ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA

SCUOLA DI SCIENZE
CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

ANALISI E BENCHMARKING DEL SISTEMA HIVE

Relazione finale in
LABORATORIO DI BASI DI DATI

Relatore
Prof. Matteo Golfarelli

Co-Relatore
Dott. Lorenzo Baldacci

Presentata da
Giovanni Di Meo

Seconda Sessione di Laurea
Anno Accademico 2014-2015

Sommario

Introduzione

Negli ultimi anni il volume dei dati prodotti quotidianamente nel mondo è aumentato tra il 40 e il 60% annuo. Ciò è dovuto alla nascita di nuove tecnologie (sensori, tablet e smartphone) e applicazioni web (si pensi ai social network come twitter, facebook e instagram) che ogni giorno producono una grande quantità di informazioni. Questi dati sono molto utili per le grandi aziende in quanto forniscono loro la possibilità di prendere decisioni basandosi su informazioni e analisi aggiornate. Ricerche e studi dimostrano infatti che le aziende che attribuiscono maggiore importanza ai processi decisionali basati sull'analisi dei dati hanno ottenuto netti miglioramenti (in termini di produzione e prestazioni) rispetto alle aziende che basano le decisioni unicamente sull'intuito e sull'esperienza. La raccolta e l'analisi dei dati sono alla base della strategia aziendale e del processo decisionale quotidiano. Le decisioni gestionali basate unicamente su intuito ed esperienza vengono oggi considerate poco affidabili, mentre le decisioni aziendali fanno sempre più spesso riferimento a "concrete informazioni analitiche". Non basta più registrare i dati delle attività di business (vendite, acquisti, costi) e quelli relativi all'andamento dei punti vendita; negli ultimi anni hanno assunto molta importanza anche nuove informazioni sui clienti, come abitudini, hobby o preferenze, reperibili ad esempio sui social media. Nasce così il termine *BigData* con il quale si intende un insieme di dati non omogenei, impossibile da gestire coi sistemi tradizionali come i DBMS relazionali e difficilmente immagazzinabili su un'unica macchina a causa del loro volume. È sorta quindi la necessità di creare sistemi capaci di sfruttare la capacità di calcolo e di memorizzazione su più macchine (cluster): il più famoso è la piattaforma *Hadoop* che mette a disposizione *HDFS*, un filesystem distribuito, il quale garantisce un discreto livello di tolleranza ai guasti, grazie a un sistema di ridondanza dei dati e a sofisticati algoritmi. Hadoop fornisce inoltre strumenti in grado di analizzare e processare una grande quantità di informazioni, trai quali *Hive*, che sfrutta *MapReduce* per l'interrogazione dei dati memorizzati sul database.

L'obiettivo di questa tesi è valutare le prestazioni del sistema Hive, utilizzando il *benchmark TPC-H*, su piattaforma Hadoop. In particolare si studieranno i piani di esecuzione adottati da MapReduce e la velocità con cui le query TPC-H vengono risolte.

Nel Primo capitolo si parla del fenomeno dei BigData, illustrando i motivi della loro nascita e cosa li caratterizza. Il capitolo proseguirà prsentando la piattaforma Hadoop: ne verranno mostrate le componenti principali, l'architettura di HDFS e i suoi componenti. Si passerà poi al movimento NoSQL, accennando i motivi della sua nascita, le caratteristiche generali e si elencheranno alcuni modelli dati NoSQL.

Il Secondo Capitolo invece si concenterà sul sistema Hive, architettura e linguaggio. Verrà descritto il formato file PARQUET; sarà introdotto il funzionamento di *MapReduce e* i principali operatori utilizzati. Infine si mostreranno in dettaglio i piani di esecuzione di alcune query TPC-H.

Il Terzo Capitolo illusterà i test sperimentali effettuati. Si accennerà brevemente al Benchmark TPC-H e alle caratteristiche delle query messe a disposizione; si forniranno informazioni sul cluster su cui sono stati eseguiti i test e infine verranno discussi i risultati ottenuti.

1. Evoluzione dei sistemi di gestione dei dati

Negli anni Sessanta, le uniche tecnologie disponibili consentivano di raccogliere su supporti magnetici i dati relativi ai processi aziendali. Le uniche analisi che potevano essere svolte erano statiche e si limitavano alla sola estrazione dei dati raccolti. Con l'avvento dei database relazionali e del linguaggio SQL *(Structured Query Language),* negli anni Ottanta, l'analisi dei dati assume una certa dinamicità: l'SQL infatti consente di estrarre in maniera semplice i dati, sia in modo aggregato, sia a livello di massimo dettaglio. Le attività di analisi avvengono su basi di dati *operazionali*, ovvero sistemi di tipo OLTP (*On Line Transaction Processing*) caratterizzati e ottimizzati prevalentemente per operazioni di tipo transazionale (inserimento, cancellazione e modifiche dei dati), piuttosto che per la lettura e l'analisi di grandi quantità di record. La maggior parte dei sistemi OLTP offrono una limitata storicizzazione dei dati, ma molto spesso risulta complesso ricostruire la situazione dei dati nel passato. Inoltre vi sono sempre più contesti in cui sono presenti numerose applicazioni che non condividono la stessa sorgente, e i cui dati sono replicati su macchine diverse e da esse manipolati, e questo non garantisce uniformità e coerenza.

La difficoltà nell'effettuare l'analisi dei dati direttamente sulle fonti operazionali ha portato, a partire dagli anni Novanta, alla creazione di database progettati appositamente per l'integrazione dei dati e l'analisi: nascono così i *Data Warehouse*, database che contengono dati integrati, consistenti e certificati, relativi ai processi di business aziendali.

All'inizio del nuovo millennio nascono nuove tecnologie, i *BigData*: un insieme di tecnolgie e fattori evolutivi volti all'analisi complessa di grandi moli di dati eterogenei e/o destrutturati. Accade sempre più spesso che le aziende abbiano la necessità di analizzare dati per i quali non possiedono strumenti adeguati in grado di elaborarli a causa della loro volume. Nonostante le aziende abbiano la possibilità di accedere a questi dati con i tradizionali strumenti messi a disposizione, non sono tuttavia in grado di estrapolarne valore perchè molto spesso essi si presentano nella loro versione più grezza, oppure in formati semistrutturati o addirittura non strutturati. Nel corso degli anni si sono resi disponibili dati che, per tipologie e per numerosità, hanno contribuito a far nascere il fenomeno dei BigData.

1.1 Fenomeno BigData

Il termine BigData viene applicato a dati e informazioni che non possono essere processati o analizzati utilizzando gli strumenti tradizionali.

Tradizionalmente le basi di dati operazionali risiedono su database relazionali RDBMS(*Relational Database Management System*), progettati utilizzando tecniche di normalizzazione (proprietà ACID: *Atomicity, Consistency, Isolation* e *Durability*) che facilitano le attività transazionali di inserimento, modifica e cancellazione dei dati, ottimizzandone le prestazioni.

I database normalizzati però non sono adatti alle analisi, per questo motivo si sfruttano appositi database, detti *Data Warehouse* che permettono di ottimizzare le performance di interrogazione. I dati provenienti dalle fonti operazionali vengono aggiunti in maniera incrementale all'interno dei Data Warehouse, garantendone la storicizzazione. Questi sistemi però, in presenza di gandi moli di dati, sono caratterizzati da una storicizzazione molto onerosa che nel tempo potrebbe risultare ingestibile dal punto di vista delle risorse e dei costi.

I dati operazionali, a seconda del business, possono assumere volumi rilevanti. Si prenda come esempio l'ambito bancario: considerando solamente una parte del patrimonio dei dati della banca, per ogni cliente devono essere registrati i saldi giornalieri dei conti e tutte le movimentazioni; inoltre anche la velocità con cui i dati vengono prodotti è un aspetto critico che deve essere tenuto in considerazione, parallelamente alla mole di informazioni da mantenere in memoria.

Le tecniche legate ai database relazionali, molto spesso, non riescono a “tenere testa” alla quantità di dati generati e alla velocità con cui essi vengono prodotti.

Le problematiche citate richiedono tecnologie diverse dagli RDBMS, tecnologie che consentano, senza investimenti proibitivi, di ottenere potenza di calcolo e scalabilità.

1.2 Caratteristiche dei dati

I BigData rappresentano quindi tutti quei dati disponibili in enorme quantità, che possono presentarsi con formati semistrutturati o addirittura destrutturati, prodotti con estrema velocità.

I BigData si caratterizzano per *le cosiddette“ 3 v “*: volume, varietà e velocità.

Il primo aspetto che caratterizza i BigData è il *volume*. Dati generati dall'utente attraverso strumenti del Web 2.0 o sistemi gestionali, oppure prodotti automaticamente da macchine, possono assumere quantità rilevanti, non più gestibili con strumenti di database tradizionali.
Ogni giorno viene generata un'impressionate mole di dati: solo Twitter e Facebook ne generano piu` di 7 TeraByte (TB) quotidianamente. Il volume di dati che al giorno d'oggi si memorizza sta esplodendo. Uno dei principi chiave per operare con i BigData è la memorizzazione di tutti i dati grezzi/originali, indipendentemente dal loro immediato utilizzo, poiché ogni operazione di pulizia o scarto potrebbe portare all'eliminazione di informazioni utili in futuro. E' evidente che, così facendo, il volume di dati da gestire diventi estremamente elevato.
In certi casi, si potrebbe pensare di utilizzare dei normali RDBMS, ma questo presuppone di investire cifre elevatissime sia per lo storage, sia per la potenza di calcolo necessaria per l'elaborazione. Tali investimenti potrebbero rivelarsi non giustificabili alla luce dei risultati ottenuti in termini di performance. Esistono pertanto soluzioni basate su architetture hardware MPP (*Massive Parallel Processing*) utilizzate in ambito data warehousing, che suddividono il volume di dati tra diversi processori, ognuno dei quali ha una propria memoria e un proprio sistema operativo. Queste architetture risolvono il problema del volume, ma non sono adatte a far fronte a un'altra caratteristica dei BigData: l'eterogeneità dei formati e la mancanza di struttura. Esistono quindi soluzioni e tecnologie alternative che permettono di gestire e analizzare al meglio l'intera mole di dati, con l'obiettivo di ottenere informazioni a supporto del business che si sta considerando. Tra le tecnologie open source, la più diffusa e utilizzata è *Apache Hadoop*, un framework in grado di processare grandi quantità di dati a costi contenuti.
Con l'esplosione dei sensori, degli smartphone, degli strumenti del Web 2.0 e dei social network i dati si sono "complicati", ovvero non presentano più una struttura predefinita e quindi non sono più riconducibili ad uno schema tabellare, ma possono presentare un formato semistrutturato o destrutturato, non più rappresentabile in modo efficiente utilizzando un database relazionale.
La diversità di formati e, spesso, l'assenza di una struttura sono il secondo aspetto che caratterizza i BigData. La *varietà* perciò, ha portato un drastico cambiamento all'interno dei processi analitici. Per il salvataggio dei dati semistrutturati, molto spesso la scelta ricade quindi sui cosidetti *database NoSql*, che forniscono i meccanismi adatti a organizzare i dati

ma, allo stesso tempo, non impongono uno schema predefinito, come invece avviene per i database relazionali; infatti vengono anche detti *schemaless database*. La mancanza di schema, che negli RDBMS deve essere progettata prima dello sviluppo, consente ai database NoSql di adattarsi alla varietà dei dati. Le imprese dunque, per poter sfruttare l'opportunità offerta dai BigData, devono essere in grado di gestire e analizzare tutti i tipi di dati, sia relazionali che non relazionali.

La *velocità* con cui i nuovi dati si rendono disponibili è il terzo fattore con cui è possibile identificare i BigData. La sfida per le aziende consiste nella capacità di sfruttare i dati provenienti ad alte velocità con altrettanta rapidità, estrapolando le informazioni utili per il business, minimizzando i tempi di elaborazione. A volte, essere in vantaggio rispetto alla concorrenza, significa identificare un problema, una tendenza o un'opportunità in pochi secondi, prima di chiunque altro; quindi, per poter trovare informazioni utili, le aziende devono possedere gli strumenti adatti ed essere in grado di analizzare tali dati (quasi) in tempo reale.

Diversamente dai sistemi tradizionali, la tipologia e la quantità di fonti diverse da cui possono provenire i dati sono molteplici. Prima del fenomeno dei BigData si aveva a che fare con fonti operazionali costituite prevalentemente da database relazionali, e quindi caratterizzate da dati strutturati. Ora, la presenza dei social network, dei sensori di controllo, del web e di qualsiasi altro dispositivo elettronico che genera masse d'informazioni (spesso semistrutturate o destrutturate) portano ad avere una moltitudine di fonti diverse da acquisire, che utilizzano tecniche diverse.

Negli ultimi anni con l'avvento dei BigData è emersa la necessità di lavorare con database sempre più flessibili, ma sopprattutto scalabili. Come si è detto, le tecnologie tradizionali utilizzate nel contesto dei BigData pongono due problemi che non possono essere trascurati:

- Gestione di una grandissima mole di dati;
- Presenza di dati non strutturati o semistrutturati;

Tali esigenze hanno portato allo sviluppo di nuovi modelli di gestione dei dati che si allontanano dal modello relazionale. La più diffusa e conosciuta è la piattaforma Hadoop.

1.3 Hadoop e NoSQL

Hadoop[2] (dal nome dell'elefantino di pezza del figlio di uno dei creatori) è un framework Open Source di Apache, affidabile e scalabile, finalizzato al calcolo distribuito di grandi quantità di dati. Hadoop nasce all'interno del progetto Nutch (sotto-progetto di Apache Lucene). Gli stessi creatori di Nutch, Doug Cutting e Michael J. Cafarella, a partire dal 2004, sfruttando le tecnologie di *Google File System* e *Google MapReduce*, svilupparono il primo prototipo di Hadoop. Allora rappresentava solamente un componente di Nutch in grado di migliorarne la scalabilità e le prestazioni. Hadoop divenne un progetto indipendete di Apache solamente quando Yahoo!, durante la ristrutturazione del sistema di generazione degli indici per il proprio motore di ricerca, assunse nel 2008 Doug Cutting, al quale fu assegnato un team di sviluppo dedicato e le risorse necessarie per sviluppare la prima release di Hadoop.
Oltre ad offrire un insieme di librerie di alto livello più semplici da utilizzare, Hadoop sfrutta la replicazione dei dati sui singoli nodi per migliorare i tempi di accesso, trascurando così la latenza dovuta alla rete. Le attività (in lettura) di gestione dei calcoli e di elaborazione di grandi moli di dati, che caratterizzano Hadoop, risultano essere esattamente l'opposto rispetto alle attività svolte da un database relazionale OLTP *(On Line Transaction Processing*), dove le singole transazioni interessano solamente pochi record. L'utilizzo di Hadoop in tali scenari non sarebbe efficiente poichè si tratta di attività gestite in modo ottimale dagli RDBMS.
Il nucleo centrale della piattaforma è costuito da 4 componenti essenziali:

- *Hadoop Common*: rappresenta lo strato di software comune che fornisce le funzioni di supporto agli altri moduli.
- *HDFS* (*Hadoop Distributed File System*): come riportato nella documentazione ufficiale, HDFS è il filesystem distribuito di Hadoop, progettato appositamente per essere eseguito su commodity hardware. Quando la mole di dati diventa "troppo grande" per la capacità di memorizzazione di una singola macchina, diventa necessario partizionare i dati su un certo numero di macchine separate. I filesystem che gestiscono l'archiviazione dei dati mediante una rete di macchine sono chiamati filesystem distribuiti. Rispetto ai normali filesystem, quelli distribuiti si basano sulla comunicazione in rete, per questo risultano essere più complessi.

- *Yarn*: si occupa della schedulazione dei task, ossia delle sotto-attività che compongono le fasi *map* e *reduce.*
- *MapReduce:* si occupa dell'esecuzione dei calcoli. Lavora secondo il principio "divide et impera": un problema complesso, che utilizza una gran mole di informazioni, viene suddiviso, assieme ai relativi dati, in piccole parti processate in modo autonomo e, una volta che ciascuna parte del problema viene "risolta", i vari risultati parziali sono "ridotti" a un unico risultato finale.

HDFS e MapReduce rappresentano il cuore del framework Hadoop; affinchè la computazione possa essere portata a termine, HDFS e MapReduce devono collaborare fra loro. A questi è poi possibile aggiungere tutti i componenti che fanno parte dell'ecosistema di Hadoop e che svolgono numerose differenti funzionalità, innestate sulla parte *core*.
Hadoop è un sistema:

- *Altamente affidabile*: essendo pensato per un cluster di commodity hardware, che può essere frequentemente soggetto a problemi, esso permette di facilitare la sotituzione di uno o più nodi in caso di guasti.
- *Scalabile*: la capacità computazionale del cluster Hadoop può essere incrementata o decrementata semplicemente aggiungendo o togliendo nodi al cluster.
- Dal punto di vista architetturale, in un cluster Hadoop non tutti i nodi sono uguali, ma esistono due tipologie:
    - *Nodo master*: esegue i processi di coordinamento di HDFS e MapReduce.
    - *Nodo worker*: utilizzato per la memorizzazione dei risultati e per l'esecuzione delle operazioni di calcolo.

HDFS

Hadoop Distributed File System (HDFS) è stato progettato per la gestione dei flussi e memorizzazione affidabile di grandi volumi di dati; in particolare, ha come funzione primaria quella di gestire l'input e l'output dei job MapReduce. Gli aspetti principali che lo caratterizzano sono:

- *Very Large Files*: non esiste un limite esplicito sulle dimensioni dei file contenuti al suo interno. Ad oggi vi sono cluster Hadoop, come per esempio quello di Yahoo!, che arrivano a gestire *Petabytes* di dati.
- *Streaming Data Access*: è particolarmente adatto per applicazioni che elaborano grandi quantità di dati. Questo perchè il tempo che occorre per accedere all'intero set di dati è relativamente trascurabile rispetto al tempo di latenza dovuto alla lettura di un solo record.
- *Commodity Hardware*: è stato progettato per essere eseguito su cluster di commodity hardware, ovvero hardware a basso costo, in modo tale da aumentare la tolleranza ai guasti (fault-tolerance), molto frequenti quando si ha a che fare con cluster di grandi dimensioni.

I file all'interno di HDFS, vengono partizionati in uno o più blocchi (*blocks*), ognuno di default da 128 MB (dimensione modificabile). Diversamente da altri filesystem, se un file risulta essere più piccolo della dimensione del blocco, non viene allocato un blocco "intero", ma soltanto la dimensione necessaria al file in questione, risparmiando così spazio utilizzabile. Affinchè venga mantenuto un certo grado di tolleranza ai guasti (*fault-tollerance*) e disponibilità (*availability*), HDFS prevede che i blocchi dei file vengano replicati e memorizzati, come unità indipendenti, fra i nodi del cluster. Se un blocco risulta non più disponibile, la copia che risiede su un altro nodo ne prende il posto, in modo completamente trasparente all'utente. Le repliche sono utilizzate sia per garantire l'accesso a tutti i dati, anche in presenza di problemi a uno o più nodi, sia per migliorarne il recupero. Sia la dimensione dei blocchi, sia il numero di repliche, possono essere configurati dall'utente. Come accennnato precedentemente, ogni cluster Hadoop presenta due tipologie di nodi, che operano secondo il pattern *master-slave*.

Architettura

Il file system (HDFS) presenta un'architettura (vedi *figura 1.1*) in cui un nodo *master* identifica il *NameNode* e un certo numero di nodi *slave* identificano i *DataNode*.

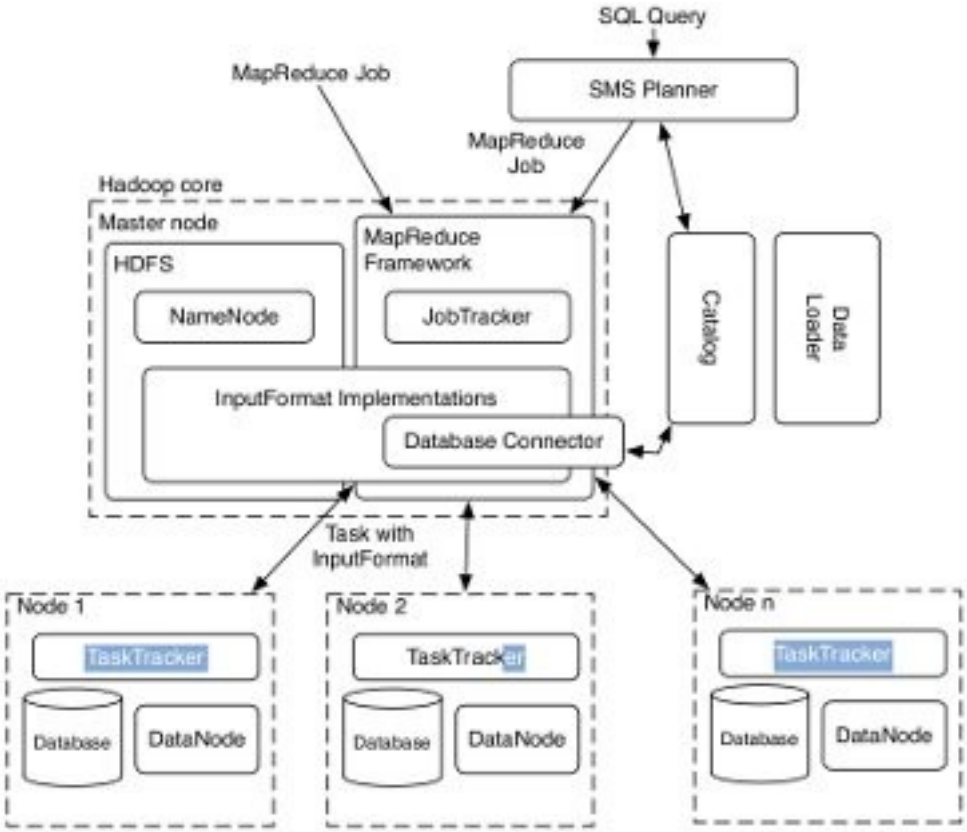


Figura 1.1 Architettura HDFS: il nodo master attribuisce i compiti ai nodi slave

- *Job Tracker*: è il servizio che prende in input il job da eseguire e lo suddivide tra i nodi (*task tracker*) map, e i nodi (*task tracker*) reduce, i quali svolgono le operazioni: la scelta avviene o tra i nodi più vicini o tra quelli che contengono i dati richiesti. Una volta che i task tracker hanno svolto il compito, il job tracker fornirà il risultato.
- *Task Tracker* (*map*): sono i nodi scelti per l'operazione di map. Questi nodi si occupano di raccogliere i dati e di generare una coppia chiave – valore che poi verrà inviata ai task reduce.
- *Task Tracker* (*reduce*): sono i nodi che ricevono le coppie chiave-valore e, dopo averle ordinate per chiave, le spediscono allo script reduce che genera il risultato.
- *Namenode* (*master*): Mantiene la struttura della directory di tutti i file presenti nel *file system* e tiene traccia della macchina in cui essi sono memorizzati.

ATTENZIONE: non memorizza i dati, ma solo le strutture. Le applicazioni client possono dialogare con il *NameNode* quando vogliono informazioni per individuare un file, o quando vogliono aggiungere / copiare / spostare / cancellare un file. Il *NameNode* risponde alle richieste restituendo un elenco di server *DataNode* dove si trovano memorizzati i dati richiesti. Quando il *NameNode* cade, cade tutto il file system. Per questo c'è un *SecondaryNameNode* opzionale che può essere ospitato su una macchina separata, il quale tuttavia non fornisce alcuna ridondanza reale.

- *Datanode* (*slave*): memorizza i dati nel *FileSystem*. Un *filesystem* funzionale ha più *DataNode* che contengono dati replicati. Le *applicazioni client* possono comunicare direttamente con un *DataNode*, una volta che il *NameNode* ha fornito la posizione dei dati. Allo stesso modo, le operazioni di *MapReduce* affidate alle istanze *TaskTracker* nei pressi di un *DataNode*, possono parlare direttamente ai *DataNode* per accedere ai file. Istanze *TaskTracker* possono, anzi devono, essere "schierate" sugli stessi server delle istanze *DataNode*, in modo che le operazioni di *MapReduce* vengano eseguite vicino ai dati. Le istanze *DataNode* possono parlare tra loro, specialmente quando si tratta di dati replicati. Di solito non c'è bisogno di utilizzare l'archiviazione *RAID* per i dati *DataNode*, in quanto i dati sono stati progettati per essere replicati su più server, piuttosto che avere più dischi sullo stesso server.

I *file system distribuiti* rappresentano una possibile soluzione alla gestione e memorizzazione dei BigData, ma non è l'unica; negli ultimi anni l'esigenza di gestire i BigData ha portato alla nascita di un nuovo movimento che prende il nome di *NoSQL*. L'espressione NoSql, che sta per "*Not Only SQL*" o "*Not Relational*", non è contraria all'utilizzo del modello relazionale, ma fa riferimento a tutti quei database che si discostano dalle regole che caratterizzano i database ralazionali (RDBMS), strutturati intorno al concetto matematico di relazione o tabella. Al crescere del volume dei dati, i problemi di scalabilità e i costi legati ai database relazionali sono soltanto una parte degli svantaggi: molto spesso anche la mancanza di una struttura fissa rappresenta una problematica da non sottovalutare. I database NoSql, a differenza di quelli costruiti basandosi sul modello relazionale, non presuppongono una struttura rigida o uno schema dove vengono descritte le proprietà che i dati dovranno avere e le relazioni tra loro. I database NoSql puntano sulla flessibilità e sulla capacità di gestire i dati

con strutture difficilmente rappresentabili in formati tabellari. La natura distribuita dei database NoSql rende non applicabili le proprietà ACID (*Atomicity, Consistency, Isolation, e Durability*), che invece caratterizzano i database tradizionali: questa è una diretta conseguenza del teorema CAP (*Consistency, Availability, Partition-tolerance*), il quale afferma l'impossibilità per un sistema distribuito di fornire simultaneamente consistenza, disponibilità e tolleranza di partizione, ma la possibilità di soddisfarne contemporaneamente al massimo due di esse. Le tre proprietà appena citate sono definite nel seguente modo:

- *Consistenza*: a seguito di una modifica sui dati, ciascun nodo del sistema dovrà visualizzare la stessa versione.
- *Disponibilità*: ogni nodo di un sistema distribuito deve sempre rispondere alla richiesta di dati, e in caso di "caduta", le informazioni devono poter essere reperibili presso un altro nodo.
- *Tolleranza di partizione*: capacità di un sistema di essere tollerante all'aggiunta o alla rimozione di un nodo del sistema.

Database NoSQL

I database NoSQL pertanto non offrono garanzie ACID, tuttavia sfruttano proprietà più flessibili e adatte al contesto NoSQL, nello specifico quelle del modello BASE (*Basically available, Soft state, Eventual consistency*), secondo cui il sistema deve essere sempre disponibile, e la consistenza, che non viene garantita ad ogni istante, deve essere verificata al termine delle operazioni/esecuzioni. Il modello BASE potrebbe non risultare adatto per ogni situazione, ma risulta essere un'alternativa flessibile al modello ACID.

La caratteristica dei database NoSql di poter scalare orizzontalmente consente di fare a meno di hardware performante ad alto costo, sostituendolo invece con commodity hardware. Infatti, le dimensioni di un cluster su cui è installato un database NoSql possono essere aumentate o diminuite, aggiungendo o rimuovendo nodi a piacere, senza particolari problematiche di gestione e in maniera trasparente all'utente, realizzando così una piena scalabilità orizzontale a costi moderati.

Come tutti i modelli, anche quello NoSql, oltre ai vantaggi, presenta svantaggi che devono essere tenuti in considerazione nel momento in cui si sceglie il modello da utilizzare in un

determinato contesto. Da un lato, i tempi di risposta all'aumentare della mole dei dati risultano essere più performanti rispetto a quelli riscontrati con i database relazionali, grazie all'assenza delle costose operazioni di join sui dati che caratterizzano gli ambienti SQL. Le prestazioni ottenute in lettura però, vanno a discapito della replicazione delle informazioni, anche se in realtà, i costi sempre meno proibitivi dei sistemi di storage rendono questo svantaggio trascurabile. Dall'altro lato, uno svantaggio è la mancanza di uno standard universale, come per esempio SQL, che caratterizza i database relazionali: ogni database appartenente al mondo NoSql ha a disposizione un insieme di API, metodi di storing e accesso ai dati che differiscono a seconda dell'implementazione che si considera. Non esiste un'unica tipologia di implementazione, infatti i database NoSql vengono classificati sulla base di come i dati sono memorizzati.

Modelli Dati NoSQL

A seconda di come vengono memorizzati i dati è possibile individuare diversi *modelli dati* NoSql. I principali sono:

- *Column-oriented*: i column-oriented database, diversamente dai tradizionali RDBMS che memorizzano i dati per riga, sfruttano la memorizzazione dei dati per colonna. Può essere pensato come un insieme di coppie chiave/valore, dove il dato viene identificato mediante la chiave primaria, detta anche *row-key*. Queste coppie vengono memorizzate in maniera ordinata sulla base della chiave di riga che le identifica. L'organizzazione dei dati per colonna, invece che per riga, permette di evitare il fenomeno della sparsità dei dati, ovvero sprechi di spazio nel momento in cui un determinato valore non esiste per una determinata colonna.
- *Key/value*: i Key/Value store rappresentano una tipologia di database NoSql che si basa sul concetto di *associative array*, implementati attraverso *HashMap*: la chiave è un valore univoco con il quale è possibile identificare e ricercare i valori nel database, accedendovi direttamente. La tipologia di memorizzazione adottata dai key/value stores garantisce tempi di esecuzione costanti per tutte le operazioni applicabili sui dati: *add, remove, modify e find*. Fra le principali, la tipologia NoSql *Key/Value* è la più semplice.

- *Document-oriented*: i database document-oriented rappresentano una specializzazione dei key/value store: è un insieme strutturato di coppie chiave/valore, spesso organizzati in formato JSON o XML che non pone vincoli allo schema dei documenti garantendo così una grande flessibilità in situazioni in cui, per loro natura, i dati hanno una struttura variabile. I documenti sono gestiti nel loro insieme, evitando di suddividere tali documenti in base alla struttura delle coppie chiave/valore. Questa tipologia di database è particolarmente adatta alla memorizzazione di tipologie di dati complessi ed eterogenei.
- *Graph*: i graph database rappresentano una particolare categoria di database NoSql, in cui le "relazioni" vengono rappresentate come grafi. Il concetto matematico di grafo consiste in un insieme di elementi detti *nodi* collegati fra loro da *archi*. Nell'ambito informatico il grafo rappresenta una struttura dati costituita da un insieme finito di coppie ordinate di oggetti. Le strutture a grafo si prestano molto bene per la rappresentazione di determinati dati semistrutturati e altamente interconnessi come, ad esempio, i dati dei social network e del Web.

2. Il sistema HIVE

Nel contesto dei BigData, molto spesso si parla di *BigData Analitycs*: insieme di processi che in grado di esaminare grandi quantità di dati, caratterizzati da una struttura non fissa, al fine di identificare patterns, correlazioni o andamenti nascosti nella moltitudine di dati grezzi. Gli aspetti che stanno alla base degli strumenti di business analitycs sono varietà, velocità e volume, non contemplati negli strumenti tradizionali, ma fondamentali per le performance dei processi in ambito BigData. Molti sono gli strumenti che possono essere applicati, ognuno con le proprie caratteristiche e il proprio obiettivo. In base alla piattaforma adottata in fase di gestione dei BigData e a seconda di cosa si intende esaminare, individuare o valutare si sceglierà lo strumento di analitycs che meglio si addice.
In questa tesi come piattaforma si è scelto di utlizzare *Hadoop,* e come strumento di analisi *HIVE.*

2.1 Architettura

Il software data warehouse *Apache Hive* ™ facilita l'esecuzione di query e la gestione di grandi quantità di dati residenti in storage distribuito (ad esempio Hadoop). Hive fornisce un meccanismo per dotare i dati di una struttura (simile alle tabelle dei database relazionali) e interrogarli utilizzando un linguaggio *SQL-like* chiamato *HiveQL*. Allo stesso tempo, questo linguaggio permette anche l'utilizzo del framework *MapReduce* per l'esecuzione delle query.

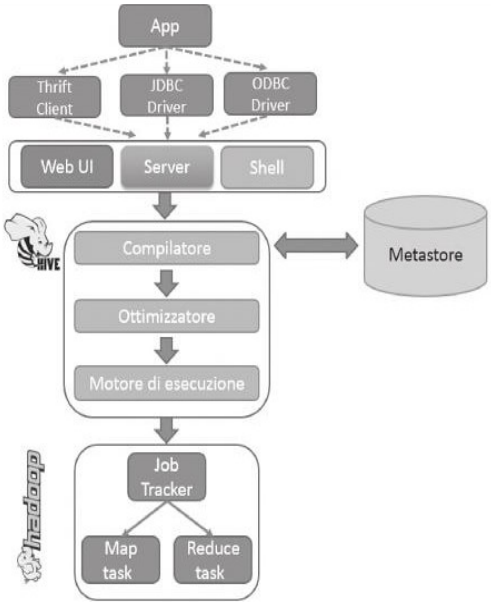


figura 2.1 Hive come metastore untilizza un RDBMS e come motore d'esecuzione astrae MapReduce

Nella figura 2.1 viene illustrata l'architettura di Hive. Hive ha 3 punti di accesso:

- *Intefaccia web*: consente di navigare il modello dati, visualizzando i *database*, le *tabelle* e i loro *metadati*; permette inoltre di creare *nuove sessioni*, dove è possibile eseguire e monitorare query utilizzando HiveQL.
- *Shell dei comandi*: interfaccia *a riga di comando* tramite la quale è possibile impostare i parametri di configurazione (ad esempio stabilire il numero di task *mapper* o *reducer* durante i job *MapReduce*) e lanciare query sfruttando il linguaggio *Hive*QL, i comandi HDFS o quelli della SHELL del sistema operativo che si sta utilizzando.
- *Server*: le applicazioni possono invece comunicare con il server, utilizzando il *client thrift* (un software che consente di definire i tipi di dati e interfacce di servizio in un semplice file di definizione. Prendendo quel file come input, il compilatore genera il codice da utilizzare per creare facilmente i client dei server che comunicano senza problemi su linguaggi di programmazione RPC), i driver JDBC (*Java DataBase Connectivity*, consentono l'accesso e la gestione della persistenza dei dati da qualsiasi programma scritto in linguaggio Java indipendentemente dal tipo di DBMS utilizzato), oppure tramire i driver ODBC (*Open DataBase Connectivity*, che utilizzano un'API standard per la connessione dal client al DBMS. Questa API è indipendente dai linguaggi di programmazione, dal database utilizzato e dal sistema operativo). Una volta connessi al server hive sarà possibile *creare oggetti*, per l'esecuzione di query.

Quando si lancia il comando per eseguire una query:

- *Compiler*: analizza la query, ne effettua l'analisi semantica e genera un piano di esecuzione, aiutandosi con le informazioni riguardanti le partizioni e tabelle dei metadati fornite dal metastore.
- *Metastore*: memorizza tutte le informazioni riguardanti la struttura delle tabelle e partizioni, le colonne e il tipo, la serializzazione e deserializzazione necessaria per scrivere e leggere i dati e il loro file corrispondente nel file system (HDFS). Fisicamente il *metastore* è un *motore database* realizzato in JAVA, chiamato *Derby*, il quale però ha il limite di non funzionare in un ambiente *multi-utente*; per questo è

possibile configurare Hive in modo tale che il metastore utilizzi un altro motore database, come ad esempio *MYSQL* o *Oracle*.

- *Execution Engine*: esegue il piano di esecuzione creato dal compiler. Il piano è suddiviso in più step, e l'Execution Engine si occupa delle relazioni tra le diverse fasi e la loro esecuzione. L'output quindi non sarà il risultato della query, bensì un grafo (aciclico o direzionato) di job *MapReduce.*

Modello Dati

Il modello dati di Hive prevede una gerarchia di oggetti simile a quella dei RDMS:

- *Database*: è il *namespace* e gli oggetti al suo interno devono avere nomi *univoci.*
- *Tabella:* è un contenitore di dati omogenei, ossia dati con lo *stesso schema. Ogni riga della tabella ha lo stesso numero di colonne.*
- *Partizione*: ogni tabella può avere più chiavi, dette chiavi di *partizionamento,* che determinano come sono salvati i dati; esiste una partizione per ogni chiave. Le chiavi di partizionamento non fanno parte della tabella, ma sono definite e create in fase di caricamento dei dati.
- *Bucket***:** i dati di ciascuna partizione possono essere suddivisi in unità più piccole dette bucket (*o cluster*) utilizzando una funzione di *hashing* su una colonna della tabella.

Oltre ai tipi di dato *primitivi*, simili a quelli di un linguaggio di programmazione ad alto livello, esistono dati complessi creati utilizzando 3 contenitori:

- *Array*: liste di elementi con lo stesso tipo di dato.
- *Map*: coppie chiave/valore.
- *Struct*: insieme di campi che possono avere tipi di dato diversi tra loro.

2.2 Linguaggio HiveQL

HiveQL (*Query Language*)[5] è il linguaggio di interrogazione dati usato da Hive. La sinstassi è simile a SQL (S*tructure Query Laguage*) con la differenza che HiveQL e quindi Hive è progettato per la creazione di job in grado di gestire grandi moli di dati, mentre SQL e i

RDBMS sono ottimi per l'esecuzione di attività transazionali.
HiveQL come SQL fa uso sia di operatori (logici, relazionali, aritmetici e operatori utilizzabili sui tipi di dati complessi), sia di funzioni *built-in* che aiutano l'utente nella costruzioni di analisi più sofisticate.
La particolarità di Hive consiste nella possibilità di descrivere i dati presenti in HDFS attraverso un *layer di metadati*:

- `show databases`: è possibile visualizzare l'elenco di tutti i database creati.
- `use` *`nome_database`*: è il comando che permette di selezionare il database sul quale si ha intenzione di lavorare.
- `describe database` *`nome_database`*: consente di ottenere la posizione del database all'interno di HDFS, aggiungendo la clausola `extended` si ottengono informazioni sulle sue proprietà (ad esempio, chi ha creato il database e quando).
- `describe table` *`nome_tabella`*: vengono visualizzati i campi che compongono la tabella e eventuali chiavi di partizionamento, anche in questo caso aggiungendo la clausola `extended` si ottengono informazioni aggiuntive, come il numero di righe, il tipo di dato di ogni campo, il formato di memorizzazione e il tipo di compressione dei dati.
- `show tables`: visualizza tutte le tabelle memorizzate all'interno del database selezionato.
- `show index on` *`nome_tabella`*: restituisce l'elenco degli indici creati su ciascuna colonna della tabella scelta.

Per quanto riguarda il modello dati invece:

- *database*:

```
create database nome_database
location
with dbproperties
```

il comando *create* crea il database che fungerà da *namespace*, *location* specifica il percorso su cui Hive salverà fisicamente i dati e infine *dbproperties* permette di aggiungere proprietà ai fini descrittivi.

- *Tabella*

```
create [External]table [if not exist]
nome_tabella (nome_campo tipo_dato)
partitioned by (nome_colonna tipo_dato)
clustered by (colonna1,colonna2)
sorted by (colonna) into num buckets
row format formato_righe
stored as formato_file
location
```

come per il database, anche per le tabelle si usa il comando *create* per assegnare un nome all'oggetto e i campi che lo compongono; la clausola *external* e consente di creare la tabella al di fuori del percorso di default. In questo modo si può specificare un percorso che punti a un file già esistente. La clausola *if not exist* permette la creazione della tabella solo se non ne esiste una con lo stesso nome. È possibile commentare la tabella o le singole colonne tramite il comando *comment*. Per definire le chiavi di partizionamento si utilizza la parola chiave *partitioned by*, mentre il con il comando *clustered by... into buckets* è possibile suddividere ulteriormente i dati in un certo *numero* di buckets. *Row format* invece serve per impostare il formato con cui le righe sono salvate: può essere *Delimited e* in questo caso bisognerà stabilire il carattere *terminatore di riga* (fields terminated by *char*), oppure SerDe (utilizzato di default se nulla è stato specificato o il formato è *Delimited*) che è il serializzatore/deserializzatore di Hive. Utilizzando la parola chiave *stored as* si specifica il formato (*textfile, sequence file* che possono essere compressi). Tblproperties permette di associare coppie chiave/valore alla tabella per scopi descrittivi. Infine con il come LIKE è possibile creare una tabella con le stesse caratteristiche di una già esistente es: create table *nome_tabella* like *tabella_esistente*

- *partizione*

```
load data local inpath percorso_dati
overwrite into table nome_tabella
partition
```

le partizioni possono essere definite in fase di *creazione* della tabella, o anche in fase di caricamento dati: con il comando `load data inpath` si possono caricare i dati che si trovano all'indirizzo specificato nella tabella desiderata. La parola chiave `partition` consenste di creare una partizione su una colonna (si può anche specificare una condizione es mese ='Feb'). La parola chiave `local,` serve per indicare che il *percorso_dati* si riferisce al filesystem locale e non a HDFS, mentre `overwrite` sovrascrive i dati caricati a quelli che erano già presenti, invece di aggiungerli in coda.

L'operazione inversa del caricamento dati, è l'esportazione della tabella:

```
insert overwrite local directory path_HDFS
select campi from tabella
```

con questa operazione, si estraggono i dati dalla tabella e si salvano su HDFS all'inidirizzo path_HDFS. La clausola `local` se omessa provoca lo spostamento dei dati, altrimenti la copia.

L'elaborazione dei dati in HiveQL è molto simile a SQL (le clausole `select, from, where, join, group by e having` hanno la stessa sintassi).

Una differenza importante riguarda l'ordinamento, in Hive esistono due tipi di ordinamento:

- `ORDER BY`: l'ordinamento avviene sul risultato finale (come per SQL), attraverso un unico task reducer.
- `SORTED BY`: l'ordinamento è parziale, infatti viene eseguito dai singoli reducer sul proprio set di dati.

Hive utilizza *MapReduce* per l'esecuzione della maggior parte dell query. In alcuni casi però, le query vengono eseguite in *local mode*, ossia senza ricorrere ai job MapReduce; questo accade nel caso in cui si richiede il contenuto di una tabella o partizione senza effettuare trasformazioni ai dati o eseguire operazioni di aggregazioni.

Lavorando con MapReduce, HiveQL permette di controllare come l'output dei *mapper* sia distrubuito tra i vari *reducer:*
l'istruzione `DISTRIBUTE BY`, associata con `SORTED BY` consente di determinare un criterio con cui le righe esaminate vengono assegnate ai reducer; se la colonna su cui avviene la distribuzione è la stessa sulla quale viene effettuato l'ordinamento allora queste istruzioni possono essere sostituite dall'istruzione `CLUSTER BY`.
HiveQL fornisce alcune funzionalità avanzate, tra le quali, la funzione `EXPLODE` abbinata a `LATERAL VIEW` e la costruzione di indici (per quanto riguarda le viste, il loro utilizzo è simile a SQL). La funzione *`explode`* ritorna i valori di un array sottoforma di risultato tabellare; questa funzione abbinata alla funzione *`lateral view`* (che definisce una specie di join tra una tabella e un risultato tabellare) permette in caso di presenza di un campo di tipo *array,* la creazione di una tabella dove ci saranno più righe, una per ciascun elemento.
Esempio: una tabella ha campo "nome" "cognome" "numeri" (numeri è un array di numero di telefono). Con la funzione *explode* e *lateral view* il risultato della select sarà una tabella dove ci saranno tante righe con lo stesso "nome" e "cognome", quante sono i numeri di telefono associati a quei valori.
In Hive è possibile creare indici, anche se rispetto ai database relazionali essi offrono funzionalità limitate. La struttura dell'indice è salvata su una tabella diversa specificata con l'istruzione `in table`. Per colonne che contengono pochi valori univoci è possibile creare indici bitmap, sostituendo il nome dell'handler:

```
Create index nome_indice
on   table     nome_tabella   (nome_colonna)As
'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'
idxproperties
in table nome_tabella
Partitioned by (nome_colonne)
Comment
```

HiveQL è un sottoinsieme di SQL-92 per cui non garantisce tutte le funzionalità di SQL:

- Hive QL non supporta la selezione da tabelle diverse. Quindi se una query SQL prevede una selezione tra più tabelle, questa va riscritta sfruttando i join in modo da creare una "tabellona" composta dalle tabelle sui cui la query originale faceva una selezione.
- Le *subquery* in Hive QL restituiscono solo il nome dei campi ma non i valori. Anche in questo caso bisogna riscrivere la query, spezzare la subquery in due query separate e salvare il risultato in una tabella di appoggio.
- Attualmente, HiveQL non supporta i costrutti `"in", "not in","exist","not exist".`
- Hive QL non supporta la clausola `"or"` nei join.
- Non sono supportate le tabelle temporanee, che si cancellano automaticamente alla fine della sessione.
- Hive QL non supporta i tipi di dato *incrementali,* usati molto spesso dai database relazionali per definire le chiavi *primarie.*

Hive rimane comunque uno degli strumenti più adatti a svolgere analisi sui dati grazie alla presenza di HiveQL, che essendo piuttosto simile a SQL non richiede tempi di apprendimento elevati, soprattutto per chi ha un po' di familiarità con i database relazionali.
Tuttavia non può essere utilizzato come strumento per interrogazioni *real-time*, visto che i tempi di risposta rispetto ai database relazionali sono molto più lunghi, decine di secondi e addirittura minuti.
Resta comunque il fatto che grazie ad Hadoop, Hive raggiunge gradi di flessibilità e scalabilità difficilmente replicabili dai sistemi relazionali.
Dal punto di vista delle performance a partire dalla versione 0.11, rilasciata il 15 Maggio 2013 ci sono stati evidenti miglioramenti soprattutto grazie al tentativo di ottimizzare la gestione dei file.
Oltre ai formati di testo e sequenziali, ora è possibile scegliere tra i formati ORC e PARQUET, un formato file sviluppato alla fine del 2013 da *twitter* e *cloudera*.

2.3 File Parquet

Hive permette di decidere quale formato utilizzare per memorizzare i dati.

Se non viene specificato niente al momento della creazione delle "tabelle", il formato standard è il formato AVRO, che memorizza i dati in sequenza. Prendiamo ad esempio la seguente tabella (figura 2.2):

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

Figura 2.2

se il formato di memorizzazione è quello standard (figura 2.3), allora i dati saranno memorizzati *per riga*:

Figura 2.3 formato AVRO

Questo tipo di memorizzazione è ottimo per i database *relazionali,* ma nel campo dei *BigData,* dove spesso le informazioni vengono *filtrate per colonna,* in quanto uno degli obiettivi è la storicizzazione delle informazioni, l'ideale è memorizzare i dati per *colonna* (figura 2.4). Un gruppo di sviluppatori *twitter e cloudera* ha pertanto introdotto per hadoop un formato file *colonnare* [6] strutturato nel seguente modo:

figura 2.4 formato PARQUET

La memorizzazione per *colonna* porta i seguenti vantaggi:

- Miglior compressione dei dati *omogenei*, soprattutto se si tratta di moli imponenti di dati.
- Diminuiscono le operazioni di *input/output* in quanto è molto più semplice individuare e filtrare le colonne desiderate.
- I dati dello stesso tipo sono raggruppati e quindi si può optare per una codifica adeguata per ogni tipo di dato.

Il formato PARQUET è stato pensato anche per memorizzare con efficacia *le strutture dati*. Ogni campo ha tre attributi: una ripetizione, un tipo e un nome:

- *Il tipo* di un campo è un gruppo o un tipo primitivo (ad esempio, int, float, boolean, stringa).
- *La ripetizione* può essere uno dei seguenti tre casi:
  - *required:* esattamente 1
  - *optional:* 0 o 1
  - *repeated*: 0 o più occorrenze

Ad esempio, per una rubrica:

```
message AddressBook {
  required string owner;
  repeated string ownerPhoneNumbers;
  repeated group contacts {
    required string name;
    optional string phoneNumber;
  }
}
```

il tipo di dato struct, può essere rappresentato graficamente da un albero (vedi figura 2.5)

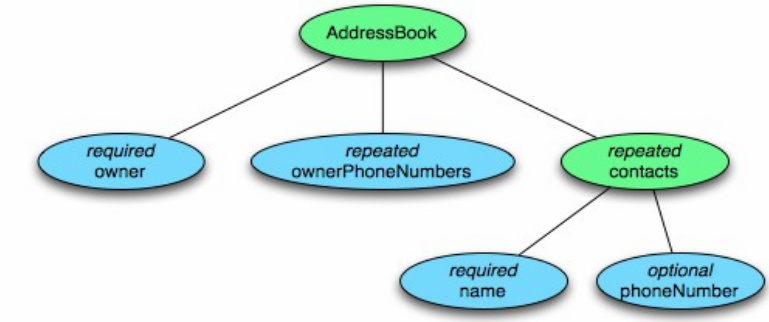


figura 2.5 i tipi di dato struct sono i nodi padre e i campi sono i figli

la struttura del record viene *definita da* due campi interi: *definition level,* e *repetition level:*

- *Definition Level:* il valore indica a che livello il campo è *nullo*, *se un campo* è definito allora *avrà valore* 1, altrimenti valore 0. In caso di record *annidati* il valore massimo di *definition level* sarà dato dalla *profondità* dell'albero.

Esempio:

```
message ExampleDefinitionLevel
{    optional group A {
          optional group b {
               optional group c;
}
```

Value	Definition Level
a: null	0
a: { b: null }	1
a: { b: { c: null } }	2
a: { b: { c: "foo" } }	3 (actually defined)

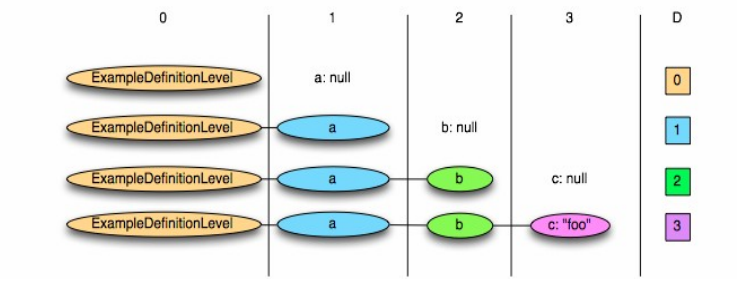


figura 2.6 determina il definition level per ogni campo e mostra graficamente le possibili combinazioni

I campi *required* sono sempre definiti e quindi *non è necessario* assegnare loro un *definition level* (figura 2.6).

Repetition Level: serve per stabilire quando *inizia* una nuova lista. Indica quando bisogna *iniziare* un nuovo elenco e a quale livello (figura 2.7).

Esempio:

Data: `[[a,b,c],[d,e,f,g]],[[h],[i,j]]`

```
{
      level1: {
            level2: a
            level2: b
            level2: c
      },
      level1: {
            level2: d
            level2: e
            level2: f
            level2: g
      }
}
{
      level1: {
            level2: h
      },
      level1: {
            level2: i
            level2: j
      }
}
```

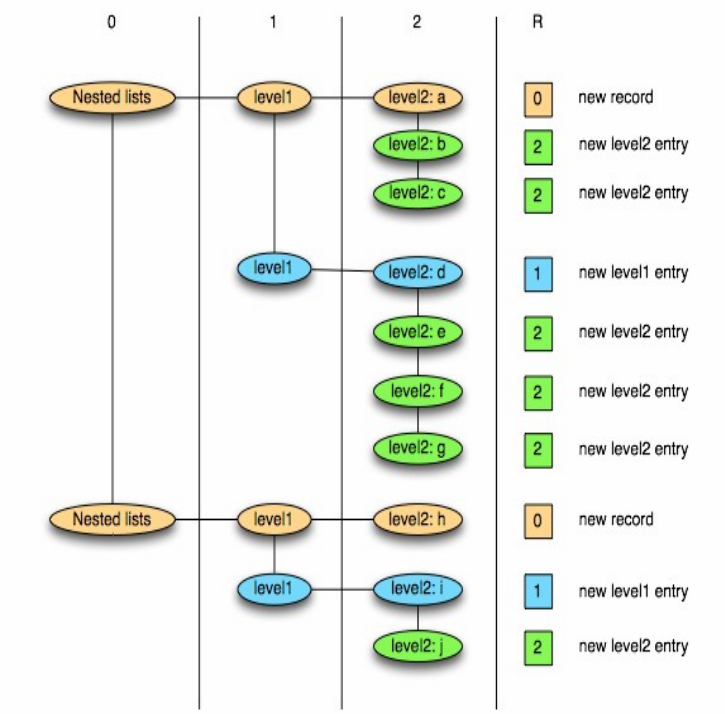


figura 2.7 rappresenta graficamente come grazie al repetition level si possa ricreare una lista

Se il *repetition level* è impostato a 0, significa che sta iniziando una nuova lista.

I campi *optional e required* non saranno mai ripetuti e quindi per essi può essere omessa questa informazione.

In questo modo, il formato Parquet è in grado di *memorizzare* strutture dati complesse garantendo la possibilità di ricostruire il record. È possibile combinare i vantaggi del *Definition* e *Repetition Level*, ma per quanto riguarda le analisi effettuate, utilizzando solo dati *primitivi,* il fattore più rilevante è la memorizzazione *colonnare;* infatti raccogliendo i tempi di esecuzione delle query è stato notato che aumentando la dimensione del database, e quindi la mole di dati coinvolti, la *formattazione* PARQUET risulta più *efficace* di quella *standard*.

2.4 Piani di esecuzione

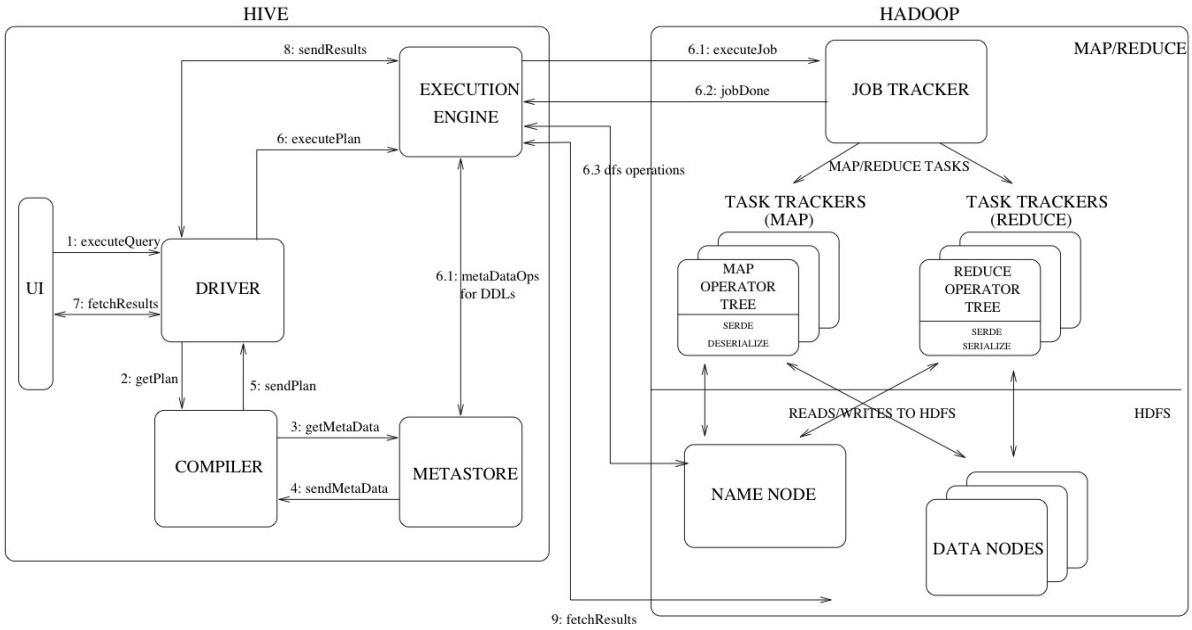


figura 2.8 percorso che effettua una query per essere eseguita

come già accennato, il compito di Hive è quello di fornire un albero di job MapReduce che poi verranno eseguiti da Hadoop[7] come mostrato in figura 2.8:

- L'*interfaccia utente* (UI) richiama l'interfaccia driver (punto 1 nella figura 2.8).
- Il *driver* crea una sessione per la query e invia la richiesta al compilatore di generare un piano di esecuzione (punto 2).
- Il *compilatore* ottiene i metadati necessari dal metastore (punti 3 e 4). Questi metadati vengono utilizzato per l'analisi semantica delle espressioni nella struttura della query nonché per "scremare i dati" in base alle partizioni e predicati specificati dalla query.
- Il piano generato dal compilatore (punto 5) è un DAG di stadi e ogni fase può essere un job MapReduce, una operazione di metadati o un'operazione su HDFS. Per i job MapReduce, il piano contiene un *map operator tree* (alberi operatore che vengono eseguite sui mapper) e un *reduce operator tree* (per le operazioni che richiedono riduttori).
- Il *motore di esecuzione* presenta queste fasi ai componenti appropriati (i punti 6, 6.1, 6.2 e 6.3).

- In ogni job MapReduce il d*eserializzatore* associato alla tabella o a output intermedi, consente di leggere i file dai file HDFS e questi sono passati attraverso l'operator tree associato. Una volta generata l'uscita, il risultato scritto in un file HDFS temporaneo attraverso il serializzatore (questo accade nel mapper in caso l'operazione non ha bisogno di un reducer). I file temporanei vengono utilizzati per fornire dati per la successiva operazione MapReduce del piano. Per le operazioni DML il file temporaneo finale viene spostato su una tabella. Questo schema viene utilizzato per garantire che non ci sia una *dirty read* (essendo un'operazione atomica in HDFS).
- Per le query, il contenuto del file temporaneo viene letto dall'*Execution Engine* direttamente da HDFS nell'ambito della restituzione del risultato dal *Driver* all *Utente* (passi 7, 8 e 9).

MapReduce

il framework *MapReduce*[8] è stato pensato per permettere a qualsiasi utente di utilizzare un sistema distribuito su larga scala, come *Hadoop*, occupandosi automaticamente di:

- Partizionamento dei dati.
- Scheduling dei thread sulle varie macchine.
- Controllo e gestione dei fallimenti.
- Comunicazioni sia tra le macchine che i nodi.

Un job[9] *MapReduce* (illustrato in figura 2.9) è composto da:

- *map*: applica una funzione ai dati in input e li trasforma in coppie chiave-valore.
- Una funzione *shuffle*: raccoglie tutte le coppie generate dai *mapper* e produce un'insieme di liste ordinate per chiave, dove ogni lista è composta da tutti i valori associati alla stessa chiave.
- *reduce*: applica una funzione per ogni lista e produce un insieme di coppie chiave-valore come risultato in output che verrà poi memorizzato sul file system (ad esempio HDFS).

Ad esempio, consideriamo di voler sfruttare MapReduce per contare quante volte una parola è ripetuta all'interno di un documento:

- Funzione m*ap*: per ogni parola w presente nel documento genera una coppia chiave-valore del tipo (w,1).
- Funzione *shuffle*: raccoglie tutte le coppie (w,1) e per ogni chiave w (che corrisponde a una parola diversa), produce una lista (w,1,1..,1) ordinate per chiave dove il numero di "1" rappresenta quante volte quella parola appare nel documento.
- Funzione *reduce:* per ogni lista, somma tutti gli "1" e restituisce in output un insieme di coppie (w, n_istanze).

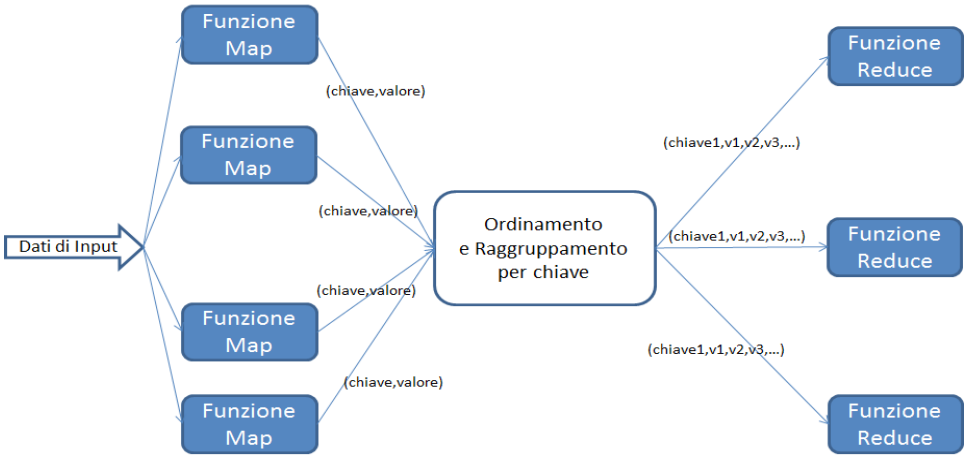


figura 2.9 la figura mostra il funzionamento di un job mapReduce. Le funzioni assegnate ai mapper e ai reducer possono essere definiti dall'utente così come il numero di mapper e reducer assegnati al job.

è possibile anche forzare l'esecuzione di mapReduce in *local mode*:

```
hive> SET mapred.job.tracker=local
```

in questo modo i job MapReduce invece di distribuirsi per tutto il cluster, sfruttano solo la macchina dalla quale è stata lanciata la query, risparmiando il tempo che si sarebbe impiegato per sottomettere il job alle altre macchine. Eseguire un job in local mode comporta però avere a disposizione un solo reducer e quindi in caso di grandi volumi di dati, l'esecuzione potrebbe risultare molto lenta.

Hive astrae la complessità di MapReduce[10], tramite HiveQL. Questo significa che l'utente finale dovrà preoccuparsi solo di eseguire la query in HiveQL: ci penserà il sistema a definire il piano di esecuzione per i job MapReduce (figura 2.10).

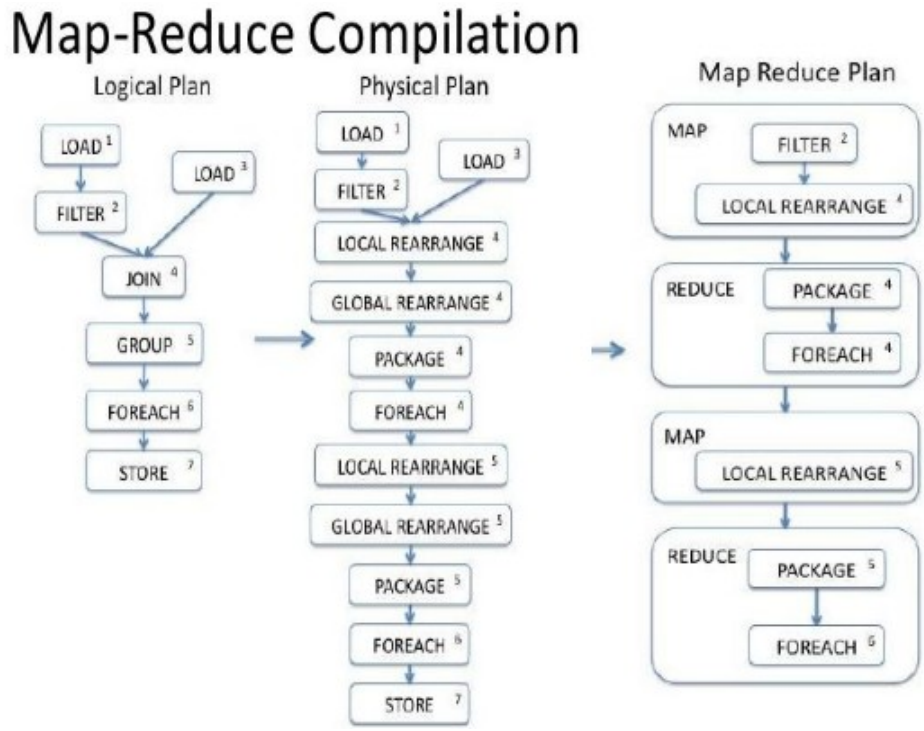


Figura 2.10 mostra i passi che trasformano un piano di esecuzione logico (una classica query scritta in un linguaggio ad alto livello come HiveQL) prima in un piano d'esecuzione fisico e infine in un piano d'esecuzione per i job MapReduce

Gli operatori del piano di esecuzione

Con il comando `Explain plain` all'inizio di una query è possibile ottenere il piano d'esecuzione della query stessa. Il piano d'esecuzione descrive come viene eseguita la query e quali operaori [11] vengono utilizzati. Qui di seguito ne descriveremo alcuni:

Tablescan (TS): indica l'accesso alla tabella; la scansione dei record in essa contenuti dipende dal formato scelto per i dati al momento della creazione della tabella stessa. Il formato *standard* prevede una scansione *sequenziale* per riga.

Filter (F): è un operazione che permettedi selezionare delle tuple dall'intero dataset in base a un predicato di selezione.

Projects (P)*:* alla fine di ogni *Stage MapReduce* la proiezione permette di selezionare solo le colonne di interesse, evitando di memorizzare l'intera tabella (specialmente dopo un join).

Hive riconosce i vari casi e ottimizza il join [13][13 bis]*:*

- *common join* (CJ) (*vedi figura 2.11):* ad ogni tabella vengono associati dei mapper che ne elaborano i dati che poi vengono inviati ad un unico reducer comune che produce il risultato del join: il mapper crea un file intermedio dove vengono indicate le coppie chiave/valore sulle qualei effettuare il join. Durante la fase di *shuffle* vengono riunite queste coppie e il reducer si occupa poi del join vero e proprio tra le tabelle.

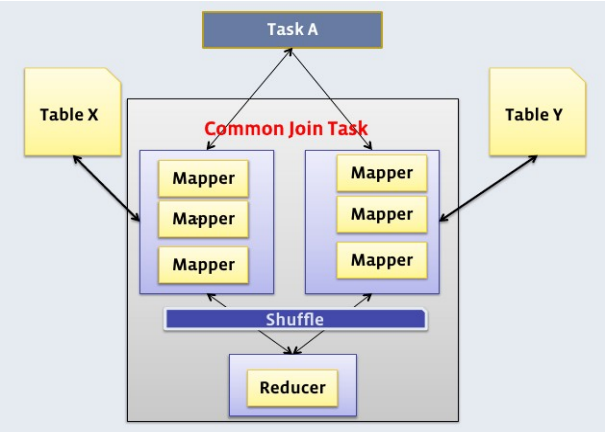


Figura 2.11 il CJ accede e processa entrambe le tabelle

- *Map join* (MJ) (vedi figura 2.12): l'obiettivo del MapJoin è quello di svolgere le operazioni di shuffle e reduce in un unica funzione mapper: riuscendo a caricare la tabella più piccola in memoria, i mapper possono infatti elaborare tutti i dati ed effettuare direttamente l'operazione di Join. Prima di effettuare il *task MapReduce* viene creato un *task locale* che ha il compito di leggere i dati della tabella in memoria e creare una tabella hash. Successivamente questa tabella viene caricata su HDFS e distribuita a tutti i mapper, i quali saranno in grado di eseguire il join.

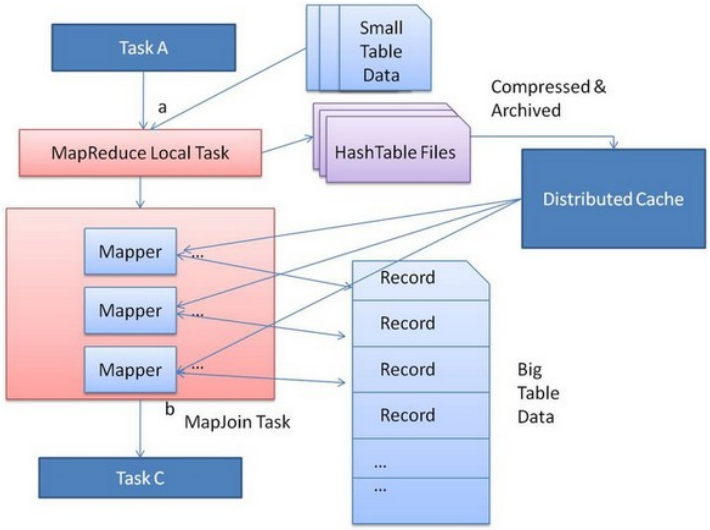


figura 2.12 MJ risparmia tempo caricando la tabella più piccola in memoria associando, grazie all'hash table precedentemente creata dal local task, le coppie do valori in modo da ottenere direttamente il join

Hive, a seconda del contesto, ottimizza i Join[15]. In caso di multi-join o quando non è possibile determinare la dimensione di una tabella Hive esegue un map-join e in caso di fallimento esegue uno stage di back-up utilizzando il *common join* (vedi figura 2.13)

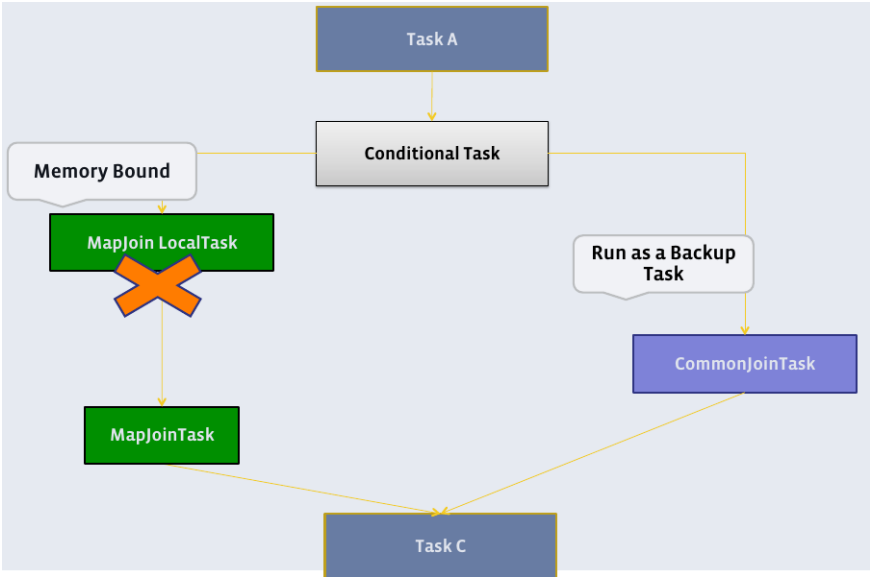


figura 2.13 MapReduce, in casi di multi-join prepara uno stage di backup in caso fallisse il MJ

Group By (GB): Hive effettua una funzione di aggregazione (es. SUM, COUNT, AVG) *locale e parziale* (merge partial) sui singoli container.

Shuffle (S): è una *funzione hash* che, avendo come chiave la colonna sulla quale è stata effettuata l'operazione di aggregazione, spedisce i risultati elaborati dai mapper ai reducer.

Ordinamento (O): è suddiviso in due fasi. La prima fase è quella di *sort* e avviene all'interno dei singoli container producendo un ordinamento parziale. La funzione di ordinamento globale (Order by) avviene eseguendo un job MapReduce apposta che raccoglie i risultati ottenuti dai mapper e li ordina come richiesto dalla query.

Vediamo nel dettaglio come vengono eseguite le query; a tal proposito sono state analizzate 3 query del benchmark TPC-H di dimensione 1GB(la query 1: pricing summary report, la query 3: shipping priority, query 6: forecasting revenue change). La scelta è ricaduta su queste 3 query perché anche se semplici, utilizzano tutte le principali parole chiavi, e soprattutto la sintassi non varia da SQL a HiveQL [16].

Il piano di esecuzione divide la query in *stage MapReduce,* composti da cicli MapReduce. Ogni stage è formato da due alberi: un *map operator tree (*in grado di processare le tuple di una particolare tabella oppure di mostrare il risultato di uno stage mapreduce precedente) e un *reduce operator tree* (si occupa di elaborare tutti i record raccolti dai vari mapper):

- La presenza di filtri non altera il piano di esecuzione, infatti ogni volta che si accede a una tabella essa viene scandita completamente ed eventualmente filtrata.
- Un singolo stage è in grado di effettuare le operazioni di *tablescan, filter, projects, group by, shuffle, join e ordinamento parziale (sort)*.
- Gli stage di back-up sono stage di supporto nel caso in cui lo stage “principale” dovesse fallire (vengono creati in caso di join multipli).
- In caso di *map join*, non sapendo ancora quale sia la tabella più piccola, il piano di esecuzione prevede due strade, una che esclude l'altra; in fase di esecuzione verrà scelta la strada che prevede l'elaborazione della tabella più piccola.

- Alcuni stage “consistono in sotto-stage”: questo avviene quando, all'interno dello stesso stage MapReduce, si accede a due tabelle distinte.
- l'operazione di ordinamento (group by) necessita di un proprio stage MapReduce, in quanto c'è bisogno di un ulteriore fase di lettura della tabella prodotta dagli stage precedenti (vedi query 1).
- Le operazioni di selezione (o proiezione) vengono effettuate alla fine di ogni stage, perché al termine di ognuno di essi viene salvata una tabella su disco.

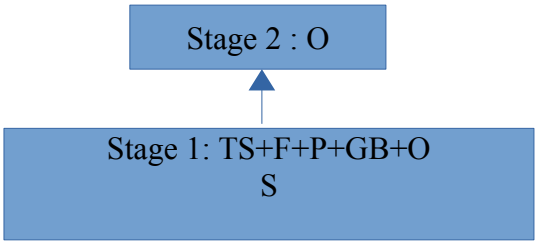
Query 1: pricing summary report

```
SELECT l_returnflag,
l_linestatus,sum(l_quantity),sum(l_extendedprice),sum(l_extende
dprice*(1l_discount)),
sum(l_extendedprice*(1l_discount)*(1+l_tax)),
avg(l_quantity),avg(l_extendedprice) as avg_price,
avg(l_discount),count(1)
FROM  lineitem
WHERE l_shipdate <= '1998-09-02'
GROUP BY l_returnflag,l_linestatus
ORDER BY l_returnflag, l_linestatus;
```

Per ottenere il piano di esecuzione della query si utilizza il comando `explain` prima della select. La prima informazione che ci viene fornita è l'elenco degli Stage MapReduce definito dall'executor engine di Hive, in questo caso:

STAGE DEPENDENCIES:
Stage-1 is a root stage
Stage -0 depends on stages: stage-1
Stage-2 depends on stages: stage-0
Stage-3 depends on stages: stage-2

Le dipendenze tra gli Stage possono essere viste come un *albero di esecuzione*, dove si parte dalla radice e *si sale* verso le foglie:



lo stage 0 e lo stage 3, e in generale gli ultimi due stage in ordine di esecuzione, sono stati omessi dal grafico perché non fanno parte del piano di esecuzione della query, ma riguardano l'invio del risultato da una tabella di supporto a quella di destinazione e la raccolta di statistiche.
A questo punto viene fornito il piano di esecuzione di ogni stage:

stage 1:
map operator tree: table scan(TS) su lineitem e filtro (F) sul predicato
proiezione (P) sui campi richiesti nella select
group by (GB) e ordina il proprio data set (O)
reduce operator tree: raccoglie i dati provenienti dai mapper (S)

stage 2:
map operator tree: esegue l'order by una volta raccolti tutti i risultati (O)

Query 6: forecasting revenue change

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

STAGE DEPENDENCIES:

Stage-1 is a root stage

Stage-0 depends on stages: stage-1

Stage-2 depends on stages: stage 0

grafo del piano di esecuzione:

Analisi in dettaglio degli Stage:

Stage 1:

map operator tree: table scan (TS) su *lineitem* + filtri (F)
proiezione (P) sui campi richiesti nella select
Group By (GB) e ordina il proprio dataset (O)

reduce operator tree: raccoglie i dati provenienti dai mapper (S)

Query 3: shipping priority

```
SELECT l_orderkey,sum(l_extendedprice * (1 - l_discount))
     as revenue,o_orderdate,o_shippriority
FROM  customer,orders,lineitem
WHERE c_mktsegment = 'BUILDING' and c_custkey = o_custkey
     and  l_orderkey = o_orderkey
     and o_orderdate < date '1995-03-15'and
     l_shipdate > date '1995-03-15'
GROUP BY   l_orderkey,o_orderdate,o_shippriority
ORDER BY  revenue desc,o_orderdate limit 10;
```

STAGE DEPENDENCIES:

Stage-13 is a root stage, consists of Stage-16, Stage-1

Stage-16 has a backup stage: Stage-1

Stage-12 depends on stages: Stage-16

Stage-10 depends on stages: Stage-1, Stage-12, consists of Stage-14, Stage-15, Stage-2

Stage-14 has a backup stage: Stage-2

Stage-8 depends on stages: Stage-14

Stage-3 depends on stages: Stage-2, Stage-8, Stage-9

Stage-4 depends on stages: Stage-3

Stage-0 depends on stages: Stage-4

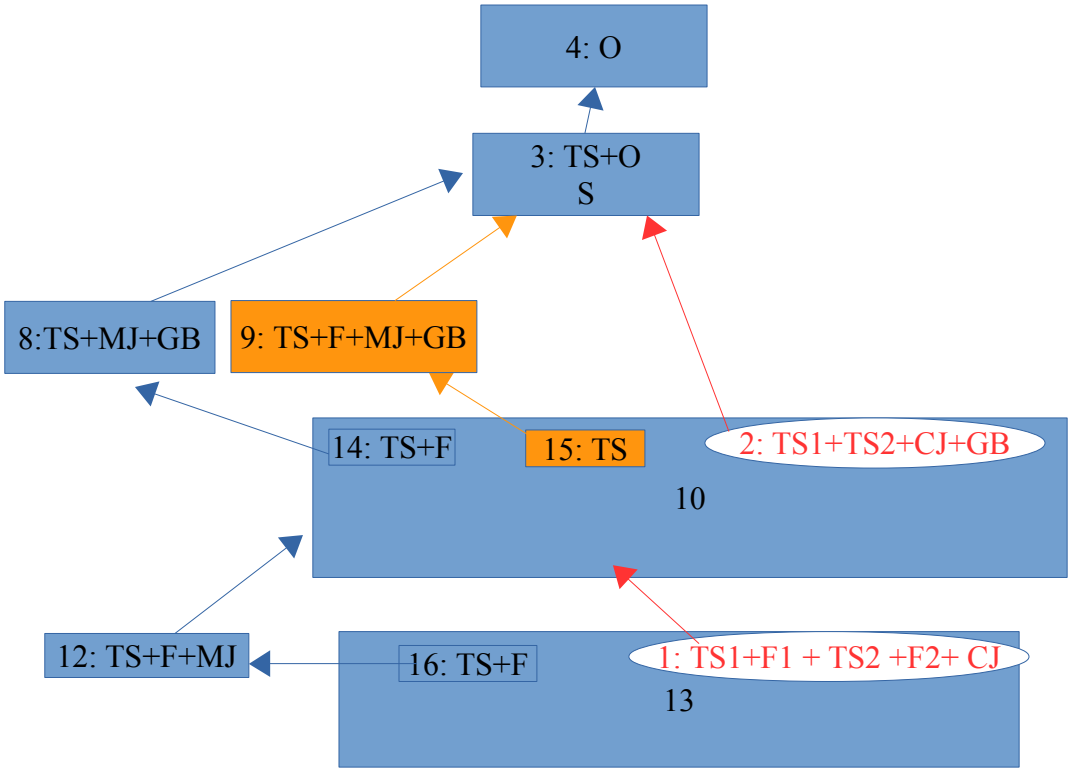
Stage-15 has a backup stage: Stage-2

Stage-9 depends on stages: Stage-15

Stage-2

Stage-1

grafo del piano di esecuzione:



Analisi in dettaglio degli Stage:

- stage 16 (local work)

  map local operator tree: tablescan customer (TS) + filtro (F)
- stage 12

  map operator tree: tablescan orders (TS) + filtro(F) poi join con customer(MJ)
- stage 14 (local work)

  map local operator tree: tablescan su lineitem (TS) + filtro(F)
- stage 8

  map operator tree: tablescan orders + customer (TS) e fa join (MJ) con lineitem Group by (GB)
- stage 3

  map operator tree: tablescan (TS) su risultato dei join + Ordina il proprio data set (O)

  reduce operator tree: raccoglie i risultati dai mapper (S)

- stage 4

  map operator tree: order by su tutto il dataset (O)

gli stage 15 e 9 sono ***alternativi*** agli stage 14 e 8 (*cambiano le combinazioni tra tabelle*):

- **stage 15** (local work): scansione (TS) su costumer + orders
- **stage 9**: scansione (TS) su lineitem + filtro (F) e fa join (MJ)con customer + orders e Group By (GB)

gli stage 1 e 2 sono di ***backup***, ossia vengono eseguiti solo in caso di fallimento degli stage "principali":

- **stage 1**: scansiona customer (TS1)+ filtri(F1), scansiona orders(TS2) + filtri(F2) e fa il join (CJ)
- **stage 2**: scansiona ordini + customer (TS1), scansiona lineitem(TS2) e poi fa il join(CJ) ed esegue il Group By (GB).

3.Test sperimentali

Una volta scelta la piattaforma per la gestione dei dati e individuato lo strumento per la loro manipolazione e interrogazione, è necessario effettuare dei test sperimentali per valutarne le prestazioni sulla macchina. In questo capitolo verranno valutate le prestazioni di Hive su piattaforma Hadoop utilizzando il Benchmark TPC-H.

3.1 Il benchmark

Si definisce Benchmark, un insieme di dati e test utilizzati come punto di riferimento per valutare le prestazioni e la qualità di una macchina. il Benchmark TPC-H[17] *versione 2.17.1* offre un dataset e un insieme di query studiati apposta per rappresentare un vasto settore industriale, pur mantenendo un sufficiente grado di facilità di implementazione. In particolare il benchmark tpch viene utilizzato per valutare un sistema in cui:

- Viene gestito un volume molto grande di dati.
- Vengono eseguite query con un alto grado di complessità.

TPC-H simula un sistema di supporto alle decisioni eseguendo un insieme di query su un database standard in condizioni controllate. Le query TPC-H:

- danno risposte a problemi critici riguardanti il settore;
- sono di gran lunga più complesse della maggior parte delle transazioni OLTP;
- fanno un ampio uso degli operatori e dei vincoli di selettività;
- generano un' intensa attività sul database lato server del sistema in esame;
- vengono eseguite su un database conforme a una specifica popolazione e alle esigenze di scala;
- sono implementate con i vincoli derivanti dal fatto di rimanere a stretto contatto e sincronizzate con un database di produzione on-line.

Il database TPC-H deve essere realizzato utilizzando un DBMS disponibile in commercio e le query eseguite tramite un'interfaccia SQL dinamica (non è specificato un particolare standard

SQL). TPC-H utilizza una terminologia e parametri che sono simili ad altri benchmark, originati dal TPC. Tale somiglianza non implica in alcun modo che i risultati TPC-H siano paragonabili ad altri benchmark. Gli unici risultati dei benchmark paragonabili a TPC-H sono altri TPC-H compatibili con la stessa versione.
Nonostante il fatto che questo benchmark offra una ricca rappresentazione di molti sistemi di supporto alle decisioni, TPC-H non rispecchia l'intera gamma di requisiti che un sistema di supporto alle decisioni potrebbe richiedere. Inoltre, i risultati che si ottengono dipendono fortemente da quanto strettamente TPC-H approssima l'applicazione finale. La performance relativa dei sistemi derivati da questo benchmark non è necessariamente adatta, o adeguatamente accurata, per altri carichi di lavoro o ambienti. I risultati dei benchmark sono altamente dipendenti dal carico di lavoro, dalle specifiche esigenze applicative, dalla progettazione dei sistemi e dalla loro implementazione. La prestazione del sistema varierà a causa di questi ed altri fattori, pertanto, TPC-H non deve essere usato come un sostituto di una specifica applicazione.
Lo scopo del benchmark TPC è quello di fornire rilevanti dati oggettivi sulle prestazioni per chi utilizzerà il sistema. Per raggiungere questo scopo, le specifiche benchmark TPC richiedono che i test siano eseguiti utilizzando sistemi, prodotti, tecnologie e prezzi che:

- siano generalmente disponibili per gli utenti;
- siano rilevanti per il segmento di mercato che il benchmark rappresenta (ad esempio, i benchmark TPC-H rappresentano un modello complesso, con un elevato volume di dati, in un ambiente di supporto decisionale);
- possano essere implementati da un numero significativo di utenti nel segmento di mercato rappresentato.

Le seguenti caratteristiche sono utilizzate come guida per giudicare se una particolare implementazione è un benchmark *special*. Non è necessario che ogni punto sia soddisfatto, la certezza assoluta, o certezza al di là di ogni ragionevole dubbio, non è richiesta per potere dare un giudizio su un tema così complesso. La domanda a cui si deve rispondere è: "Sulla base delle prove disponibili, la maggioranza di esse indica che questa implementazione è un benchmark *special* ?"

Per poter rispondere, è necessario basarsi sulle seguenti questioni:

a) L' implementazione è generalmente utilizzabile, documentata, e supportata?
b) L' implementazione ha restrizioni significative sul suo uso che ne limitino l'utilizzo al benchmark TPC?
c) La realizzazione o parte di essa è scarsamente integrata con il prodotto più grande?
d) l'implementazione sfrutta la natura limitata del benchmark TPC (ad esempio, query, mix di query, concorrenza e / o conflitti, requisiti di isolamento, ecc) in un modo che non sarebbe applicabile nell'ambiente che rappresenta?
e) L' uso di tale implementazione è scoraggiata dal fornitore? (Questo include la mancanza di promozione dell' implementazione in modo simile ad altri prodotti e tecnologie).
f) L' applicazione richiede conoscenze particolari da parte dell'utente finale, il programmatore, o l'amministratore di sistema?
g) L' implementazione (compresa la beta) è stata acquistata o è utilizzata da applicazioni nel settore del mercato che il benchmark rappresenta? Quanti sistemi l'hanno implementata? Quante utenti ne beneficiano? Se attualmente non è stata acquistata o utilizzata, vi è alcuna prova per indicare che in futuro verrà implementata da molti utenti?

I risultati dei benchmark TPC dovrebbero simulare accuratamente le prestazioni del sistema rappresentato. Pertanto, vi sono alcune linee guida che dovrebbero essere seguite quando si effettuano le misurazioni. L'approccio o la metodologia da utilizzare è esplicitamente descritto nella specifica o lasciato alla discrezione del *test sponsor*. Quando non sono descritte nelle specifiche, le metodologie e gli approcci utilizzati devono soddisfare i seguenti requisiti:

- L'approccio è una pratica di ingegneria accettata o è una pratica standard;
- L'approccio non migliora il risultato;
- Le attrezzature utilizzate per la misurazione dei risultati sono calibrate secondo gli standard di qualità stabiliti;
- Fedeltà nel segnalare eventuali anomalie nei risultati, anche se non specificati nei requisiti.

Le query TPC-H

Il TPC-H Benchmark è composto da una serie di query di business destinate ad esercitare funzionalità del sistema in maniera da rappresentare applicazioni di analisi di business complesse. Queste query descrivono un contesto realistico, raffigurante l'attività di un fornitore all'ingrosso. TPC-H non rappresenta l'attività di un particolare settore, ma piuttosto ogni industria che deve gestire la vendita, o distribuire un prodotto in tutto il mondo (ad esempio, il noleggio auto, distribuzione alimentare, i fornitori, ecc.). TPC-H non tenta di essere un modello di come creare un'applicazione di analisi di dati reali.
Lo scopo è quello di ridurre le diverse operazioni eseguibili dall'applicazione pur mantenendo le caratteristiche essenziali, come il livello di utilizzo del sistema e la complessità delle operazioni. Un gran numero di richieste di vario tipo e complessità deve essere eseguito per gestire in maniera completa un ambiente di *business analysis*. Molte di queste però non sono di interesse primario per l'analisi delle prestazioni a causa del tempo di esecuzione delle query, delle risorse di sistema richieste e della frequenza con cui vengono eseguite.
Le query che sono state selezionate per l'analisi delle prestazioni mostrano le seguenti caratteristiche:

- alto grado di complessità;
- accessi vari
- sono studiate *ad hoc*;
- Esaminano una grande percentuale dei dati disponibili;
- Sono tutte diverse le une dalle altre;
- Contengono parametri che possono essere cambiati ad ogni esecuzione.

E forniscono risposte alle seguenti categorie:

- I prezzi e le promozioni;
- gestione della fornitura e domanda;
- gestione delle entrate e degli utili;
- studio della soddisfazione del cliente;

- studio della quota di mercato;
- Gestione Spedizioni.

Altri benchmark TPC modellano l'operatività del contesto economico in cui le transazioni sono eseguite in tempo reale. Il benchmark TPC - H invece ne modella l' analisi: le tendenze vengono esaminate e i dati raffinati sono prodotti per fare da supporto alle decisioni.
In OLTP benchmark, i dati grezzi fluiscono da varie fonti e vengono memorizzati per un certo periodo di tempo. In TPC - H le funzioni di aggiornamento periodiche vengono eseguite in un database DSS, il cui contenuto è interrogato per conto o da vari decisori (Figura 3.1).

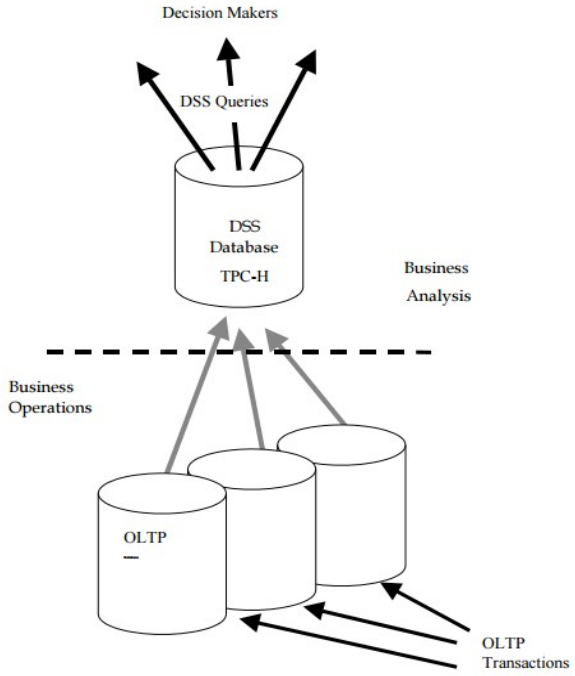


Figura 3.1 Il TPC-H Business Environment illustra il contesto imprenditoriale TPC-H e mette in evidenza le differenze di base tra TPC-H e altri benchmark TPC

TPC-H Schema

Il database TPC-H è composto da 8 entità in relazione tra loro, come mostrato nello schema qui sotto:

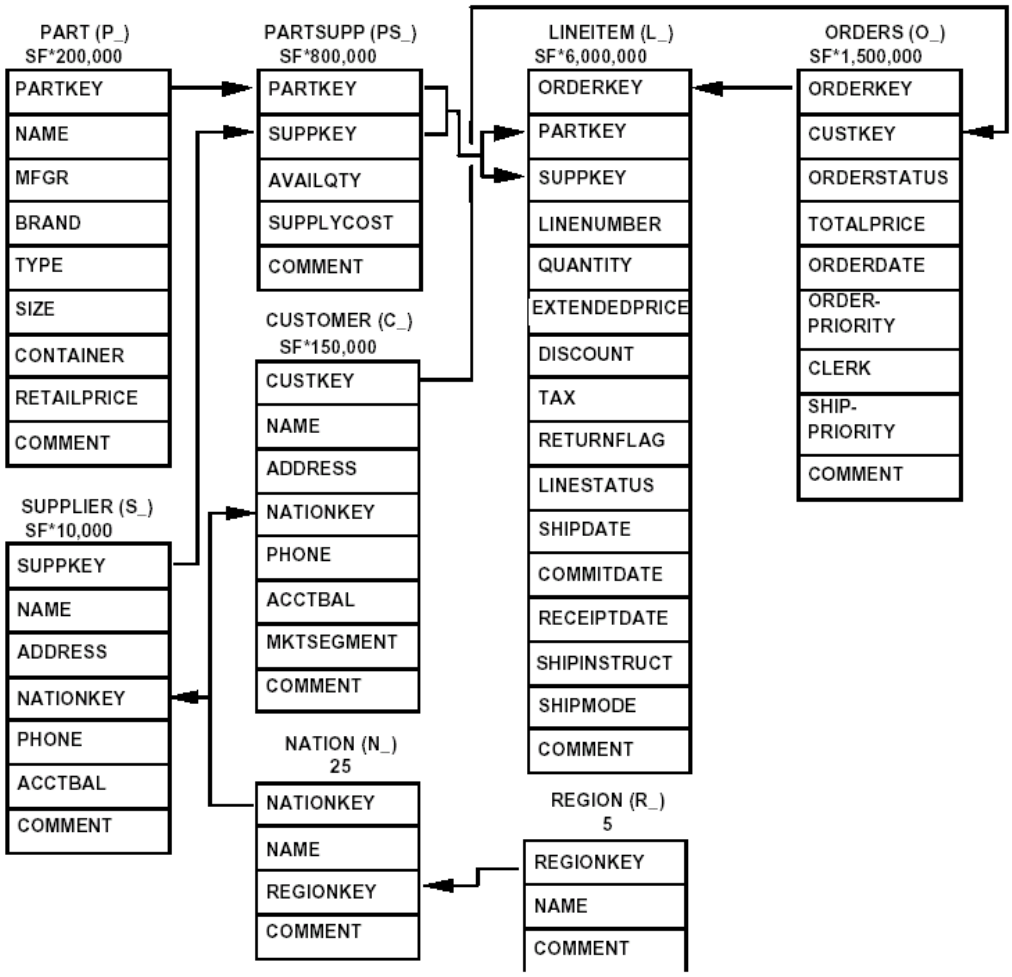


figura 3.2 schema TPC.H modello relazionale

- La lettera tra parentesi nel nome delle varie tabelle indica il prefisso da aggiungere ai loro attributi (es. All'interno di *lineitem*, l'attributo orderkey verrà chiamato *l_orderkey*).
- Le relazioni sono indicate con una freccia orientata in direzione "uno a molti" (es. a una *region* appartengono più *nation*).
- i numeri indicano la cardinalità delle tabelle. Per alcune di esse questo numero va moltiplicato per un *fattore di scala* che varia in base alla dimensione scelta per il database.

Il *qualification database* è un database creato e popolato con lo scopo di testare e convalidare le query. L'intento è che le funzionalità esercitate dalle query sul *qualification database* siano le stesse anche sul *test database* durante la valutazione delle prestazioni. A tal fine, il *qualification database* deve essere identico al *test database* (utilizzato per *l'executed load test* e *performance* test) sotto quasi ogni punto di vista. Il *qualification database* può differire dal *test database* solo se la differenza è direttamente correlata alle dimensioni. Ad esempio, se il *test database* utilizza il partizionamento orizzontale, anche il *qualification database* deve essere strutturato allo stesso modo, tuttavia il numero di partizioni può essere diverso. Il *qualification database* può però essere configurato in modo tale da utilizzare un sottoinsieme rappresentativo dei processori / core / thread /memoria /dischi, utilizzati per la configurazione del *test database*. La popolazione del *qualification database* deve essere esattamente uguale a un fattore di scala (SF) di 1 (equivale a 1GB di dati). DBGen è un pacchetto software messo a disposizione da TPC che deve essere utilizzato per popolare il database.

Popolazione Database

La popolazione del *test database* si compone di due fasi logiche:

- *Generation Phase*: utilizza DBGen per generare record in un formato che ne facilita l'uso da parte del DBMS. I record generati possono essere trasmessi attraverso un canale di comunicazione, memorizzati nella memoria o memorizzati in file su supporti di memorizzazione.
- *Loading Phase*: i record generati nella fase precedente vengono caricati nelle tabelle del database.

Generazione Dati

Il processo di generazione e caricamento dei record può essere realizzato in due modi:

- *Load from store*: I record generati da DBGen vengono prima memorizzati (in memoria o su un supporto di memorizzazione); essi possono essere ordinati, partizionato o trasferiti al SUT (*System Under* Test). Dopo la creazione delle tabelle sul SUT, i record vengono infine caricati nelle tabelle del database.
- *In-line load*: I record generati da DBGen vengono fatti passare attraverso un canale di comunicazione e direttamente caricati nelle tabelle del database.

Componenti

Il benchmark è definito come l'esecuzione del *load test* seguito dal *performance test*.

- Il *load test* inizia con la creazione delle tabelle del database e comprende tutte le attività necessarie per portare il sistema in esame alla configurazione che precede immediatamente l'inizio del *performance test*. Non può includere l'esecuzione di qualsiasi tipo di query.
- Il *performance test* è composto da due *run*, eseguite consecutivamente dopo il *load test*.

Una *run* consiste nell' esecuzione del *power test*, seguita dall' esecuzione del *throughput test.*
Componenti di una *Run*:

- Una query è definita come una qualsiasi delle 22 query TPC-H.
- Un *set* di query è definito come l'esecuzione sequenziale di tutte le query.
- Uno *stream* di query è definito come l'esecuzione sequenziale di un unico insieme di query presentato da un singolo utente.
- Un *refresh stream* è definito come l'esecuzione sequenziale di un numero intero di coppie di funzioni di aggiornamento lanciato da un programma batch.
- Una *coppia di refresh function* è l'insieme delle due *refresh function* TPC-H.
- Una sessione è definita come un processo in grado di supportare l'esecuzione di un flusso query (*query* stream) o un flusso di aggiornamento *(refresh* stream).

Il meccanismo utilizzato per inviare query, aggiornare le funzioni del sistema in prova (SUT) e misurare il loro tempo di esecuzione viene chiamato *driver.* Il driver è un'entità logica che può essere implementata utilizzando uno o più programmi fisici, processi o sistemi.
La comunicazione tra il driver e il SUT deve essere limitata a una sola sessione (un flusso di query o un flusso di funzioni di aggiornamento). Alle sessioni è proibito comunicare tra loro, tranne ai fini dello *scheduling* delle funzioni di aggiornamento.

Tutte le sessioni che eseguono un flusso di query devono essere inizializzate esattamente nello stesso modo. L'inizializzazione della sessioni che eseguono un *refresh stream* possono differire dalle sessioni di *query stream*.

3.2 Configurazione cluster

Il benchmark TPC-H è stato eseguito su un cluster composto da 7 nodi di commody Hardware.
Di seguito sono riportate le caratteristiche tecniche di ogni nodo:

- CPU: Intel i7-4790, 4 Core, 8 threads, 3.6Ghz
- RAM: 32 GB
- HARD-Drive: 2x2TB HDD
- Ethernet: Gigabit
- Sistema Operativo: CentOS 6.6 (Linux)

E' stata stata installata la piattaforma Hadoop version 5.3.1 e il sistema di analisi Hive versione 0.13.

3.3 Risultati sperimentali

Lo scopo del test è quello di valutare le prestazioni del sistema Hive effettuando un benchmarking di TPC-H variando la modalità di memorizzazione dei dati e la configurazione del sistema stesso. In particolare si è studiata la differenza di performance memorizzando prima i dati nel formato AVRO e poi in quello PARQUET.
Ci si è focalizzati sul *power test*, ossia sono stati calcolati i tempi di esecuzione delle 22 query TPC-H cambiando la dimensione del set di dati, trascurando però le funzioni di aggiornamento.
Per popolare il database si è scelto di usare DBGen, il software messo a disposizione dai Benchmark TPC. Il sistema operativo installato sul cluster, al momento dei test, non aveva disponibile la libreria GLIBC versione 2.14 richiesta da DBGen (tale versione non è ancora stata rilasciata per la distribuzione CentOS) e quindi il *load test* è stato eseguito su un laptop

(che monta una partizione linux, su cui è installata la distribuzione Xubuntu) generando i dati su una cartella condivisa col cluster. Il caricamento dei record è stato eseguito secondo la modalità *load from store*: DBGen ha generato 8 file di testo (uno per ogni *entità*) che sono stati memorizzati su HDFS; in un secondo momento i file contenenti i dati sono stati caricati sulle tabelle, tramite HiveQL. Sul laptop è stato anche creato il *qualification database,* dove sono state validate le query.
Sono stati creati più *test database* per valutare le prestazioni variando anche la dimensione del dataset. in particolare sono stati creati 8 *test database*:

- 4 in formato AVRO e 4 in formato PARQUET.
- Le size scelte per i 4 *test database (*di entrambi i formati) sono state 1 GB, 10 GB, 100 GB, 1000 GB (=1 TB)
- Sui database di dimensione 1, 10, 100 GB sono state eseguite tutte e 22 le query del benchmark.
- Sul database da 1 TB di dati sono state eseguite le 3 query più semplici e complete possibili, più precisamente la query 1, la query 3 e la query 6.

Anche per il sistema Hive sono state testate due diverse configurazioni:

- 51 *container* da 1 Core, 1 GB di RAM.
- 25 *container* da 2 Core, 2 GB di RAM.

Per ogni configurazione un container funge da *application master,* ovvero ha il compito di allocare le risorse e distribuire il carico di lavoro da svolgere tra gli altri container.

Formato File: PARQUET

Query ID / Size	1GB	10GB	100GB	1TB
1	75	194	2217	20721
2	143	225	1001	
3	140	205	4493	43349
4	153	338	3562	
5	195	313	3406	
6	40	97	903	4976
7	204	/	8098	
8	205	270	1476	
9	342	136	2750	
10	161	175	2601	
11	144	163	288	
12	113	/	2552	
13	141	109	1604	
14	68	150	1188	
15	108	183	2098	
16	149	276	1395	
17	146	821	9304	
18	229	/	11427	
19	89	594	6573	
20	179	329	2299	
21	372	1119	12287	
22	157	163	1175	
totale	3553	5860	82697	69046
Inc.		165%	1411%	

Formato file: AVRO

Query ID / Size	1 GB	10 GB	100GB	1TB
1	98	451	4335	47994
2	141	231	1116	
3	143	340	6319	64630
4	174	692	6401	
5	203	517	4886	
6	53	293	3033	30247
7	224	704	15077	
8	206	339	2423	
9	342	682	4150	
10	169	301	4608	
11	146	165	360	
12	125	563	5077	
13	138	175	1823	
14	81	353	3589	
15	118	380	4756	
16	150	275	1499	
17	158	1000	11615	
18	249	973	12404	
19	93	675	7730	
20	189	528	5027	
21	376	1557	17871	
22	153	195	1346	
totale	3729	11389	125445	142871
Inc.		305%	1101%	

Figura 3.3 sono riportati i tempi parziali di ogni query, e il totale ottenuto da Hive con la seguente *Configurazione:* 51 container, 1 Core, 1 GB Ram. L'ultima Riga invece illustra l'incremento medio in percentuale dei tempi di esecuzione tra un size e la precedente.

la figura 3.3 elenca i tempi di esecuzione delle 22 Query TPC-H registrati adottando per ogni database la prima configurazione; i tempi sono espressi in secondi arrotondati all'unità. Si può notare che:

- La query 21 è quella che impiega più tempo rispetto alle altre, indipendentemente dalla size del database e dal tipo di formato di memorizzazione: questo avviene perché sono coinvolte molte tabelle, il che richiede molteplici join e quindi molti stage *mapReduce.* Inoltre essa presenta una *subquery* che, riscritta in HiveQL, diventa una query separata e richiede quindi la creazione di una *tabella ad hoc* che conterrà il risultato intermedio.
- La memorizzazione PARQUET e AVRO tra 10 e 100GB non mostra miglioramenti di prestazione se si utilizza la prima configurazione (aumento la size di 10 volte i tempi di esecuzione in media aumentano anche essi di 10 volte). Questo è dovuto al fatto che la capacità assegnata ai container (1 Core,1 GB di Ram) non è sufficiente per determinare un aumento significativo di prestazioni. Inoltre per queste dimensioni la memorizzazione *colonnare* determina un aumento netto di prestazione rispetto alla memorizzazione *sequenziale,* infatti i tempi di esecuzione in media aumentano di 3 volte per il formato file *AVRO.*
- se si presta attenzione infatti al tempo totale di esecuzione, si nota come la memorizzazione PARQUET garantisca prestazioni *migliori* rispetto a quella *sequenziale,* proprio grazie al tipo di accesso *colonnare* che permette di evitare di accedere a record "*inutili"*.

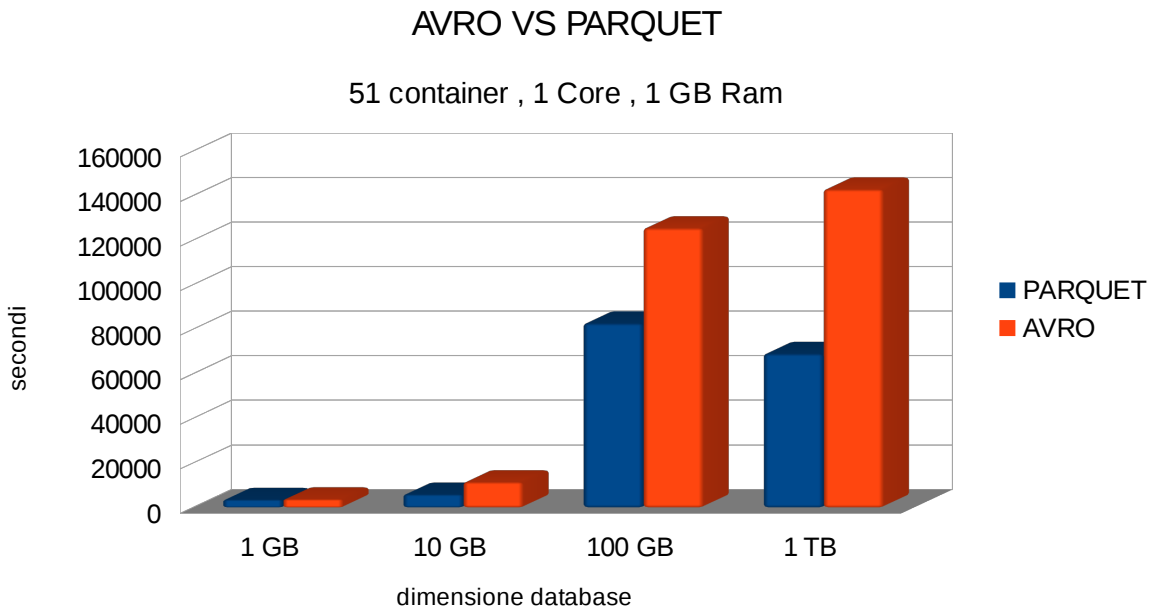


Figura 3.4 viene mostrata la differenza di prestazioni in termini di tempi di esecuzione di tutte le query TPC-H in formato AVRO e formato PARQUET

Tralasciando il database da 1TB, in quanto sono state eseguite solo 3 query, la figura 3.4 evidenzia nel database da 100 GB la netta differenza di prestazione tra la memorizzazione *PARQUET* e la memorizzazione *AVRO*. In figura 3.5 verranno mostrati nel dettaglio i tempi di esecuzione delle singole query sul database da 100 GB.

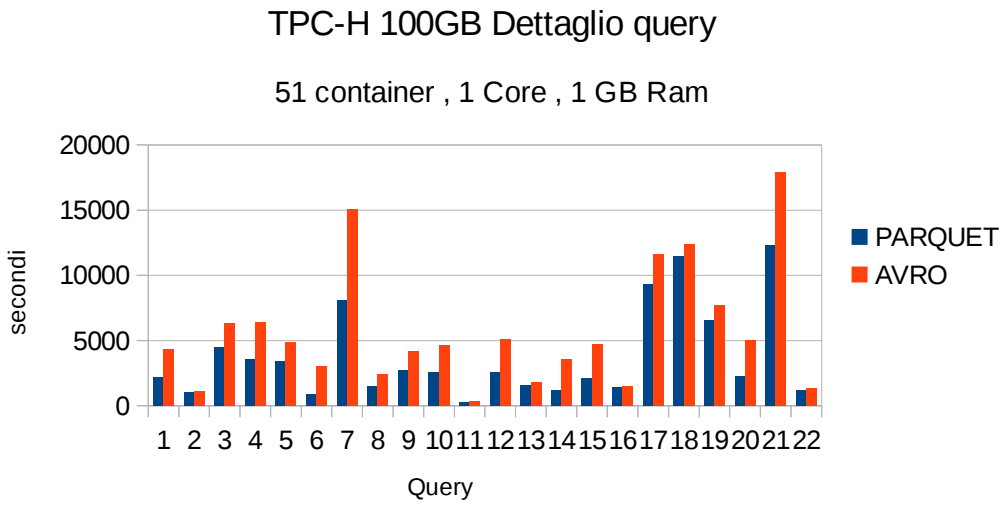


Figura 3.5: nella figura vengono evidenziati i tempi di esecuzione di tutte e 22 le query TPC-H nei diversi formati sul database da 100GB

- Lo scarto più netto è evidenziato dalle query 7 e 21. questo principalmente è dovuto alla presenza di predicati di selezione che, nel caso della formattazione *PARQUET*, evita al sistema di accedere a "record inutili" per determinare il risultato della query. Al contrario ad esempio, la query 22 e la query 2 hanno tempi di esecuzione *simili* per entrambi i formati, in quanto i predicati di selezione non offrono un vantaggio così evidente.
- La query 11 risulta essere una delle query più rapide in quanto richiede una *somma* e il predicato di selezione riduce molto i record da considerare.

Formato File: PARQUET

Query ID / Size	1GB	10GB	100GB	1TB
1	80	195	1680	16021
2	137	225	963	
3	137	475	3693	36078
4	155	399	3222	
5	189	458	3220	
6	40	95	754	7210
7	206	668	6928	
8	202	325	1380	
9	341	577	2433	
10	159	359	2266	
11	145	158	275	
12	111	273	2085	
13	139	265	1455	
14	66	148	1048	
15	106	181	2082	
16	149	279	1387	
17	145	821	7366	
18	232	1143	9856	
19	89	596	5254	
20	177	321	2025	
21	377	1264	10230	
22	159	178	1059	
totale	3541	9403	70661	59309
Inc.		265%	751%	

Formato File : AVRO

Query ID / Size	1 GB	10 GB	100GB	1TB
1	96	373	3429	38721
2	141	227	1043	
3	140	568	5154	58024
4	169	628	5476	
5	195	572	4319	
6	54	256	2426	26105
7	213	934	8796	
8	210	406	2050	
9	221	664	3371	
10	168	470	3801	
11	145	163	316	
12	121	434	3994	
13	135	258	1500	
14	81	314	2932	
15	120	350	3970	
16	146	267	1354	
17	153	931	9154	
18	235	1106	10268	
19	92	630	6072	
20	189	504	4147	
21	381	1597	14575	
22	155	186	1180	
totale	3560	11838	99327	122850
Inc.		333%	839%	

Figura 3.6 Configurazione HIVE: 25 container, 2 Core, 2 GB Ram. S*ono riportati i tempi parziali di ogni query espressi in secondi arrotondati all'unità, e il totale ottenuto. L' ultima riga invece illustra l'incremento medio in percentuale dei tempi di esecuzione tra un size e la precedente.*

La figura 3.6 mostra i tempi che si ottengono cambiando la configurazione del sistema:

- aumentando i numeri di Core e soprattutto la Ram, si può notare che le prestazioni *migliorano* all'aumentare della dimensione del database, anche se i container si dimezzano. Questo è dovuto al fatto che ogni container è in grado di elaborare una mole maggiore di dati alla volta diminuendo il numero di accessi alla memoria.
- Questo però non avviene per il database da 10GB presumibilmente perché la configurazione precedente era già sufficiente e quindi un aumento di memoria a spese del numero di container non porta sostanziali miglioramenti.

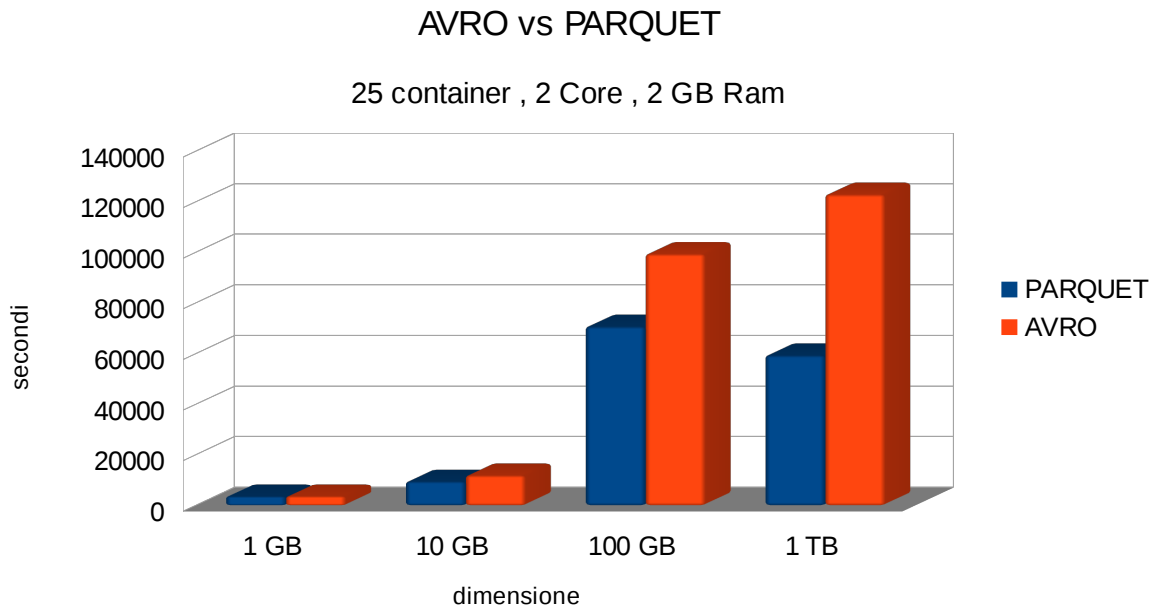


Figura 3.7 viene mostrata la differenza di prestazioni in termini di tempi di esecuzione di tutte le query TPC-H in formato AVRO e formato PARQUET con la seconda configurazione

Nell' illustrazione (figura 3.7) viene mostrata in maniera più chiara la differenza di prestazione totale tra i due tipi di formato file per ogni database esaminato.come già anticipato, anche questo grafico evidenzia l'efficacia del formato *PARQUET* sui database di grandi dimensioni.

Tralasciando sempre il database da 1 TB, nel grafico (figura 3.8) successivo verranno analizzati in dettaglio i tempi di tutte e 22 le query eseguite sul database da 100GB.

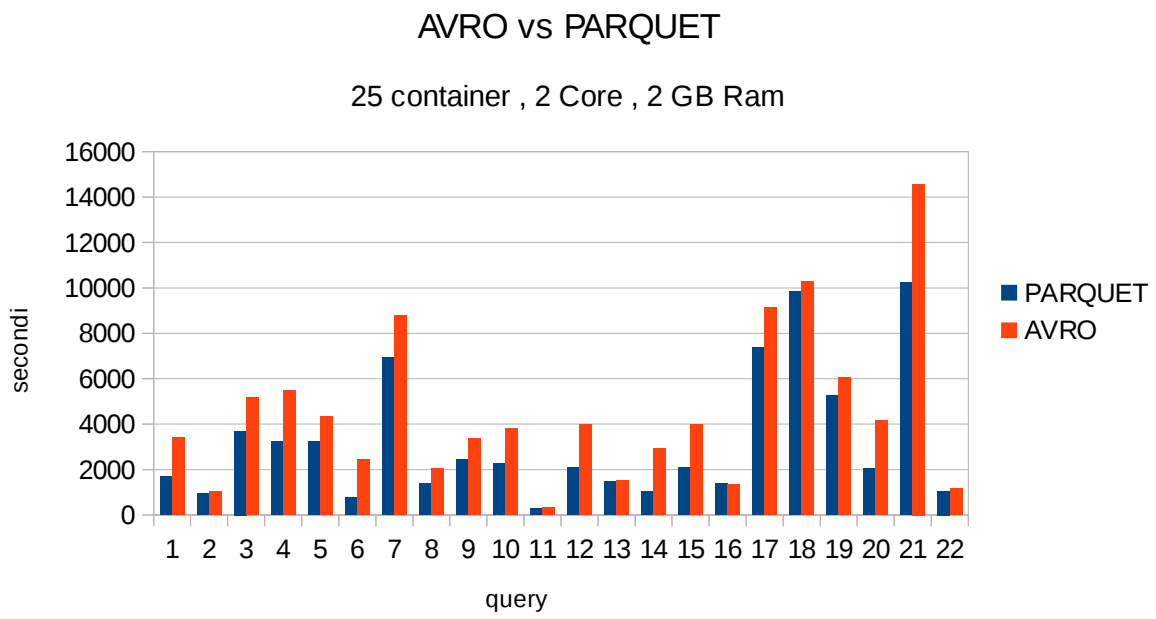


Figura 3.8 vengono confrontati i risultati ottenuti sul database da 100GB cambiando il formato dei file

in questo grafico si nota che, cambiando la configurazione, le differenze di prestazioni tra il formato PARQUET e il formato AVRO rimangono sostanzialmente le stesse poiché il cambiamento di configurazione influisce in entrambi solo sul tempo totale di esecuzione.
Nelle prossime *illustrazioni* (figura 3.9 e 3.10) verranno messi a confronto i tempi d'esecuzione totali ottenuti su tutti i database con entrambe le *configurazioni,* memorizzando prima i dati in formato PARQUET e poi in formato AVRO (sul database da 1TB sono state eseguite solo 3 delle 22 query a disposizione).

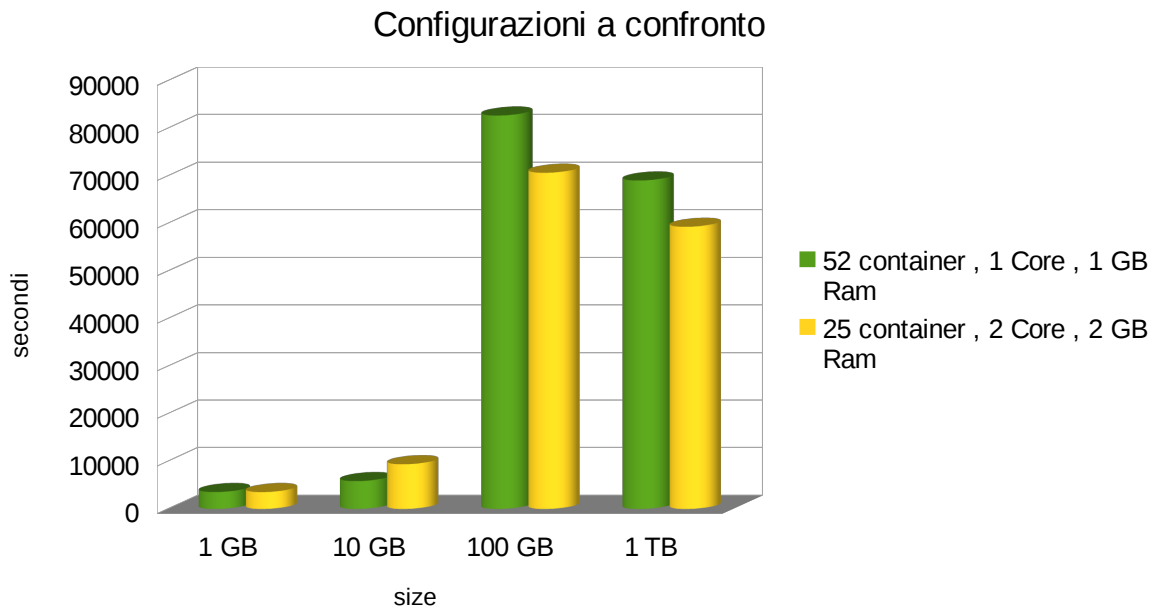


figura 3.9 formato PARQUET

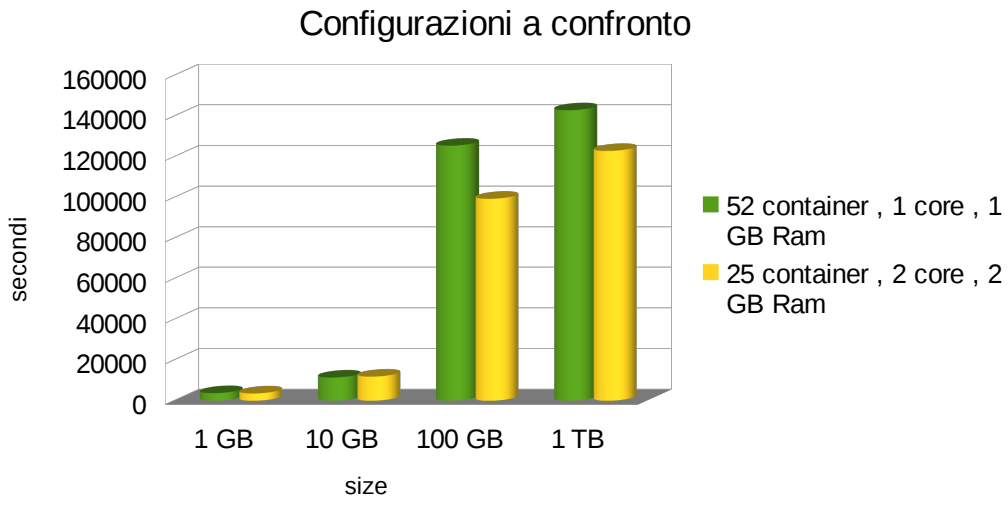


figura 3.10 formato AVRO

esaminando i tempi totali per ogni database si può notare come la memorizzazione *PARQUET* sia molto più efficace all'aumentare della dimensione del database e la stessa considerazione si può fare per la *configurazione* che prevede meno container, dotati di più core e memoria.

Il motivo di tale miglioramento sicuramente è dovuto alla possibilità di elaborare più dati effettuando meno accessi alla memoria.
Infine l'ultima analisi (figura 3.11) è un confronto sul database da 100GB fornato *PARQUET*, tra Hive (25 container, 2 core e 2 GB di Ram) e Spark SQL (6 executors, 7 core, 21 GB di RAM), un framework che invece di sfruttare MapReduce lavora in memoria centrale grazie alla possibilità di usare un core Spark.

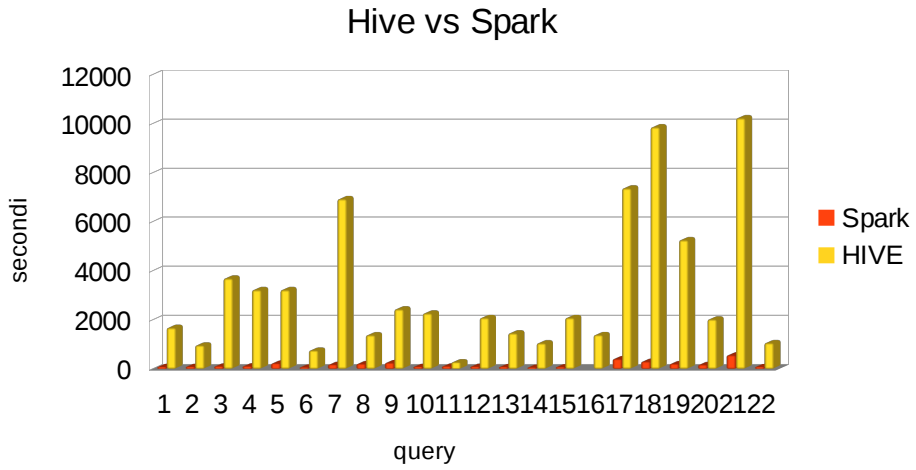


Figura 3.11 in questa figura si notano le enormi differenze prestazionali in termini di tempi d'esecuzione tra i due sistemi

Il grafico evidenzia la notevole differenza di prestazioni raggiunte dai due sistemi. Spark è nettamente più veloce, spesso anche di un ordine di grandezza, e mantiene questa differenza prestazionale anche sugli altri database. Questo è dovuto al fatto che Hive ad ogni stage *MapReduce* memorizza i risultati su disco, mentre spark lavora direttamente in memoria, col rischio di non averne a sufficienza e di fallire l'esecuzione (come è successo in questo caso, per la query 16).

4.Conclusioni

l'avvento dei BigData ha reso necessario lo studio di nuove tecnologie e strumenti per la memorizzazione e l'analisi dei dati. Hadoop è un software open-source ampiamente utilizzato, in quanto offre un filesystem distribuito(HDFS) molto flessibile e con un discreto grado di tolleranza ai guasti e di una piattaforma per l'elaborazione dei dati. Per l'interrogazione dei dati esistono molteplici software, tra cui Hive che sfrutta MapReduce. Queste nuove tecnologie sono in grado di processare grandi volumi di dati e più il volume è grande, più questi sistemi sono prestazionali.

Sono stati effettuai dei test, valutando il sistema modificando la dimensione dei database, e il formato di memorizzazione dei file:

il formato file AVRO, a scansione *sequenziale*, migliora le proprie prestazioni passando dal database da 1GB a quello da 10GB, mentre aumentando ulteriormente il volume dei dati (100GB) in tempi di esecuzione aumentano in maniera più che proporzionale. Lo stesso accade anche utilizzando Il formato PARQUET (memorizzazione colonnare).

Le differenze di prestazioni tra i 2 formati di memorizzazione però cresce al crescere dei volumi di dati: se per il database da 1GB i tempi AVRO sono più alti del 104% rispetto a quelli PARQUET, il divario aumenta fino a un +207% per il database da 1TB. Questo è dovuto dal fatto che la memorizzazione colonnare consente di leggere direttamente i campi, e questo permette di evitare la lettura di record inutili ai fini della query a differenza di AVRO, che accedendo ai file in maniera sequenziale è costretto a scandirli/leggerli tutti.

Hive, comunque, impiega molto tempo per l'esecuzione delle query perché MapReduce ad ogni stage/job impone la memorizzazione del risultato su disco, a differenza di altri sistemi come ad esempio spark che, lavorando in memoria centrale, ottiene risultati nettamente migliori.

Per ridurre i tempi d'esecuzione, bisogna trovare un giusto equilibrio tra il numero di container (nodi che eseguono i job MapReduce) e la loro potenza di calcolo (numeri di Core e GB di Ram assegnati ad ognuno), in modo da ridurre il più possibile gli accessi al disco; dai test effettuati infatti si nota che, all'aumentare del volume dei dati, è più performante la configurazione che assegna la metà dei container ma con il doppio dei core e della Ram, e quindi è in grado di elaborare una quantità maggiore di dati alla volta:

sia il formato AVRO che il Formato PARQUET configurando Hive con 25 container, 2 Core e 2GB di RAM, migliorano le relative prestazioni anche sui database più grandi (100GB). Per il formato AVRO, i tempi ottenuti con la configurazione precedente (51 container, 1 Core e 1GB RAM) sono più lunghi del 126% per il database da 100GB e del 116% per quello da 1TB (considerando solo 3 query su 22). Per il formato PARQUET invece, la prima configurazione registra un'aumento del 116% per entrambi i database (100GB e 1TB) rispetto alla seconda. Tra i due formati, cambiando per entrambi la configurazione, il divario prestazionale resta pressochè invariato.

D'altro canto Hive oltre a garantire l'esecuzione della query (può impiegare molto tempo ma non fallisce), utilizza un DBMS tradizionale per la memorizzazione dei metadati e un linguaggio molto simile allo standard SQL-92: questo ne semplifica l'utilizzo per chi è abituato a lavorare con i database relazionali.

Per risolvere il problema del tempo di esecuzione si sta sviluppando il progetto Hive on Spark che permetterà ad Hive di elaborare i dati in memoria centrale, sfruttando il Core di Spark, evitando di accedere al disco, ad ogni iterazione.

Sicuramente le piattaforme studiate per gestire e memorizzare i BigData rappresentano il futuro, ma al momento non sono ancora in grado di sostituire definitivamente i DBMS tradizionali poiché i BigData richiedono ingenti investimenti infrastrutturali, non sempre motivati; bisogna inoltre ancora risolvere il problema della sincronizzazione dei file memorizzati sui sistemi distribuiti, il che rende molto complessa la migrazione dei dati dai DBMS tradizionali ai NoSQL. Per questi motivi, e per il fatto che moltissime aziende ancora oggi utilizzano i DBMS tradizionali, si pensa che questi sistemi coesisteranno per un lungo periodo tempo.

.

Bibliografia

[1] “BigData: Architettura, tecnologie e metodi per l'utilizzo di gradi basi di dati” di Alessandro Rezzani editore Maggioli 2013

[2] Documentazione Hadoop: http://wiki.apache.org/hadoop/

[3] Architettura Hadoop https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[4] “Nosql Distilled: A Brief Guide to the Emerging World of Polyglot Persistence” di Pramod J. Salad e Martin Fowler editore Addison-Wesley 2013

[5]HiveQL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual

[6]Formato PARQUET: https://blog.twitter.com/2013/dremel-made-simple-with-parquet

[7]Architettura Hive: https://cwiki.apache.org/confluence/display/Hive/Design

[8]MapReduce: http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Mapper

[10]Piani Esecuzione MapReduce: http:// www.slideshare.net/qiuxiafei/pig-map-reduce-execution

[9]Funzionamento MapReduce: http://didamatica2010.di.uniroma1.it/twiki/viewfile/AA/AlgoritmiAvanzati_2011_12?rev=1;filename=Presentazione_AV_Dario_Frascaria.pdf

[11]HiveQL operator: https://cwiki.apache.org/confluence/display/Hive/GettingStarted#GettingStarted-SQLOperations

[12] Operatori MapReduce: http://www.slideshare.net/zshao/hive-data-warehousing-analytics-on-hadoop-presentation

[13]Join MapReduce: https://cwiki.apache.org/confluence/download/attachments/27362054/Hive+Summit+2011-*join.pdf?version=1&modificationDate=1309986642000*

[13 bis] Common Join e MapJoin: *https://www.facebook.com/notes/facebook-engineering/join-optimization-in-apache-hive/470667928919*

[14] Group by MapReduce: http://www.slideshare.net/zshao/hadoop-and-hive-presentation

[15]Ottimizzazione Join:
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+JoinOptimization
[16]TPC-H query in HiveQL: https://issues.apache.org/jira/browse/HIVE-600
[17]Benchmark TPC-H:
http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf benchmark
[18]Utilizzo DBGen: https://husnusensoy.wordpress.com/2010/10/22/create-your-own-oracle-tpc-h-playground-on-linux/
[19]TPC-H Benchmarking: R.Moussa "Massive data analytics in the cloud: TPC-H experience on hadoop clusters"
[20]TPC-H sul cluster: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.9455&rep=rep1&type=pdf