Il diagramma E-R delle entità presenti nel database.

(costruendo grafici a torta, alberi, grafi, tabelle... ecc). Una sua caratteristica, che lo differenzia dagli altri framework, è la straordinaria flessibilità ed efficienza dovute al fatto che usa standard web quali HTML, SVG e CSS. All'interno dell'applicazione questo framework è stato usato solo per rappresentare l'albero di esecuzione.

Inoltre basta un semplice click sul nodo per mostrare una scheda con tutti i dettagli dell'esecuzione del task in quel nodo (chiavi, predicati, filtri... ecc). Ho implementato questa funzionalità sfruttando la combinazione di PHP e JavaScript. Lato server costruisco queste finestre e le inserisco nella pagina; lato client javascript si occupa di nasconderle tutte e mostrarle al click.

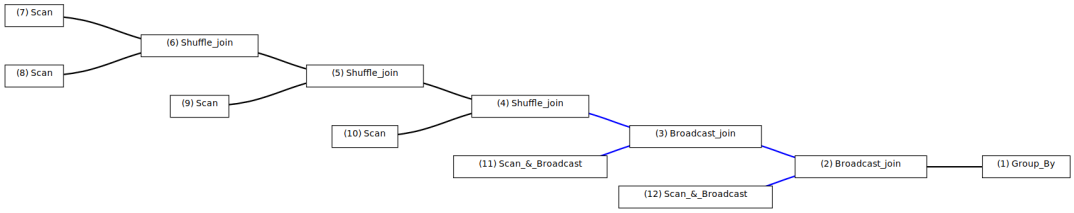


FIGURA 5.13: Un esempio di rappresentazione grafica dell'albero di esecuzione della query SQL generata con il framework d3.js. I nodi rappresentano dei task, gli archi collegano i task precedenti ai successivi. Se gli archi sono blu vi è uno shuffeling dei dati tra i nodi, quindi vi è scrittura; se l'arco è blu lo scambio dei dati avviene in broadcast.

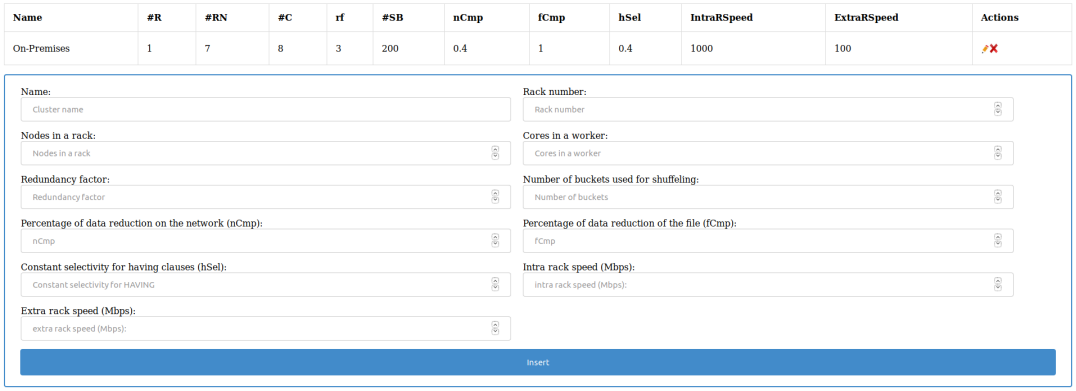


FIGURA 5.14: Un esempio della pagina `Cluster`. Questo utente ha già caricato la configurazione di un cluster (*On-Premises*). Sotto alla tabella vi e il form per l'inserimento di un nuovo cluster o la modifica di uno già presente.

5.7 L'interfaccia dell'applicazione

Per poter eseguire il calcolo del costo di una qualunque query SQL è necessario registrarsi, perciò quando si accede al sito viene mostrata solo una finestra che spiega lo scopo ed il funzionamento del modello di costo. Da questa pagina è possibile registrarsi o effettuare l'accesso, se l'utente è già registrato. Nella fase di registrazione viene richiesta la conferma via mail per ragioni di sicurezza, ma non viene richiesto all'utente di inserire alcuna informazione personale.

Una volta all'interno del sistema all'utente si aprono diverse possibilità. Tramite il menù avrà a disposizione tre pagine: *Cost model*, *Cluster* e *Database*. Ognuna di queste pagine ha uno scopo ben preciso, di seguito le analizziamo nel dettaglio.

5.7.1 Cluster

In questa pagina l'utente può gestire i suoi cluster, quelli sui quali vengono eseguite le query SQL che si vogliono testare. Quando la apre, la prima cosa che vede è la lista dei suoi cluster, se presenti, altrimenti un messaggio che lo avvisa del fatto che ancora non ha inserito nessun cluster. Ogni operazione che esegue all'interno di questa pagina lo notifica con degli avvisi se è stata completata con successo o meno (se per esempio il cluster è stato inserito o se c'è stato un problema). Le operazioni che l'utente può eseguire all'interno di questa pagina sono 3:

- **Inserimento**: nella seconda parte della pagina è sempre presente un form che permette l'inserimento di un nuovo cluster. Una volta compilati tutti i campi, tutti obbligatori per un corretto inserimento, è sufficiente premere il pulsante `Insert` per vedere il proprio cluster inserito nel sistema, se l'operazione termina correttamente.
- **Modifica**: cliccando sull'apposita icona alla destra del cluster che si vuole modificare, viene compilato il form in fondo alla pagina con i parametri del cluster che si sta modificando. Una volta applicate le modifiche, l'utente deve cliccare sul pulsante `Update` per salvarle.
- **Cancellazione**: per cancellare il cluster l'utente non deve fare altro che cliccare sull'apposita icona . Si aprirà una finestra popup che chiede di confermare l'operazione, se confermata il cluster verrà cancellato.

Nella Figura 5.14 è possibile vedere un esempio di questa pagina.

5.7.2 Database

Tramite questa pagina l'utente può gestire i suoi database. Come struttura è analoga alla pagina di gestione dei cluster: all'inizio della pagina viene mostrata la lista dei database, a seguire vi è il form per inserirne uno nuovo. A differenza dei cluster, i database non possono essere modificati. Infatti la loro complicata struttura non consente una semplice gestione. Già il solo lavoro di rendere dinamico l'inserimento di un nuovo database ha portato via moltissimo tempo, e non è questo lo scopo della tesi. Quindi le operazioni che si possono eseguire in questa pagina sono 2:

- **Inserimento**: il form di inserimento di un nuovo database è piuttosto articolato, a causa della sua struttura altrettanto articolata. Il form lo si può vedere come costituito da 3 atomi: i parametri del database, i parametri delle tabelle ed i parametri degli attributi. I parametri del database riguardano la vera e propria struttura del database, e consistono nel nome. Tale parametro può

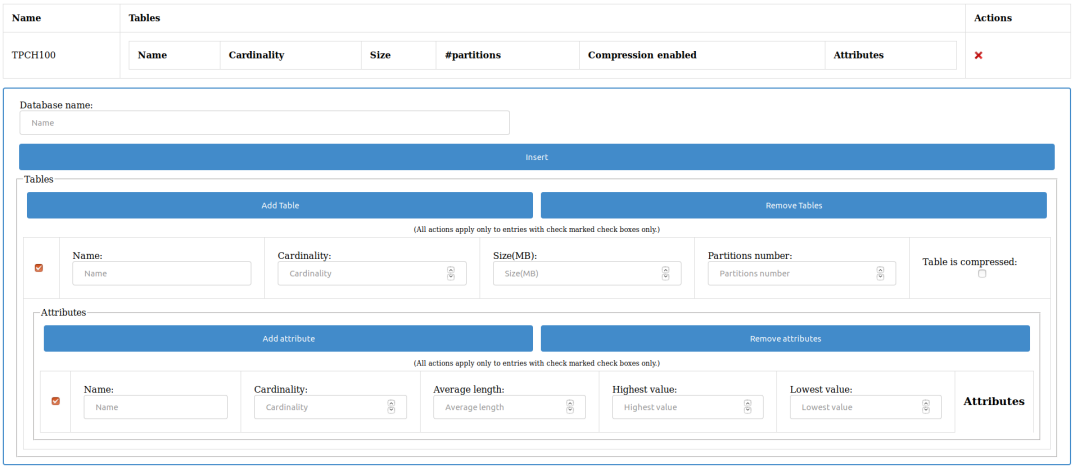


FIGURA 5.15: Un esempio della pagina `Database`. È già stato inserito un database di default (*TPCH100*), le tabelle e gli attributi sono stati compattati per motivi di spazio. Se si clicca sull'intestazione della tabelle del database, vengono espanse e visualizzate. Sotto alla tabella vi è il form per l'inserimento di un nuovo database.

essere inserito una sola volta. I parametri delle tabelle invece memorizzano i dati di ciascuna di esse. Essendo possibile averne più di una all'interno di uno stesso database, tramite il pulsante `Add table` è possibile aggiungere altre tabelle. I parametri degli attributi memorizzano i dati di ciascun attributo e si comportano in maniera analoga alle tabelle. Infatti ogni tabella è fornita di un pulsante `Add attribute` che consente l'aggiunta di un nuovo attributo alla lista. Vi sono anche altri due pulsanti: `Remove tables` e `Remove attributes` che rimuovono rispettivamente le tabelle e gli attributi selezionati in quella sezione. Una volta completati la configurazione della struttura e l'inserimento dei dati, cliccando sul pulsante `Insert` posizionato in cima al form verrà archiviata la struttura del database in memoria. Nella Figura 5.15 si può vedere lo scheletro base di tale form.

- **Cancellazione**: analogamente al cluster, accanto ad ogni database vi è l'apposita icona ✖ che, se cliccata, mostra un popup per la conferma. Nel caso in cui venga confermata la cancellazione, il database per intero viene cancellato. Per intero si intende che vengono cancellate anche tutte le tabelle e gli attributi a cascata, naturalmente.

Nella Figura 5.15 è possibile vedere un esempio di questa pagina.

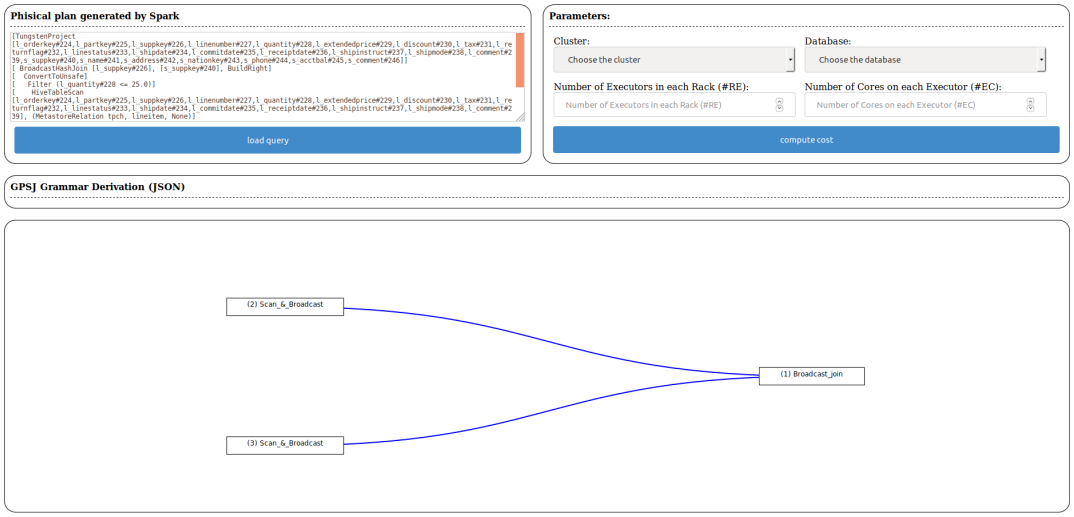
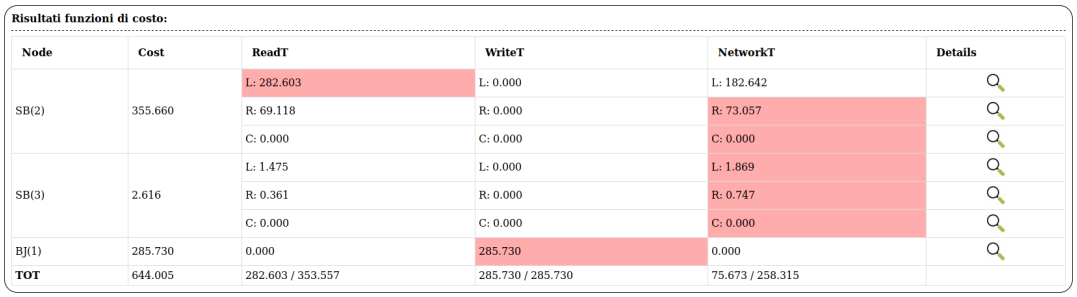


FIGURA 5.16: La pagina `Cost model` dopo aver caricato il piano di esecuzione, se è stato effettuato un parsing corretto.



Risultati funzioni di costo:

Node	Cost	ReadT	WriteT	NetworkT	Details
SB(2)	355.660	L: 282.603	L: 0.000	L: 182.642	
		R: 69.118	R: 0.000	R: 73.057	
		C: 0.000	C: 0.000	C: 0.000	
SB(3)	2.616	L: 1.475	L: 0.000	L: 1.869	
		R: 0.361	R: 0.000	R: 0.747	
		C: 0.000	C: 0.000	C: 0.000	
BJ(1)	285.730	0.000	285.730	0.000	
TOT	644.005	282.603 / 353.557	285.730 / 285.730	75.673 / 258.315	

FIGURA 5.17: Una tabella di esempio della stima dei tempi di esecuzione di una query SQL.

5.7.3 Cost model

Cost model è la pagina più importante, al suo interno vengono effettuati sia il parsing del piano fisico di una query SQL che il calcolo del costo applicando il modello di costo. Quando vi sia accede la pagina richiede solo l'inserimento del piano fisico. Una volta che questo piano è stato caricato (premendo il pulsante `load query`), se il parsing è andato a buon fine, la pagina cambia completamente aspetto: viene visualizzato il risultato del parsing sia come testo in formato json, sia come rappresentazione grafica dell'albero (Figura 5.16). Inoltre viene visualizzata una nuova sezione del form che permette l'inserimento dei parametri di esecuzione del cluster. Per eseguire il calcolo del costo è necessario compilare ognuno di questi campi, quindi cliccare il pulsante `compute cost`. Se il calcolo termina correttamente, in fondo alla pagina verrà mostrata una tabella contenente il tempo totale impiegato ed alcuni tempi parziali (Figura 5.17). In tale tabella vengono mostrati i tempi di ogni

nodo e, di ciascuno di essi, la parte di lettura (*ReadT*), quella di scrittura (*WriteT*) e quella di rete (*NetworkT*). Se le celle sono colorate in rosso vuol dire che influiscono attivamente sul tempo di esecuzione, altrimenti vengono eseguite in parallelo senza rallentare l'esecuzione complessiva. Se si posiziona il cursore sull'icona alla destra di ciascuna riga 🔍, vengono visualizzate alcune informazioni dettagliate di ciascun nodo. Tali informazioni possono variare da nodo a nodo, per ora riportano il numero di waves e le dimensioni dei dati rispettivamente letti, scritti o spediti in rete.

Se si vogliono provare nuove configurazioni del cluster o diversi parametri per l'applicazione, è sufficiente cambiare i rispettivi valori e premere nuovamente il pulsante `compute cost`.

5.8 La correttezza del modello

Sono stati condotti alcuni test per verificare la correttezza del modello adottato. Parlando di correttezza si intendono sia quella assoluta che quella relativa. La differenza tra le due consiste nel fatto che è verificata la prima condizione se tutte le stime coincidono in maniera perfetta con i risultati sperimentali dei test, se invece i dati discordano ma viene mantenuto un rapporto costante tra i due costi, allora si ha una correttezza relativa. Per questa raguione, dato un carico di lavoro $Q = \{q_1, ...q_n\}$, sono state definite due misure di accuratezza basate sulle due funzioni $t(q)$, che restituisce il tempo di esecuzione preso da Spark, e $et(q)$, che restituisce il tempo di esecuzione stimato dal modello. La prima misura dell'accuratezza è l'errore relativo $err(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{|t(q) - et(q)|}{t(q)}$ che definisce la capacità del modello di stimare in maniera adeguata il tempo di esecuzione, la seconda misura la correlazione statistica $cor(Q)$ tra $t()$ e $et()$ sul carico di lavoro Q, cioè misura quanto le stime del modello di costo siano correlate ai tempi di esecuzione in Spark. Per ragioni di stabilità la correlazione non è stata calcolata su carichi di lavoro tali che $|Q| < 10$.

La Tabella 5.3 mostra le caratteristiche di due cluster che sono stati usati per il testing. I clusters hanno dimensioni diverse, inoltre C_1 è su on-premisis, mentre C_2 è stato costruito sulla piattaforma cloud di Google. I tempi di throughput di rete e disco sono stati calcolati sperimentalmente tramite una procedura separata. La valutazione della correttezza è stata eseguita sfruttando 3 benchmarks molto noti:

- Q_1 è il benchmark di Big Data [6]; ha una dimensione di 120GB e la massima cardinalità delle tabelle vale $7.5 * 10^8$.

TABELLA 5.3: Caratteristiche dei cluster & uso del benchmark.

Name	Installation	#R	#N	#C	Main Mem.	Disk	Sw. Releases	Q_1	Q_2	Q_3
C_1	On Premises	2	7	8	32 GB	6 TB	Hadoop 2.6.0 + Spark 1.5.0	✓	✓	
C_2	Cloud (Google)	1	51	8	30 GB	512 GB	Hadoop 2.7.2 + Spark 1.6.2			✓

- Q_2 è il benchmark per TPC-H [7] di dimensione 100GB; la massima cardinalità delle tabelle vale $6 * 10^8$.
- Q_3 è il benchmark per TPC-H di dimensione 1TB; la massima cardinalità delle tabelle è $6 * 10^9$.

Mentre le query del primo cluster sono piuttosto semplici e tipicamente includono giusto uno o pochi task type, le query in Q_2 e Q_3 sono piuttosto articolate e mostrano la piena espressività delle query GPSJ. Perciò abbiamo scelto solo 4 query base (in TPC-H si chiamano $q1$,$q3$,$q6$ e $q10$) e le abbiamo usate per effettuare degli stress test al nostro modello. Nei vari test abbiamo variato il numero di executor ($\#E \in [1, .., 6]$ per C_1; $\#E \in \{10, 30, 50\}$ per C_2) e quello di core ($\#C \in \{2, 4, 6, 8\}$ per C_1; $\#C \in \{2, 6\}$ per C_2) per un totale di rispettivamente 24 e 6 configurazioni per C_1 e C_2. Tutte le query sono state eseguite tre volte ed è stato considerato il tempo medio delle tre, inoltre ciascun benchmark è stato eseguito solo sul cluster con l'appropriata potenza di calcolo.

In una prima fase lo scopo è stato quello di verificare l'accuratezza del modello di costo per ciascuno specifico tipo di task. Per fare ciò si sono considerate tutte le query di Q_1 ed una semplificazione di quelle in Q_2 e Q_3 per ottenere delle query che includessero il minor insieme possibile di task (per esempio non è possibile eseguire uno SJ() senza eseguire una SC()). Per testare il sistema al meglio abbiamo variato anche la selettività dei predicati (per esempio $Sel() \in \{0, 0.25, 0.5, 0.75, 1\}$) e la rapporto di riduzione delle aggregazioni ($Group() \in \{0.66, 0.95, 0.99\}$). Nella Tabella 5.4 è possibile osservare il risultato dei test, si può osservare come la precisione resta piuttosto alta (un margine di errore del 20%) e indipendente dal cluster sul quale viene eseguita la query.

La fase successiva ha previsto il test delle query GPSJ con la massima espressività. Come si può osservare dalla Tabella 5.5, anche in questo caso abbiamo osservato una percentuale di errore del 20% in media. Si è potuto constatare che l'accuratezza del modello di costo è stabile su diversi cluster e per query che differiscano di durata, numero di task e waves. Tale comportamento può essere meglio apprezzato nella figura 5.18 che, per ogni query q riassunta nella Tabella 5.5, confronta $t(q)$ e $et(q)$. È evidente che la maggior parte delle query sono vicine alla diagonale del

TABELLA 5.4: Accuratezza delle query minimali, in grassetto il task su cui il test si è concentrato.

Task types	*Bench.*	$\|Q\|$	*AVG t(q)*	err_r	*cor*
SC()	Q_1	120	27	0.25	0.94
	Q_2	120	570	0.26	0.92
	Q_3	30	1104	0.28	0.96
SC()+GB()	Q_1	72	566	0.13	0.98
	Q_2	72	302	0.24	0.98
	Q_3	18	309	0.25	0.99
SC()+SJ()	Q_1	120	1074	0.09	0.99
	Q_2	120	1559	0.26	0.99
	Q_3	30	2435	0.26	0.96
SC()+SB()+BJ()	Q_1	120	427	0.21	0.96
	Q_2	120	575	0.21	0.94
	Q_3	30	968	0.21	0.95
			Overall Average	0.22	0.96

grafico (asse nel quale $t(q) = et(q)$) e che, nonostante l'errore sia maggiore per query più lunghe, resta costante in termini di errore relativo.

TABELLA 5.5: *Accuratezza per le query con la massima espressività GPSJ.*

Base Query	*Bench.*	$\|Q\|$	*Total tasks*	*AVG waves*	*AVG t(q)*	err_r	*cor*
q_1	Q_2	24	1010	58	374	0.17	0.96
	Q_3	6	6263	40	666	0.23	
q_3	Q_2	24	1562	90	516	0.24	0.97
	Q_3	6	8197	52	374	0.15	
q_6	Q_2	24	1010	58	313	0.34	0.99
	Q_3	6	6263	40	529	0.05	
q_{10}	Q_2	24	1563	90	452	0.20	0.98
	Q_3	6	8198	52	656	0.19	
					Overall Average	0.20	0.98

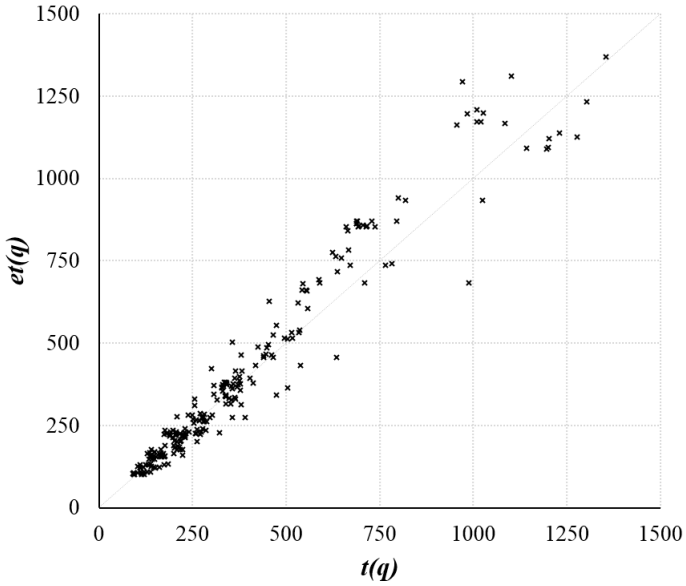


FIGURA 5.18: Grafico di dispersione del tempo di esecuzione effettivo VS tempo stimato dal modello per le query di Q_2.

Capitolo 6

Conclusioni

Il fenomeno dei Big Data ha causato una rivoluzione nella gestione delle grandi quantità di dati. Negli ultimi anni questo fenomeno è stato accelerato dal sempre maggior interesse delle aziende, che sono vogliono sfruttare al meglio le informazioni contenute in queste enormi quantità di dati grezzi. Questa accelerazione è stata caratterizzata da un frenetico sviluppo, da parte di un numero estremamente ristretto di persone, di nuove tecnologie in grado di gestire i Big Data.

In mezzo a questo ginepraio è riuscito ad emergere Hadoop, una piattaforma di calcolo distribuita basata su MapReduce. Le ragioni del suo successo sono molteplici e vanno dal fatto che è open-source alla possibilità di estenderlo tramite nuovi moduli, passando per i costi ridotti dei cluster di commodity hardware. Nonostante il suo sviluppo, Hadoop non costituisce ancora uno standard da poter applicare con metodo a progetti reali. Vi è spesso la necessità di ricorrere a quelle componenti esterne che nel tempo sono entrate a far parte dell'ecosistema di Hadoop. Uno dei componenti più importanti che è entrato a far parte dell'ecosistema è Spark. Spark sfrutta Hadoop e tra le varie funzionalità, ne ottimizza le esecuzioni facendo sì che, combinati assieme, costituiscano una soluzione estremamente performante di gestione dei Big Data. Oltre ciò, Spark introduce all'interno del sistema, tramite il modulo Spark SQL, la possibilità di eseguire direttamente query SQL.

Compito di un buon ottimizzatore è ridurre al minimo il tempo di esecuzione delle query, ma per poter effettuare questa operazione deve conoscere in quale maniera vengono eseguite le sue query e quali potrebbero essere i punti critici. Nel contempo un'azienda intenzionata ad acquistare o configurare un nuovo cluster Hadoop è interessata a ridurre al minimo i costi, quindi dimensioni e potenza del cluster, senza perdere eccessivamente nelle performance. Si è quindi cercato di modellare il comportamento di Hadoop durante l'esecuzione di delle query SQL. La carenza di esperti nel settore comporta che, nonostante si parli tanto del fenomeno, questo mondo risulta ancora molto distante e complesso. Essendo gli esperti dedicati quasi completamente ad uno sviluppo frenetico dei nuovi framework, viene riservato pochissimo tempo allo sviluppo di una documentazione ed una standardizzazione

di tali framework. Per questa ragione si è dovuta studiare e comprendere la struttura del framework prima di poterne modellare il comportamento. Svolta questa operazione, abbiamo capito quali parametri del cluster influiscono sull'esecuzione, come viene ottimizzata la query e quali sono le operazioni maggiormente critiche. Sfruttando tale modello un ottimizzatore è in grado di configurare il cluster o le query nella maniera maggiormente performante possibile. La funzione di costo si concentra sulle principale operazioni svolte dal sistema, come la lettura, la scrittura od il trasferimento in rete dei dati. La quantità di variabili che modificano tali costi è elevata, infatti un cluster Hadoop ha un considerevole numero di parametri. Principalmente le variabili sono dei parametri di configurazione, ma in alcuni casi si tratta di valori dinamici che all'interno del modello sono stati stimati. Ad ogni modo per semplicità tali parametri sono invariabili durante l'esecuzione.

Dopo aver teorizzato e formulato il modello, è stata creata un'applicazione che ne permettesse l'uso. L'applicazione è un portale web che consente a chiunque fosse interessato di stimare i tempi di esecuzione di alcune query all'interno del suo cluster. Questo potrebbe essere uno strumento formidabile nelle mani di un ottimizzatore, fornendogli a colpo d'occhio una stima dei tempi di esecuzione e delle operazioni critiche. Un uso corretto di questo portale potrebbe comportare per un'azienda una riduzione dei costi, sia a livello di tempi sia a livello di risorse, ed un aumento delle performance. Sviluppi futuri potrebbero testare l'esecuzione della query con piccole variazioni dei parametri per poi mostrare dei grafici dei tempi. È infatti noto che l'aumento delle risorse del cluster ha una riduzione dei tempi logaritmica. Tale sviluppo espliciterebbe all'utente quale è la configurazione ottimale del cluster per l'esecuzione della query inserita.

Sono stati poi effettuati dei test su due cluster per valutare la correttezza del modello. I dati sperimentali ottenuti si sono dimostrati molto vicini ai valori stimati matematicamente sia per quanto riguarda la quantità di dati letti e scritti, sia per quanto riguarda i tempi impiegati. Mentre i risultati non lasciano dubbi alla correttezza nella stima della quantità di dati letti e scritti, il calcolo dei tempi si è dimostrato molto vicino ai valori reali quando si ha un discreto numero di executor, se invece si usano uno o due executor si è notato un leggero scostamento. Il secondo caso è però di importanza minore dato che nell'uso classico di Spark, vengono usati più executor del cluster.

Bibliografia

[1] The parquet project. parquet.apache.org, 2016.

[2] A. F. Cárdenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5):253–263, 1975.

[3] S. Christodoulakis. Implications of certain assumptions in database performance evauation. *ACM Trans. Database Syst.*, 9(2):163–186, 1984.

[4] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[5] P. J. Haas, J. E. Lumby, and C. P. Zuzarte. Selectivity estimation for processing sql queries containing having clauses, 2004. US Patent 6,778,976.

[6] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[7] M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[8] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[9] A. Swami and K. B. Schiefer. On the estimation of join result sizes. In *EDBT*, pages 287–300, 1994.