0 Column Stores

Fino a questo momento abbiamo di varie tipologie di basi, ovvero modelli relazionali, key-value stores e document stores. Tra queste tipologie i modello NoSQL si rendono necessari nel momento in cui non serve più garantire le proprietà ACID, ma si preferisce garantire efficienza e velocità di accesso ai dati.

In tutti i precedenti capitoli è sempre stata discussa la **modalità di interazione** con il **data model** per ognuna delle categorie viste ma non è mai stato mostrato **come** i dati vengano effettivamente memorizzati all'interno dei database.

Prima di introdurre i **column stores** è necessario fare una breve digressione dove andremo a vedere come i dati vengono memorizzati all'interno dei database relazionali, useremo i database relazionali dal momento che sono quelli con la quale la maggior parte ha familiarità. Tutto questo verrà ulteriormente approfondito in uno dei successivi capitoli.

0 Architettura di un R-DBMS

Quello che vediamo in Figure 0.1 è una rappresentazione dell'organizzazione interna delle componenti di un DBMS relazionale.

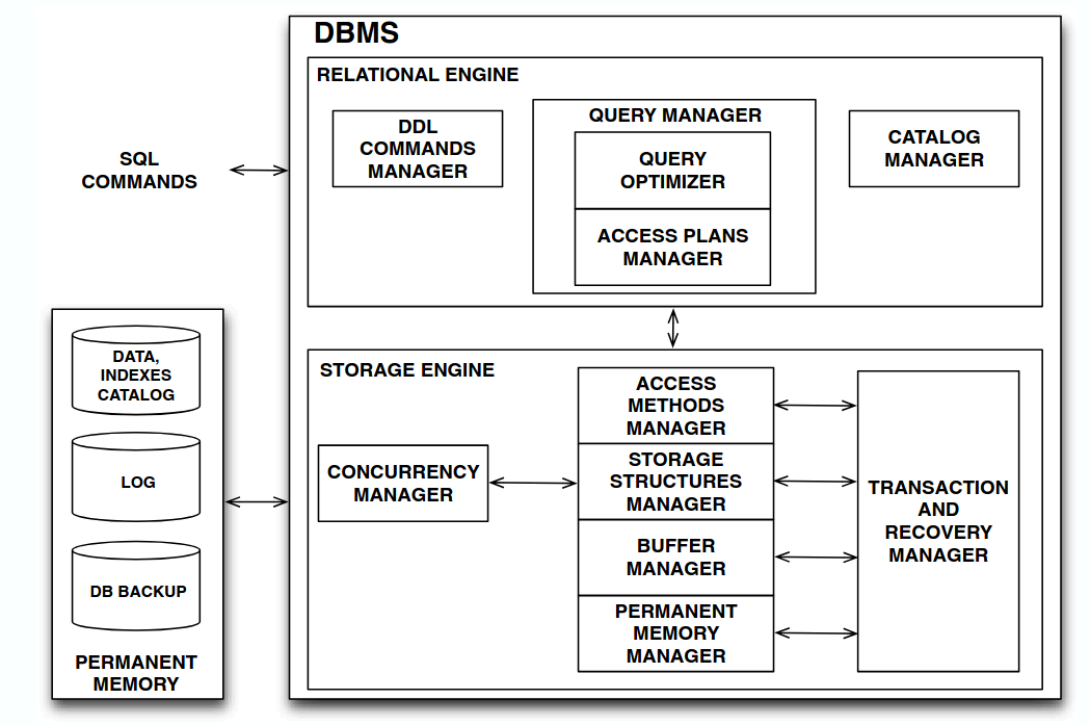

Figure 0.1: Organizzazione interna delle componenti di un R-DBMS

Ciò che è importante notare è che possiamo distinguere due macro aree:

- La parte superiore dell'immagine rappresenta la parte logica del database, anche detta **relational engine**, la quale si occupa di gestire le query, l'ottimizzazione delle stesse e la gestione delle transazioni e accede alla parte inferiore tramite delle API
- La parte inferiore dell'immagine rappresenta la parte fisica del database, anche detta **storage engine**, la quale si occupa di gestire l'effettiva memorizzazione dei dati

Possiamo notare come la gestione delle transazioni sia demandata alla parte dello storage engine, il quale è l'unico componente che ha accesso diretto al file system. Ciò che è importante per questo capitolo è comprendere il funzionamento delle seguenti componenti:

- **Storage structures manager**
- **Buffer manager**
- **Persistent memory manager**

Andremo ad analizzare il funzionamento di queste componenti nelle sezioni successive.

Persistent Memory Manager
La componente del **persistent memory manager** si occupa di **astrarre** i dispositivi fisici di memorizzazione fornendone una **vista logica**. Tipicamente la memoria che viene gestita si può vedere costituita di due livelli:

- **Memoria principale** (RAM), che è molto veloce (10-100ns), piccola (nell'ordine dei GigaByte), volatile e molto costosa
- **Memoria secondaria** (Dischi), che è al contrario permanente, di più grande capacità (svariati TeraByte) ed economica ma molto più lenta (5-10ms).

Il principale motivo della latenza nell'accesso ai dati su un disco fisico (hard disk) è dovuto al fatto che il disco è un dispositivo meccanico, dunque per accedere fisicamente ai file è necessario accedere alla posizione dei dati richiesti muovendo la testina sulla porzione di disco corretta. Dal momento che questo movimento è estremamente lento rispetto al resto del sistema, è ottimale cercare di minimizzare il numero di accessi, cercando di trasferite la maggior quantità di dati utili possibile.

Gestore del Buffer
Il compito del **buffer manager** è quello di trasferire pagine di dati tra la memoria principale e quella persistente, fornendo un'astrazione della memoria persistente come un insieme di pagine utilizzabili in memoria principale, nascondendo di fatto il meccanismo di trasferimento dei dati tra memoria persistente e *buffer pool*.

L'obiettivo principale di questo componente è quello di evitare il più possibile che vengano effettuate letture ripetute di una stessa pagina, cercando di mantenere una sorta di **cache**, e di minimizzare il numero di scritture su disco.

L'utilizzo di questo componente è fondamentale per poter sfruttare al meglio il principio della località spaziale e temporale, ovvero il fatto che se un dato viene richiesto, è probabile che vengano richiesti anche dati 'vicini' (spaziale) o che lo stesso dato venga richiesto più volte in un breve lasso di tempo (temporale).

Strutture di Memorizzazione
In questa sezione andiamo a vedere come vengono memorizzati i record all'interno di una base di dati. In generale possiamo dire che i dati vengono salvati su **file** ovvero degli oggetti che siano **linearmente indirizzabili** (tramite degli indirizzi di un certo numero di byte).

Sappiamo che all'interno di una base di dati relazionale, i dati all'interno di una tabella sono organizzati in **record**, dove ogni record è costituito da un insieme di **fields** (o attributi), ognuno di questi può essere strutturato in maniera diversa:

- Lunghezza fissa: il campo occupa sempre lo stesso numero di byte (es. `integer`, `float`, `date`)
- Lunghezza variabile: il campo può occupare un numero variabile di byte (es. `char(n)`, `varchar(n)`, `text`)

I record possono inoltre essere separati tra loro tramite un **delimitatore** (es. nei file `csv`) o ancora possono avere un prefisso speciale che ne indica la lunghezza: `record = (prefix, fields)`, dove il prefisso può contenere diversi tipi di informazioni:

- La lunghezza in byte della parte del valore

- Il numero di componenti
- L'offset di inizio del record successivo
- Altre informazioni di controllo

Normalmente i record vengono memorizzati all'interno di una **pagina di memoria** e per ognuno viene memorizzato un **physical record identifier** che ne indica la posizione tramite le seguenti informazioni:

- Page number: il numero della pagina di memoria
- Offset: la posizione del record all'interno della pagina

Il modo più semplice in cui i record possono essere memorizzati all'interno di una pagina è quello di utilizzare un approccio **seriale**, ovvero memorizzare i record uno di seguito all'altro. Questo approccio però non consente di accedere ai dati in maniera efficiente.

Supponimao per esempio che per una relazione siano presenti dei campi che sono più importanti di altri, per esempio l'età di una persona (ad esempio in un contesto in cui si voglia fare analisi demografica). Se riuscissimo a memorizzare i record in maniera ordinata rispetto a questo campo, potremmo velocizzare di molto le operazioni di lettura che coinvolgono questo campo (ad esempio tutte le persone con età maggiore di 30 anni). Questo approccio prende il nome di **sequenziale**. Il problema di questo approccio è che le operazioni di **inserimento**, cancellazione e aggiornamento diventano molto più complesse e costose dal momento che è necessario mantenere l'ordinamento. Tutto ciò che viene dopo il record inserito deve essere spostato in avanti.

0 Column Stores

I **column stores** (o **columnar databases**) sono una tipologia di basi di dati NoSQL che memorizzano i dati organizzandoli per colonne invece che per righe come avviene nei database relazionali tradizionali (row stores).

In un database relazionale tradizionale, nel momento in cui siamo interessati a leggere uno specifico valore di un record, siamo costretti a leggere l'intero record e scartare tutto ciò che non serve allo scopo della query in esame; questo è particolarmente inefficiente nel momento in cui i record sono composti da molti campi. Consideriamo per esempio la seguente relazione `BookLending`:

BookLending	BookID	ReaderID	ReturnDate
	123	225	25-10-2016
	234	347	31-10-2016

All'interno di un **row store** i dati verrebbero memorizzati in questo modo: 123, 225, 25-10-2016, 234, 347, 31-10-2016,

Utilizzando un column store, questa operazione diventa molto più efficiente, dal momento che ci permettono di leggere solo i campi di interesse. Questo approccio è particolarmente vantaggioso in scenari di **analisi dei dati** e **data warehousing**, dove spesso si eseguono query che coinvolgono solo un sottoinsieme di colonne su grandi volumi di dati. La stessa tabella di sopra, all'interno di un **column store** verrebbe memorrizzata nel modo seguente: 123, 234, 225, 347, 25-10-2016, 31-10-2016,

Come anticipato nella sezione precedente è possibile andare ad utilizzare questo approccio indipendentemente dal modello dei dati utilizzato, dunque è possibile trovare column stores che implementano modelli relazionali, key-value o documentali.

0 Vantaggi e Svantaggi dei Column Stores

Andiamo si seguito a specificare tutti i pro e contro dell'adozione di un column store:

- Soltanto i dati che sono effettivamente necessari vengono letti dal disco nella memoria principale, dal momento che le pagine non contengono più record, bensì colonne, riducendo così il carico di I/O ✅
- I valori di una colonna hatto tutti lo stesso dominio e possono essere **meglio compressi** grazie alla località dei dati ✅
- **Iterazione** e **aggregazione** di valori (es. calcolo di somma, media, massimi e minimi valori di una colonna) possono essere effettuate in maniera veloce, dal momento che i valori sono stati memorizzati in maniera contigua ✅
- Aggiungere **nuovi campi** ad una relazione (o documento) è molto semplice ✅
- **Combinare** i valori da diverse colonne è particolarmente costoso, dal momento che è necessario capire quali valori nelle colonne corrispondono ad una stessa tupla ❌
- Aggiungere **nuovi record** è particolarmente costoso, dal momento che è necessario andare ad aggiornare tutte le colonne ❌

0 Compressione delle colonne

Un algoritmo di compressione serve a prendere dei dati in input e a trasformarli in una rappresentazione più compatta in modo da ridurne lo spazio di memorizzazione necessario. Esistono diversi algoritmi di compressione, in particolare li possiamo dividere in algoritmi **lossless** (senza perdita di informazione) e **lossy** (con perdita di informazione). Nel contesto delle basi di dati, è di fondamentale importanza utilizzare algoritmi **lossless**, dal momento che non è accettabile perdere informazioni.

Esistono diversi modi di applicare compressione sui dati memorizzati in un column store, tutti questi sfruttano il fatto che i dati all'interno di una colonna sono spesso **ripetuti** o **simili** tra loro. Nelle prossime sezioni andremo a vedere gli approcci più comuni.

Run-Length Encoding (RLE)

Si tratta di un algoritmo di compressione alla base del quale c'è il principio di contare quante volte consecutive un certo valore viene ripetuto. Dopo la compressione, ogni valore verrà rappresentato dalla tupla che segue: `(value, start row, run-length)`. Figure 0.2 mostra un esempio del funzionamento di questo algoritmo.

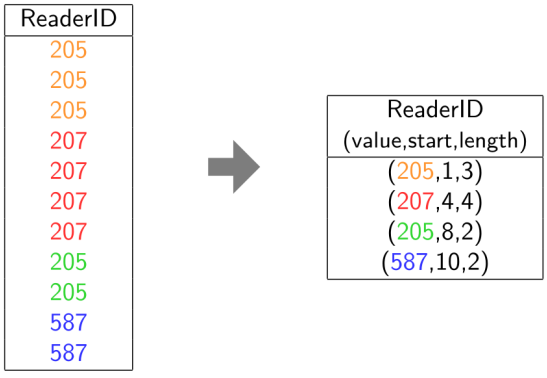

Figure 0.2: Esempio di Run-Length Encoding

> **Remark**
>
> Apparentemente mantenere memorizzata anche l'informazione legata al valore di start di un record sembra superfluo, tuttavia questa informazione si rivela fondamentale nel momento in cui vogliamo andare a ricostruire dati parziali.

Uno dei primi strumenti che hanno utilizzato questo tipo di encoding è stato quello per comprimere immagini bianco e nero in formato PBM (Portable Bitmap). Il grande vantaggio dell'utilizzo di questo approccio consiste nel forte tasso di compressione.

Bit Vector Encoding

In questo algoritmo di compressione, per ogni valore tra quelli presenti nella colonna viene creato un vettore con un bit per ogni riga. Se il valore è presente allora il bit viene settato a 1, altrimenti a 0. Figure 0.3 mostra un esempio del funzionamento di questo algoritmo.

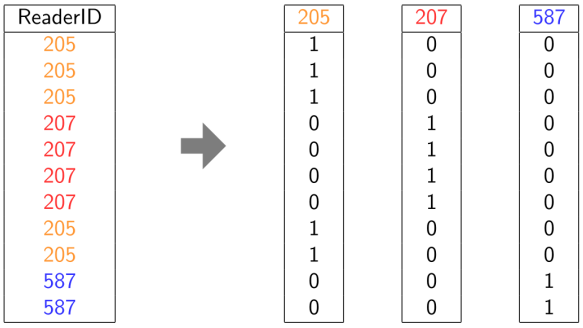

Figure 0.3: Esempio di Bit-vector encoding

Chiaramente, l'utilizzo di questo approccio risulta più oneroso in termini di spazio utilizzato rispetto a RLE, ma consente di effettuare facilmente operazioni di **AND**, **OR** e **NOT** tra vettori, il che lo rende particolarmente adatto per operazioni di filtro e selezione.

Dictionary Encoding

In questo algoritmo di compressione, ogni valore viene rimpiazzato ogni valore con valori che siano rappresentabili in maniera più efficiente. Per esempio, effettuando le seguenti associazioni: $1 \longrightarrow 205$, $2 \longrightarrow 207$ e $3 \longrightarrow 587$ otteniamo la rappresentazione di Figure 0.4.

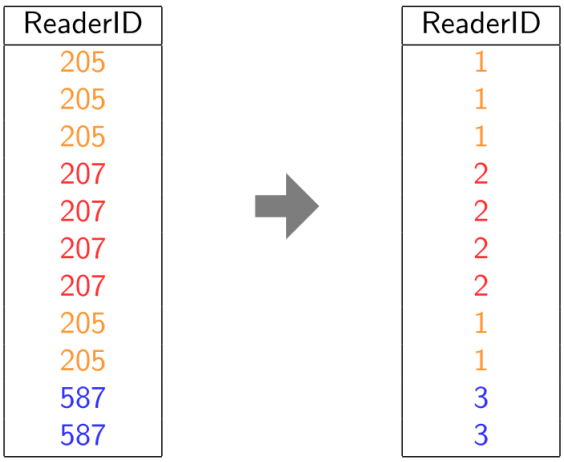

Figure 0.4: Esempio di dictionary-encoding

Questo approccio risulta particolarmente vantaggioso nel momento in cui i valori presenti (e ripetuti) all'interno di una colonna sono particolarmente lunghi o complessi, come stringhe di testo o strutture dati complesse. Chiaramente nel momento in cui si voglia operare sulla colonna sarà necessario andare a ricostruire i valori originali utilizzando il dizionario.

È anche possibile andare ad utilizzare questo approccio per **sequenze** di valori, andando a mappare intere sequenze a valori più compatti; questo approccio risulta di complicata implementazione ma può portare a tassi di compressione molto elevati. Per esempio effettuando il mapping $1 \longrightarrow 1002, 1010, 1004$, $2 \longrightarrow 1008, 1006$ otteniamo la rappresentazione di Figure 0.5

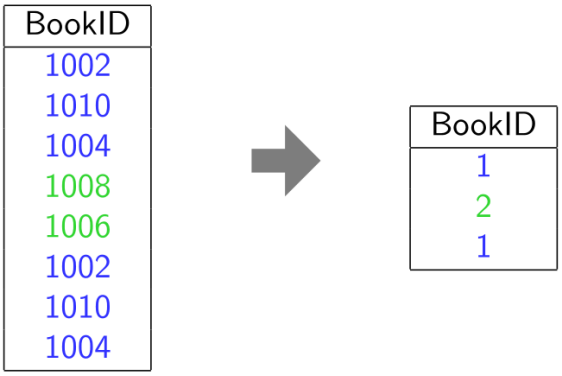

Figure 0.5: Esempio di dictionary-encoding per sequenze

Frame of Reference Encoding (FoR)
In questo algoritmo di compressione, viene scelto un **valore di riferimento** (es. minimo, mediano, massimo, ...) e ogni valore della colonna viene rappresentato come un **offset** (positivo o negativo) rispetto al riferimento.

È possibile scegliere che ci sia un valore massimo di offset dopo il quale i valori non vengano più compressi e marcati con un simbolo speciale. Questo potrebbe essere particolarmente utile per identificare e gestire i **valori anomali** (outliers). Figure 0.6 mostra un esempio del funzionamento di questo algoritmo.

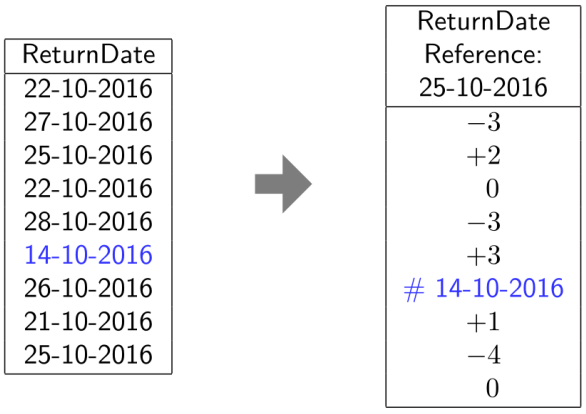

Figure 0.6: Esempio di Frame-of-Reference encoding con valori oltre l'offset massimo

Una variante di questa tecnica di encoding è data dal **differential encoding**, dove invece di memorizzare l'offset rispetto ad un valore predefinito, ogni valore viene memorizzato come differenza rispetto al valore precedente. Questo approccio potrebbe aiutare specialmente nella riduzione del numero di valori che sono fuori dall'offset massimo. Figure 0.7 mostra un esempio del funzionamento di questo algoritmo.

ReturnDate
22-10-2016
27-10-2016
25-10-2016
22-10-2016
28-10-2016
14-10-2016
26-10-2016
21-10-2016
25-10-2016

ReturnDate
22-10-2016
+5
−2
−3
+6
14-10-2016
−2
−5
+4

Figure 0.7: Esempio di encoding differenziale con valori oltre l'offset massimo

> **Remark**
>
> Si noti come in Figure 0.7 il valore successivo al valore fuori dall'offset sia preso a partire dal primo valore 'valido'. Questo potrebbe tornare particolarmente utile per trovare outlier.

Altre tecniche di compressione

Ci possiamo trovare davanti a situazioni in cui abbiamo colonne con molti **valori nulli**, se vogliamo effettivamente essere in grado di ricostruire colonne con questi valori è comunque necessario memorizzarli. Esistono diverse tecniche per andare a comprimere questi valori:

- **Bitmap encoding**: viene creato un vettore di bit dove ogni bit indica se il valore in quella riga è nullo o meno. I valori nulli non vengono memorizzati
- **Sparse encoding**: vengono memorizzate solo le righe che contengono valori non nulli, insieme ai loro identificatori di riga
- **Run-length encoding per nulli**: viene applicato un algoritmo di run-length encoding specifico per i valori nulli, memorizzando le sequenze di valori nulli in modo compatto
- **Dictionary encoding per nulli**: viene creato un dizionario che include un'entrata speciale per i valori nulli, permettendo di rappresentarli in modo compatto

È possibile andare a rappresentare un **documento complesso** (es. JSON) utilizzando un approccio colonnare. Possiamo vedere il documento come un albero e andare a memorizzare *una colonna per ogni path root-leaf* dell'albero. In questo modo è possibile andare a memorizzare in maniera efficiente anche documenti con strutture molto complesse. Questa tecnica è nota con il nome di **column striping**.

Query su Dati Compressi

In questa sezione andiamo a concentrarci su quale approccio sia necessario per effettuare query su dati compressi. In alcune situazioni è possibile effettuare le operazioni di cui abbiamo bisogno direttamente sui dati compressi. Ne vediamo di seguito un esempio.

> **Example: Query su dati compressi**
>
> Supponiamo di avere effettuato una compressione *run-length encoding* sulla colonna `ReaderID` di una tabella `BookLending`, ottenendo la seguente compressione:
>
> $$((205, 0, 3), (207, 4, 2), (206, 6, 1))$$
>
> Supponiamo di avere la seguente query in linguaggio naturale: *'Quanti libri ha ciascun lettore?'*, che si può tradurre in SQL tramite il seguente codice:

```sql
SELECT ReaderID, COUNT(*)
FROM BookLending
GROUP BY ReaderID
```

Utilizzando l'encoding run-length è possibile semplicemente ritornare la somma dei run-length per ogni valore di `ReaderID`, ottenendo così il risultato della query senza dover de-comprimere i dati:

$$\{(205, 4), (207, 2)\}$$

Remark

Si noti come nell'esempio in precedenza, se non avessimo avuto a disposizione i run-length, avremmo dovuto de-comprimere l'intera colonna e poi effettuare il conteggio per ogni `ReaderID`, il che sarebbe stato molto più costoso in termini di tempo di calcolo. Questo ci suggerisce come sia importante scegliere algoritmi di compressione adeguati al tipo di query che ci aspettiamo di dover eseguire sui dati.