Advanced Data Management

Federico Segala

Anno Accademico: 2025-2026
Appunti del corso di Advanced Data Management
prof. Claudio Silvestri

Sommario

1. Introduzione

Le basi di dati sono elementi fondamentali in molti aspetti della tecnologia quotidiana. Ogni giorno infatti, è prodotta, memorizzata e elaborata una **immensa quantità di dati**.

Un **database management system** che funzioni in maniera corretta è dunque cruciale per rendere queste attività il più semplici ed efficaci possibili. Questo capitolo si occupa di introdurre *principi* e *proprietà* che un database dovrebbe soddisfare.

1.1. Proprietà di una Base di Dati

Dal momento che lo storage di dati è cruciale, un database dovrebbe garantire le proprietà che andiamo di seguito a definire.

- **Gestione dei dati**: Una base di dati non si occupa solo del salvataggio, ma deve supportare operazioni che permettano recupero, ricerca, e aggiornamento dei dati. Per fare ciò spesso è necessario avere delle interfacce tramite le quali sia possibile comunicare con la base di dati. Un altro aspetto importante è quello del supporto alle transazioni, ossia insiemi di operazioni atomici, non interrompibili.
- **Scalabilità**: La quantità di dati processati è di solito enorme. Elaborare questa mole di dati è fattibile solamente **distribuendoli** in una rete e garantendo un alto livello di parallelismo. La necessità in questo caso è di *adattarsi al workload corrente* del sistema e allocare risorse di conseguenza.
- **Eterogeneità**: Nel momento in cui andiamo a memorizzare dati questi non sono tipicamente nella forma corretta per essere memorizzati in forma relazionale; I dati possono essere memorizzati in maniera **strutturata** ma non solo. Possono infatti essere **semi-strutturati**, altre forme tipiche sono strutture ad **albero** (XML) o a **grafo**, nel peggiore dei casi, si può avere un dato che è completamente non strutturato.
- **Efficienza**: La maggioranza delle applicazioni hanno bisogno di sistemi molto veloci in modo da riflettere nel minor tempo possibile i cambiamenti (real time applications).
- **Persistenza**: Lo scopo principale di una base di dati è quello di fornire storage a lungo termine dei dati. Ci sono delle eccezioni a questo, casi in cui solo parti dei dati necessità permanenza a lungo termine, mentre altri sono più *volatili*; questo comportamento è chiamato **persistenza selettiva**.
- **Affidabilità**: Un buon sistema è tipicamente in grado di prevenire la perdita di dati o l'avvenimento di distorsioni degli stessi. In pratica ciò cu sui ci si concentra è l'**integrità** dei dati. Ciò avviene tipicamente tramite *ridondanza fisica* e *replicazione*.
- **Consistenza**: È importante che la base di dati garantisca che non siano presenti dati contradditori o errati nel sistema. Ciò si ottiene tipicamente tramite chiavi primarie, integrità referenziale e aggiornamento automatico delle repliche dei dati.
- **Non Ridondanza**: La ridondanza fisica è cruciale per garantire affidabilità, la duplicazione di valori (*ridondanza logica*) è invece da evitare per quanto possibile. Ciò aumenta inutilmente il consumo di spazio e la possibilità di *anomalie*.

- **Supporto multi-utente**: Nei sistemi moderni è spesso richiesto il supporto all'accesso concorrente alle risorse da parte di più utenti o applicazioni.

Tutte queste caratteristiche ci consentono di formalizzare nel modo più completo possibile cosa sia una database management system (DBMS) come riporta Definizione 1.1°

Definizione 1.1 (Base di Dati)

Una base di dati è un sistema che consente di gestire grandi quantità di dati eterogenei, in maniera efficiente, persistente, affidabile, consistente e non ridondante. È inoltre in grado di supportare accesso concorrente da parte di più utenti.

♣

Tipicamente è complicato che una base di dati supporti tutte le caratteristiche elencate di sopra; si rivela dunque fondamentale un'**analisi** dettagliata dei **requisiti** di ogni caso d'uso per rendere il più ponderata possibile la scelta del sistema da utilizzare.

1.2. Componenti di un Database

Il componente software che si fa carico di tutte le operazioni sulla base di dati è il **database management system** (DBMS). Figura 1.1 illustra come molti altri componenti nel sistema operativo vadano a interagire tra di loro e con questo importante elemento.

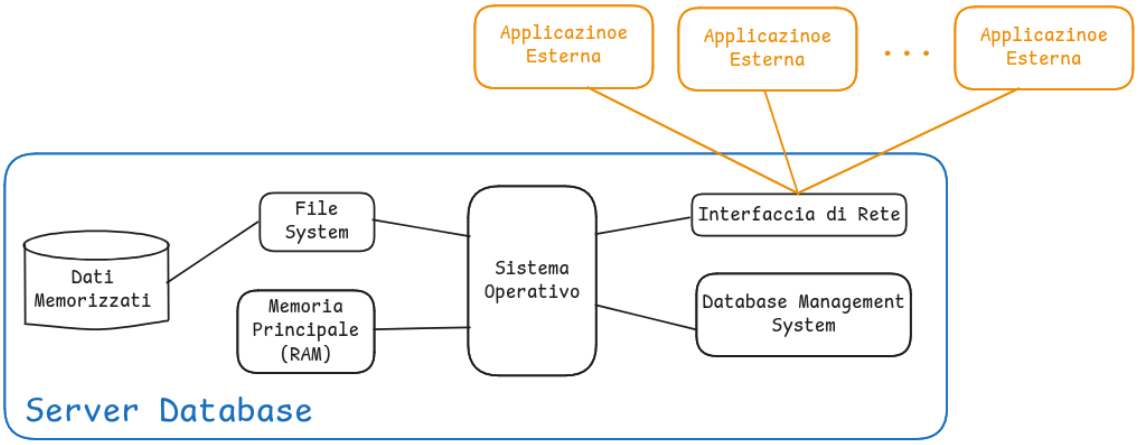

Figura 1.1: Interazioni del DBMS

A seconda delle necessità che vengono specificate dai requisiti, è possibile andare a concentrarsi su una specifica implementazione. Andando oltre alle peculiarità associate ad ogni diverso prodotto, è possibile identificare i seguenti elementi comuni a tutti i DMBS:

- **Gestione della memoria persistente**: si occupa di salvare i dati nel file system
- **Gestione del buffer**: si occupa, collaborando con il gestore della memoria, di effettuare *swap in/out* di varie pagine della memoria persistente in base a ciò che è richiesto dall'operazione che si sta eseguendo

Il gestore del buffer potrebbe sembrare molto simile al *sistema* di paginazione utilizzato sui comuni sistemi operativi. La differenza consiste nel fatto che il gestore del buffer è a conoscenza del tipo di operazione che è in esecuzione all'interno del database.

- **Strutture dati** per la memoria secondaria: si usano tipicamente per ottenere maggiore efficienza rispetto allo swap in/out delle pagine
- **Metodi di accesso**: consistono in un modo per accedere ai dati in memoria secondaria seguendo specifici *pattern* in base all'operazione che è necessario eseguire
- **Gestione delle transazioni**: si occupa di eseguire sequenze di operazioni in maniera atomica, mantenendo la consistenza del sistema
- **Gestione della concorrenza**: si occupa di garantire che molti processi siano abilitati ad accedere al database nello stesso momento. In linea di massima garantisce che le richieste parallele seguano uno schedule sequenziale

Per fare un esempio di ciò, immaginiamo che sia necessario gestire in maniera concorrente le seguenti transazioni: T_1, T_2, T_3; ci sono $(3!)$ 6 possibili scheduling sequenziali che possiamo scegliere di seguire per soddisfarle:

$$\begin{array}{ll} \bullet\ T_1, T_2, T_3 & \bullet\ T_1, T_3, T_2 \\ \bullet\ T_2, T_1, T_3 & \bullet\ T_2, T_3, T_1 \\ \bullet\ T_3, T_2, T_1 & \bullet\ T_3, T_1, T_2 \end{array}$$

compito del gestore della concorrenza è quello di fare in modo che il risultato del sistema dopo aver eseguito in concorrenza le tre transazioni sia equivalente al risultato di uno casuale degli scheduling sequenziali di sopra riportati.

- **Operatori fisici**: ne esistono alcuni che sono comuni a più famiglie di basi di dati, altri che sono specifici di singole famiglie. Si tratta di operazioni rese disponibili tramite API che quando eseguite su un insieme di dati garantiscono un certo risultato

La maggior parte di questi operatori fisici, nel caso di database relazionali, sono in diretta corrispondenza con operatori dell'*algebra relazionale*; ad esempio, operazioni di *filtering* corrispondono alla clausola `WHERE`, mentre alcune operazioni di *proiezione* corrispondono alla clausola `SELECT`.

- **Ottimizzatore delle query**: ipotizziamo di voler effettuare un'operazione di `JOIN`. Sappiamo che esistono molte variante di questa operazione, il compito di questa componente è quello di scegliere il tipo di implementazione da utilizzare per rendere il più efficiente possibile la valutazione della query in esame

- **Frammentazione dei dati**: esistono casi in cui i dati non sono salvati in un'unica posizione (potrebbero essere salvati su macchine diverse, o sulla stessa macchina ma in file system separati, oppure sullo stesso file system ma non sullo stesso disco). In questi casi è necessario decidere come *partizionare* i dati

- **Replicazione e sharding**: nel caso in cui i dati siano memorizzati su **sistemi diversi** è necessario decidere cosa e dove *replicare* i dati, questo è un aspetto comune a praticamente tutte le diverse famiglie di database

- **Gestore della consistenza**: si occupa di fare in modo che tutte le repliche di uno stesso dato salvate su uno o diversi sistemi siano tra loro consistenti

- **Esecuzione distribuita**: per consentire ai sistemi di essere il più efficienti possibili, nel momento in cui le informazioni sono salvate in maniera frammentaria, è possibile sfruttare il calcolo distribuito in parallelo, in modo da abbattere i costi di esecuzione
- **Gestione dei dati in streaming**
- **Code di messaggi**: si tratta di un meccanismo pensato ad hoc per la gestione di esecuzione distribuita e dei dati in streaming

1.3. Database Management System Relazionali

Tutti i database relazionali sono basati sul **modello dei dati relazionale**. Andiamo di seguito a definirne le varie peculiarità:

- Un database è un *insieme di tabelle* ognuna delle quali è caratterizzata da un **nome** che nello specifico è chiamato **relation symbol**
- L'intestazione della relazione va ad identificare i nomi delle *colonne* che nello specifico sono chiamati **attribute names** insieme al *dominio* dal quale ciascun attributo può prendere valore
- L'insieme delle *righe* (**tuple**) di una tabella ne vanno a definire il contenuto

Di seguito vedremo le definizioni di **schema relazionale** e **schema della base di dati** con alcuni esempi per meglio comprendere questi concetti fondamentali.

Definizione 1.2 (Schema Relazionale)

È possibile definire lo **schema relazionale** di una base di dati tramite i seguenti oggetti:

- Simbolo di relazione R
- Insieme degli attributi $A_1, ..., A_n$
- Insieme delle dipendenze locali Σ_R

Possiamo unire gli oggetti di cui sopra tramite la seguente formula:

$$R = (\{A_1, ..., A_n\}, \Sigma_R) \tag{1}$$

♣

Di seguito possiamo osservare la come lo schema relazionale possa essere visto dal punto di vista di una tabella:

Simbolo di relazione R	Attributo A_1	Attributo A_2	Attributo A_3
tupla t_1	v_{11}	v_{12}	v_{13}
tupla t_2	v_{21}	v_{22}	v_{23}

Esempio: Modello relazionale

Di seguito mostriamo un'istanza del modello relazionale precedentemente illustrato:

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

Definizione 1.3 (Schema della Base di Dati)

È possibile andare a definire lo **schema della base di dati** tramite i seguenti oggetti:

- Simbolo della base di dati D
- Insieme di schemi relazionali $R_1, ..., R_m$
- Insieme di dipendenze globali Σ

Possiamo unire gli oggetti di cui sopra tramite la seguente formula:

$$D = (\{R_1, ..., R_m\}, \Sigma) \tag{2}$$

♣

Esempio: Schema di una Base di Dati

- Schema relazionale 1:

$$\text{Book} = (\{\text{BookID}, \text{Author}, \text{Title}\}, \Sigma_{\text{Book}})$$

- Schema relazionale 2:

$$\text{BookLending} = (\{\text{BookID}, \text{ReaderID}, \text{ReturnDate}\}, \Sigma_{\text{Book}})$$

- Schema relazionale 3:

$$\text{Reader} = (\{\text{ReaderID}, \text{Name}\}, \Sigma_{\text{Reader}})$$

- Schema della base di dati:

$$\text{Library} = (\{\text{Book}, \text{BookLending}, \text{Reader}\}, \Sigma)$$

Sia in Definizione 1.2° che in Definizione 1.3° compare la nozione di **dipendenza**; è però necessario chiarire le differenze tra queste:

- quando il concetto di dipendenza compare con un pedice, per esempio Σ_R, ci stiamo riferendo a **dipendenze intra-relazionali**; cioè che si verificano all'interno di una tabella.

Un esempio di dipendenze intra-relazionali sono le *dipendenze funzionali*, più in particolare le dipendenze indotte da attributi *chiave*: BookID $\longrightarrow$ BookID, Author, Title è una dipendenza funzionale all'interno della relazione Book

- quando le dipendenze compaiono senza pedice, significa che sono **globali**, anche dette **inter-relazionali**

Un esempio di queste dipendenze sono *dipendenze di inclusione* su particolari chiavi esterne: BookLending.BookID $\subseteq$ Book.BookID o ancora BookLending.ReaderID $\subseteq$ Reader.ReaderID.

1.3.1. Progettazione di una Base di Dati

Una volta presi in esame la situazione da rappresentare e i requisiti da questa richiesti, è possibile iniziare con la progettazione della base di dati. Dal momento che molti contesti presentano molte complicazioni e requisiti specifici, è bene avere un quadro il più generale possibile di ciò che si renderà necessario implementare; per questo motivo possiamo dividere la progettazione di un database in tre fasi fondamentali:

- definizione di un **modello concettuale**: serve a modellare ad alto livello la situazione presa in esame; tipicamente vengono impiegati i diagrammi entità-relazione.

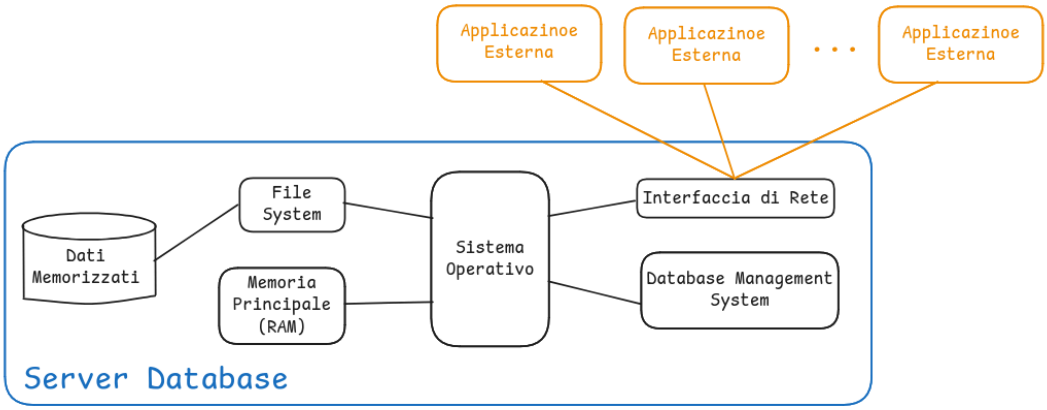

Figura 1.2: Diagramma Entità-Relazione di una libreria

- *traduzione* dello schema relazionale in un **modello logico**: il processo da seguire per la traduzione è piuttosto semplice e standard; infatti tipicamente ogni entità viene di solito mappata in una relazione, così come ogni relazione, in base alla sua cardinalità può essere tradotta in maniera differente

Alcuni design per una base di dati possono essere problematici. Di seguito andiamo ad indicare alcune forme di inconsistenza che possono presentarsi:

- Alcune tabelle potrebbero contenere troppi valori, o addirittura valori **duplicati**
- Potrebbero verificarsi anomalie nel momento in cui andiamo a manipolare i dati:
  - **Anomalie di inserimento**: nel momento in cui andiamo ad inserire i dati abbiamo bisogno di tutti i valori ma alcuni potrebbero essere ancora sconosciuti
  - **Anomalie di cancellazione**: nel momento in cui andiamo a cancellare una tupla, potremmo andare a cancellare informazioni di cui abbiamo ancora bisogno in altri record di altre relazioni
  - **Anomalie di aggiornamento**: quando i dati sono memorizzati in maniera ridondante, questi devono essere modificati per tutte le loro occorrenze

Le problematiche e le anomalie sopra elencate sono tipicamente ammortizzate applicando tecniche di **normalizzazione** della base di dati. L'obiettivo della normalizzazione è quello di distribuire i dati in maniera omogenea tra le tabelle. In base al tipo di normalizzazione che andiamo a garantire riusciamo a prevenire diversi tipi di anomalia:

- **1° Forma Normale**: non vengono ammessi attributi multivalore o composti
- **2° Forma Normale**: tutti gli attributi non chiave sono completamente dipendenti dagli attributi chiave, in altre parole non deve esistere un sottoinsieme degli attributi chiave che può essere usato per derivare attributo non chiave
- **3° Forma Normale**: tutti gli attributi non chiave sono direttamente dipendenti dagli attributi chiave, in altre parole si dice che non sono ammesse dipendenze transitive
- **4°, 5° Forma Normale** e **Forma Normale di Backus-Naur** sono altri tipi di forma normale ma non così comuni e comunque fuori dallo scopo di questo corso

1.3.2. Query Relazionali

Una volta che i dati sono memorizzati all'interno della nostra base di dati, possiamo chiederci in quale modo ottenere informazioni da questi. A questo scopo abbiamo a disposizione degli strumenti che sono noti con il nome di **query**.

Le query sono strumenti che ci consentono di effettuare diversi tipi di operazioni sui dati:

- Specificare condizioni per selezionare solo tuple rilevanti
- Restringere tabelle ad un sottoinsieme di attributi
- Combinare valori provenienti da diverse tabelle

Esistono diversi linguaggi per effettuare query a una base di dati: *calcolo relazionale*, *algebra relazionale*, *SQL*. Di seguito andiamo ad elencare alcuni operatori dell'algebra relazionale:

- **Proiezione** π: utilizzata per restringere una tabella ad un sottoinsieme di attributi
- **Selezione** σ: utilizzata per selezionare solo alcune tuple di una tabella
- **Rinominazione** ρ: utilizzata per cambiare nomi ad un attributo
- **Operazioni insiemistiche**: unione $\cup$, differenza $-$, intersezione $\cap$
- **Join naturale** $\bowtie$: utilizzato per combinare due tabelle sulla base di attributi comuni
- **Operatori di join avanzati** come θ - join, equi-join, ...

Le query relazionali,che si possono vedere come una catena di applicazioni di opperatori relazionali, possono essere visualizzate in strutture ad albero, questo tipo di visualizzazione porta con se alcuni vantaggi:

- Mostra l'ordine di valutazione di ogni operazione
- È utile nel contesto dell'ottimizzazione delle query

Esempio: Alberi equivalenti per una query che lista il nome di tutti i lettori dei libri prestati che abbiano ReturnDate < 20/10/2016

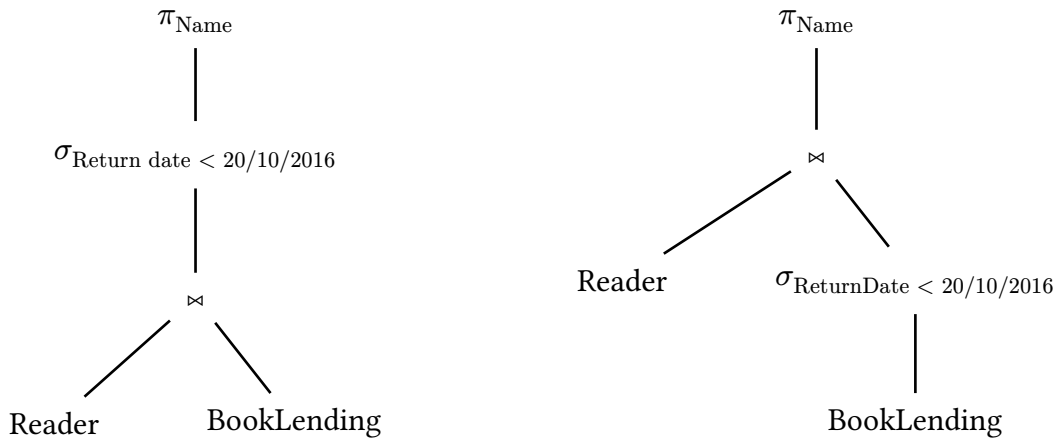

Figura 1.3: Due diversi piani di query per la stessa richiesta in algebra relazionale.

Possiamo osservare come Figura 1.3 illustri due modi alternativi di eseguire la stessa query relazionale. Osservando attentamente l'ordine tra *join* e *selezione*, è possibile notare che nell'albero di destra otteniamo una query più efficiente, dal momento che ci troveremo a effettuare un join dove una delle due tabelle da unire è stata prima filtrata. In questo modo ci troveremo a dover effettuare un confronto con molti meno record rispetto alla rappresentazione di sinistra.

1.3.3. Transazioni e Gestione della Concorrenza

Quando andiamo a modificare i dati all'interno di una base di dati, possiamo andare incontro a diverse tipologie di problematiche. Di seguito andiamo ad elencarne alcune:

- **Integrità logica dei dati**: dobbiamo assicurarci che tutti i valori scritti siano corretti e che siano effettivamente il risultato atteso dall'esecuzione di un'operazione
- **Integrità fisica e recovery**: dobbiamo garantire la persistenza dei dati assieme alla possibilità di recuperarli nel caso in cui si verifichino dei crash di sistema
- **Gestione di più utenti**: dobbiamo permettere agli utenti di operare in maniera concorrente sulla stessa base di dati senza che ci siano interferenze

Tutte le questioni sopra elencate possono essere indirizzate tramite l'impiego di **transazioni**.

Definizione 1.4 (Transazione)

Una **transazione** è una sequenza di operazioni di *lettura e scrittura* su una base di dati con le seguenti proprietà:

- Deve essere trattata come **entità atomica** di esecuzione
- Deve portare la base di dati da uno stato consistente ad un nuovo stato anch'esso consistente

L'esempio più tipico di una transazione è quello di un trasferimento di denaro da un conto bancario ad un altro. Di seguito specifichiamo alcune delle proprietà che le basi di dati devono rispettare affinché possiamo affermare che gestiscano le transazioni correttamente:

- **Atomicità**: data una transazione possiamo eseguire tutte le operazioni di questa, oppure nessuna, non è possibile eseguire una transazione in modo *parziale*
- **Consistenza**: dopo l'esecuzione di una transazione tutti i valori nella base di dati devono essere corretti rispetto ai vincoli e alle dipendenze intra-inter relazionali
- **Isolamento**: transazioni concorrenti di utenti differenti non devono interferire tra loro
- **Durabilità**: è necessario che i risultati di una transazione rimangano persistenti anche a seguito di possibili crash di sistema. Per garantire questa proprietà di solito si utilizza un **transaction log**, nel quale tutte le transazioni vengono registrate

Le proprietà sopra elencate sono anche dette **ACID**, utilizzando le loro iniziali per formare l'acronimo.

1.3.4. Problematiche del Modello Relazionale

L'impiego di modelli relazionali può portare con sé alcune problematiche dovute intrinsecamente a come vengono impiegate tabelle e relazioni. Andiamo ad elencare alcune problematiche di seguito:

- **Overloading semantico**: il modello relazionale rappresenta sia entità che relazione tramite l'utilizzo di *tabelle*, non esiste infatti in modo per rappresentare separatamente i due concetti
- **Struttura dati omogenea**: il modello relazionale assume omogeneità sia orizzontale che verticale. L'omogeneità orizzontale implica che tutte le tuple hanno valori per gli stessi attributi; quella verticale si riferisce al fatto che, data una colonna, i suoi valori provengono tutti dallo stesso dominio. Inoltre ogni cella può contenere soltanto valori atomici, il che potrebbe risultare limitante in alcuni contesti
- **Supporto limitato alla ricorsione**: è molto complicato andare a definire query ricorsive in SQL; nel caso ad esempio in cui ci trovassimo a dover lavorare su strutture a grafo sarebbe veramente complicato utilizzare un modello relazionale

1.4. Nuovi Requisiti

Dopo aver descritto in maniera approfondita tutte le peculiarità di SQL e del modello relazionale che va ad implementare, spostiamo la nostra attenzione su dei *nuovi requisiti* che situazioni odierne ci portano a dover soddisfare.

Il primo di questi è sicuramente legato a dati che hanno bisogno di essere organizzati in **strutture** sempre più **complesse** come ad esempio nei social network.

Se un tempo le operazioni di lettura erano molto più frequenti rispetto alle operazioni di scrittura, il mondo moderno e l'utilizzo di nuove tecnologie ci pongono davanti ad un cambio di paradigma dove spesso è anche necessario andare a **scrivere in maniera frequente** sulle nostre basi di dati.

Dal momento che il mondo contemporaneo è sempre più *data-centric*, è necessario dover gestire una quantità sempre maggiore di dati, il che sarebbe impossibile utilizzando una singola macchina, rendendo necessario **distribuire i dati** su molti server interconnessi (*cloud storage*).

Tutte le esigenze di sopra aprono la strada all'utilizzo di un nuovo approccio, quello **NoSQL**, nel quale alcune proprietà e garanzie del modello relazionale vengono trascurate al fine di provare a soddisfare in maniera migliore i nuovi requisiti. Di seguito elenchiamo alcune delle differenze tra gli approcci NoSQL e il tradizionale SQL:

- Il modello dei dati potrebbe essere diverso da quello tradizionale basato sulle tabelle
- Accesso programmatico alla base di dati o con strumenti diversi da SQL
- Capacità di gestire modifiche allo schema dei dati
- Capacità di gestire dato senza uno schema specifico
- Supporto alla distribuzione dei dati
- Requisito di aderenza alle proprietà ACID che viene alleggerito, specialmente in termini di consistenza, rispetto ai DBMS tradizionali

2. Key-Value Stores e Document Databases

Nell'ultimo decennio i database relazionali sono stati particolarmente apprezzati grazie alla loro flessibilità; purtroppo non sono noti per le loro **performance**. Come accennato nell'ultima parte del capitolo precedente, gli importanti avanzamenti tecnologici degli ultimi anni hanno portato alla luce delle forti limitazioni legate alle caratteristiche dei sistemi relazionali.

L'idea è dunque quella di passare a dei sistemi che siano in generale meno flessibili rispetto a sistemi relazionali sotto alcuni punti di vista, ma che possano adattarsi meglio e con maggiore efficienza ai casi d'uso nei quali è necessaria la loro applicazione.

2.1. Key-Value Stores

L'idea alla base di questo approccio è molto semplice: viene costruito un **array associativo permanente**. Come il nome suggerisce, gli elementi chiave di un array associativo sono una **chiave** e **valori** associati alla chiave; volendo fare un paragone con i linguaggi di programmazione, possiamo associare questo concetto a quello di *dizionario in Python* e a quello di *hashMap in Java*. Di seguito andiamo ad elencare alcune delle proprietà fondamentali:

- È possibile accedere a valori (o cancellarli) tramite l'utilizzo delle *chiavi*
- È possibile inserire coppie chiave-valore arbitrarie senza che queste aderiscano necessariamente ad uno schema (**schemaless**)
- I valori possono avere tipi di dato molteplici (liste, stringhe, valori atomici, array, ...)
- È un sistema molto semplice ma veloce: grazie alla semplicità della struttura dati, non abbiamo bisogno di un query language molto avanzato per accedere ai dati. Si tratta di un'alternativa ottimale nel caso di applicazioni *data intensive*.

Proprio riguardo all'ultimo punto, è necessario specificare che il compito di combinare più coppie chiave-valore in oggetti complessi è tipicamente responsabilità dell'applicazione che si interfaccia con il sistema. Alcuni esempi molto comuni di questo approccio sono Amazon Dynamo, Riak e Redis

2.1.1. Map-Reduce

In generale, ci si riferisce a MapReduce come un approccio di programmazione che consente di processare enormi quantità di dati in parallelo sfruttando diversi cluster di calcolo dividendo grandi operazioni in piccoli passi di map e reduce. Nell'ambito dei key-value stores si può vedere la procedura divisa nei seguenti passaggi:

- **Splittare** l'input e iterare sulle coppie chiave-valore in sottinsiemi disgiunti
- Calcolare la funzione di **map** su ognuno dei sottoinsiemi splittati
- Raggruppare tutti i valori intermedi per chiave (**shuffle**)
- Iterare su tutti i gruppi applicare **reduce** in modo da riunire i vari gruppi

Figura 2.4 illustra chiaramente il funzionamento dei vari passaggi necessari al funzionamento dell'algoritmo MapReduce.

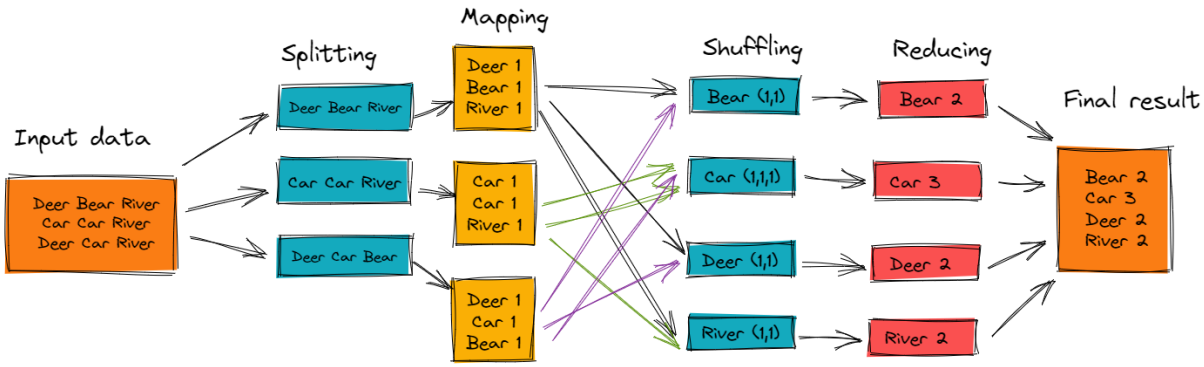

Figura 2.4: Esempio di applicazione dell'algoritmo MapReduce per contare le occorrenze di ogni parola all'interno di un input

È possibile notare i seguenti aspetti:

- L'approccio è estramamente adatto alla **parallelizzazione**, infatti i passaggi di mapping e di riduzione possono essere eseguiti in parallelo, ad esempio lanciando un processo di map per ogni «frase» o, nel caso dell'esempio, ogni n parole e un processo di *reduce* per ogni parola
- È possibile sfruttare la **località** dei dati in modo da processare i dati direttamente sulla macchina che li sta «ospitando» in modo da ridurre il più possibile il traffico sulla rete.
- È possibile migliorare ulteriormente quanto presente in Figura 2.4 applicando una procedura di **combinazione**, che consenta di combinare i risultati intermedi invece di mandarli alla procedura di riduzione in formato grezzo, tuttavia non è sempre garantito che questa operazione sia implementata sui sistemi che scegliamo di utilizzare

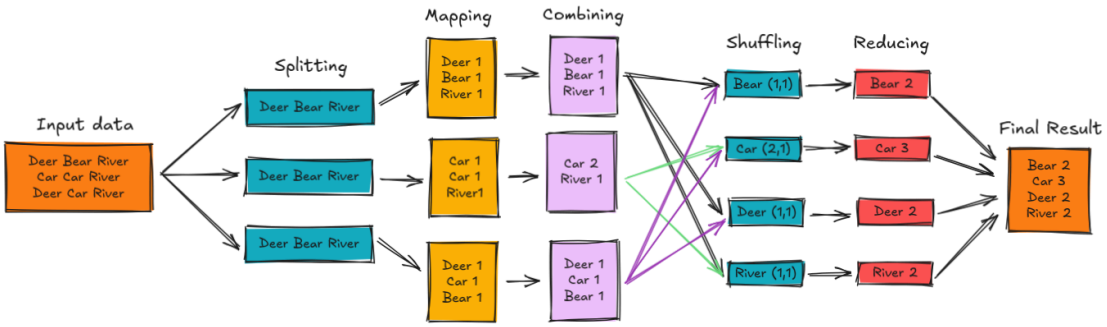

Figura 2.5: Applicazione della funzionalità di combine al metodo MapReduce

2.2. Document Database

Se i key-value store sono metodi molto semplici per memorizzare dati, questo approccio a volte può risultare fin troppo semplice: memorizzare soltanto dati primitivi a volte è fin troppo riduttivo per quelle che potrebbero essere le necessità di un'applicazione.

Per questo motivo nascono i **document database**, i quali permettono di memorizzare documenti in un formato di **testo strutturato** (JSON, XML, YAML, ...). Anche in questo caso ogni documento è identificato da una *chiave univoca*, mostrando quindi una forte correlazione al concetto di key-value store, ma i dati memorizzati hanno un requisito in più sulla loro struttura. Questo è spesso molto utile in quanto ci consente di effettuare validazione dei dati, per esempio tramite XML o JSON schema.

2.2.1. JavaScript Object Notation

Per lo scopo di questo corso non andremo a soffermarci su database di documenti che utilizzano XML per dare struttura ai propri documenti, per semplicità sceglieremo di concentrare la nostra attenzione su quelli che memorizzano oggetti JSON. Per fare ciò è necessario andare a comprendere quali siano le peculiarità di questo linguaggio:

- Si tratta di un **formato testuale** di facile lettura per rappresentazione di strutture dati
- Ogni documento JSON è sostanzialmente un annidamento di coppie **chiave-valore** separate da un simbolo « `:` »
- Per dare struttura al documento vengono utilizzate le parentesi graffe « `{}` »
- La chiave è sempre definita tramite una **stringa**, mentre i valori possono essere vari:
  - floating point
  - stringhe unicode
  - booleani
  - array
  - oggetti

> Esempio: Semplice Descrizione JSON di un oggetto Person

```JSON
{
  "firstName": "Alice",
  "lastName" : "Smith",
  "age"      : 31,
}
```

> Esempio: Oggetto complesso con figli composti e array di valori

```JSON
{
  "firstName" : "Alice",
  "lastName"  : "Smith",
  "age"       : 31,
  "address"   : {
    "street"    : "Main Street",
    "number"    : 12,
    "city"      : "Newtown",
    "zip"       : 31141,
  },
  "telephone" :  [123456, 908077, 2782783],
}
```

Il linguaggio JSON presenta tuttavia alcune limitazioni:

- Non supporta referenze da un documento all'altro, dunque non è possibile adottare un meccanismo di foreign key come in SQL
- Non supporta referenze all'interno di uno stesso documento JSON

Esistono tuttavia alcuni strumenti per il processing di file JSON che supportano referenze basate su ID: per esempio, è possibile aggiungere una chiave `id` per l'oggetto persona e impostare un valore univoco per questo, per fare in modo di utilizzare tale `id` per riferirci all'oggetto di tipo persona appena costruito in altri oggetti.

2.2.2. MongoDB

Uno degli esempi più noti di document database è sicuramente **MongoDB**. Se volessimo andare a fare un confronto con un DBMS relazionale avremmo le seguenti differenze:

- Capacità di **scalare orizzontalmente** su più macchine
- Rispetto a un DBMS relazionale abbiamo una **miglior località dei dati**; questa proprietà è garantita proprio dal fatto che ogni oggetti contiene tutti i dati di cui ha bisogno, senza necessità di dover effettuare «join» con altre tabelle
- Mancanza di possibilità di far rispettare ai dati memorizzati uno **schema** con conseguente mancata possibilità di validare i dati in ingresso
- Mancata possibilità di eseguire operazioni di **join** per unire risultati
- Mancata possibilità di supportare **transazioni**

Similmente ad un database relazionale, è invece possibile operare query per il recupero di dati o la costruzione di indici sia primari che secondari per migliorare l'efficienza. Per semplicità andiamo di seguito a stabilire un mapping tra un DBMS relazionale e MongoDB:

RDBMS		MongoDB
Database		Database
Table, View		Collection
Row		Document (JSON, BSON)
Column		Field
Index		Index
Join		Embedded Document
Foreign Key		Reference
Partition		Shard

De-normalizzazione

L'aspetto più rilevante nell'utilizzo di questo modello risiede nella **semplicità** con cui è possibile rappresentare e gestire diverse strutture dati. Il concetto chiave a cui si deve questa immediatezza è la **de-normalizzazione**. Come mostrato in Figura 2.6, la de-normalizzazione semplifica la struttura della base di dati accorpando le informazioni che altrimenti sarebbero distribuite su più tabelle o entità.

Questa scelta, tuttavia, introduce un importante svantaggio: la **gestione delle modifiche ai dati**. Se, ad esempio, un utente compare in più contesti e si rende necessario aggiornare i dati relativi agli ordini a lui associati, sarà necessario applicare la modifica in **ogni copia** presente nel sistema. In caso contrario, la base di dati rischierebbe di diventare incoerente.

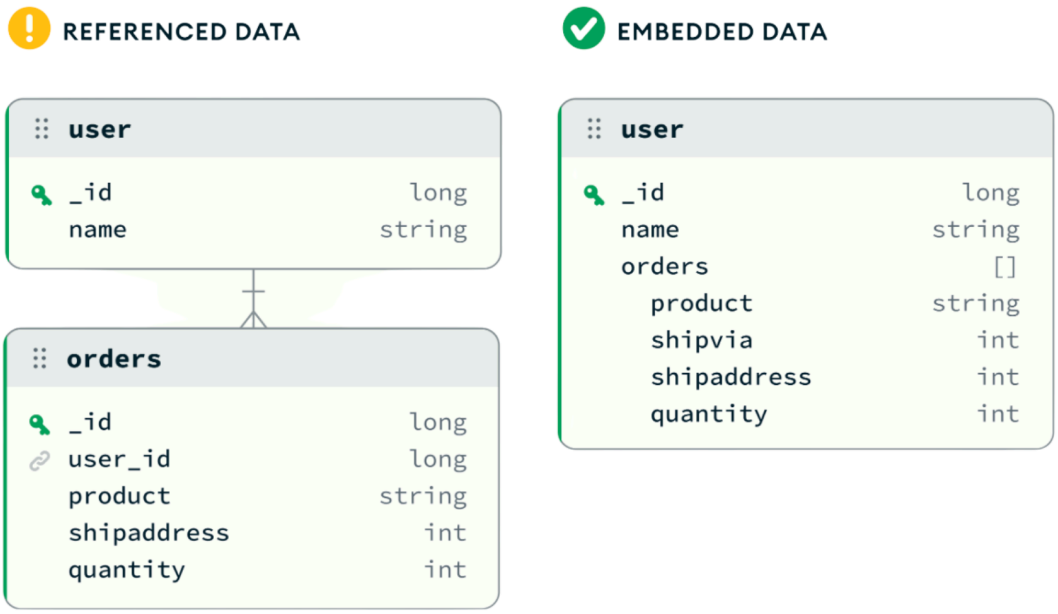

Figura 2.6: Esempio di de-normalizzazione: a sinistra una struttura normalizzata basata su due entità distinte, a destra una rappresentazione de-normalizzata della stessa relazione.

Salvataggio Atomico

Per quanto abbiamo citato che sia stato abbandonato il concetto di transazioni e delle loro proprietà «ACID», è comunque necessario anche in questo contesto andare a garantire consistenza, specialmente nel momento in cui più processi concorrenti vanno ad interrogare la base di dati. Per questo il paradigma adottato è quello del **salvataggio atomico**, che risulta comunque più semplice e rapido da effettuare rispetto all'esecuzione di una transazione.

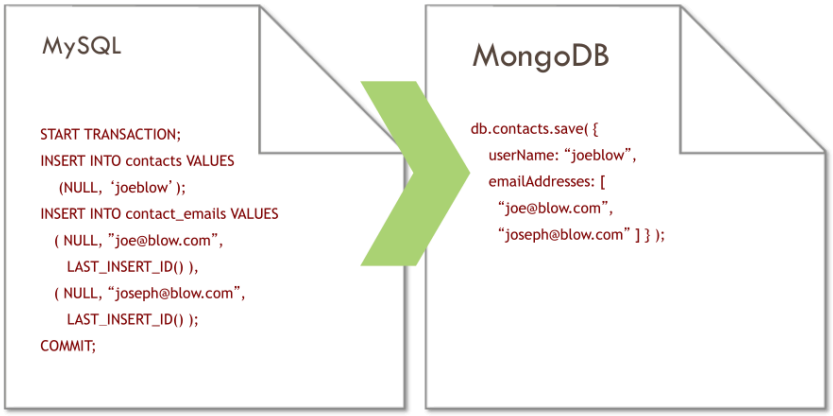

Figura 2.7: Esempio di Salvataggio atomico

Altre caratteristiche di MongoDB

Di seguito andiamo ad elencare altre caratteristiche peculiari di MongoDB che lo differenziano rispetto ad altri sistemi:

- Ogni documento JSON memorizzato deve essere provvisto di un campo identificativo con nome `_id`, se non fornito, questo campo viene creato in automatico dal sistema
- I dati vengono in realtà memorizzati in formato **BSON** che consiste in una rappresentazione binaria dei dati JSON per garantire maggiore efficienza e semplicità di manipolazione dei dati
- MongoDB è in grado di capire quali sono i dati ai quali sono richiesti accessi più frequenti e di **cachare** in memoria principale i loro valori, così da garantirne un accesso più rapido ed efficiente

CRUD

Questa sezione andrà ad illustrare i vari comandi che è possibile utilizzare per effettuare le operazioni CRUD su MongoDB:

Per quanto riguarda l'operazione di **create** abbiamo a disposizione i seguenti comandi:

- `db.collection.insert(<document>)`
- `db.collection.save(<document>)`
- `db.collection.update(<query>,<update>, {upsert: true})`: prova ad aggiornare un record ma se non esiste nulla che corrisponda alla `query` allora va ad inserire il valore che avrebbe dovuto aggiornare

> Esempio: Inserimento di un documento

```javascript
> db.user.insert({
  firstName : "John",
  lastName  : "Doe",
  age       : 39
})
```

Per quanto riguarda l'operazione di **read**, in modo simile a come facciamo in SQL andiamo a leggere tutti i dati che soddisfano una certa condizione:

- `db.collection.find(<query>, <projection>)`
- `db.collection.findOne(<query>, <projection>)`

> Esempio: Lettura di tutti gli elementi

```javascript
> db.user.find()

> result : {
    "_id"       : ObjectId("51.."),  // assegnato automaticamente
    "firstName" : "John",
```

```
6      "lastName"  : "Doe",
7      "age"       : 39
8    }
```

Si può notare come in questa occasione, dal momento che non sono stati passati parametri per `<query>`, il database abbia restituito tutti gli elementi della collection

Per quello che riguarda le operazioni di **update**:

- `db.collection.update(<query>, <update>, <options>)`

Esempio: Aggiornamento di un documento

```
1 > db.user.update(                                          JavaScript
2   {"_id": ObjectId("51.")}, // query per modificare specifico objectId
3   {
4     $set: {   // aggiornamento che si intende fare
5       age: 40, salary: 7000
6     }
7   }
8 )
```

Per ciò che invece riguarda le operazioni di **delete** abbiamo la seguente funzionalità:

- `db.collection.remove(<query>, <justOne>)`

Esempio: Eliminazione di un documento in base ad una query

```
1 > db.user.remove({                                         JavaScript
2   "firstName": /^J/     // regexp per identificare tutti i documenti
    dove firstName inizia per "J"
3 })
```

Proprietà ACID vs. BASE

Se abbiamo visto che nei database relazionali vengono tenute in forze considerazione proprietà *ACID* (atomicità, consistenza, isolamento, durevolezza); in sistemi come MongoDB è stato preferito richiedere l'aderenza ad un nuovo tipo di paradigma, più lasco, ma che consente maggior efficienza e meglio si adatta ai casi d'uso:

- **Basically Available**: il sistema rimane operazionale anche durante possibili crash parziali di sistema. Anche in questi casi dovrebbe essere possibile l'accesso alla base di dati, assicurando che il servizio continui ad essere disponibile. Si tratta di una proprietà fondamentale in sistemi che richiedono «constant uptime», come ad esempio applicazioni di e-commerce o applicazioni social media
- **Soft State**: questo concetto si riferisce all'idea che lo stato del database potrebbe cambiare nel corso del tempo, anche se non dovessero essere stati aggiunti o modificati

dati memorizzati. Queste modifiche sono tipicamente atte al raggiungimento di uno stato che sia consistente per tutti i nodi della base di dati

- **Eventually Consistent**: questo significa che dopo un aggiornamento non è necessario che le modifiche apportate alla base di dati siano rese note ad ogni istanza. Questo è particolarmente utile in un *contesto distribuito*, dove sarebbe altrimenti necessario aggiornare tutte le possibili istanze del dato aggiornato, che si trovano potenzialmente in più locazioni fisiche

2.2.3. Caso d'uso: Location-Based Application

Vogliamo costruire un'applicazione con le seguenti caratteristiche:

- Gli utenti deveono avere la possibilità di effettuare il *check-in*
- Gli utenti possono lasciare *note* o *commenti* riguardo una location

Per ogni **location** vogliamo le seguenti possibilità:

- Salvare il *nome*, l'*indirizzo* e dei *tag*
- Possibilità di memorizzare contenuti generati dagli utenti (*tips*, *note*)
- Possibilità di trovare altre location nei paraggi

Per quanto riguarda invece i **check-in** abbiamo i seguenti requisiti:

- Gli utenti dovrebbero essere in grado di effettuare il check-in
- Possibilità di generare *statistiche* sui check-in per ogni location

In primo luogo è necessario andare a definire le **collection** (tabelle) che andranno in qualche modo a rappresentare le entità coinvolte all'interno del sistema.

Collection `locations`

Per prima cosa andiamo a dare un rudimentale schema per la collection `locations`, che andrà a rappresentare le varie location del nostro sistema:

> Esempio: Locations v1 - Possibilità di filtrare per zipcode e per tags

JSON
```
location = {
   name: "10gen East Coast",
   address: "134 5th Avenue 3rd Floor",
   city: "New York",
   zip: "10011",

   tags: ["business", "offices"]
}
```

Di seguito andiamo a mostrare alcune query che sarà possibile effettuare su questa collection per andarne a visualizzarne i valori:

JavaScript
```
// 1.trova le prime 10 location con zip code 10011
db.locations.find({zip:"10011"}).limit(10)
```

```
3
4 // 2. trova le prime 10 location con tag business
5 db.locations.find({tag: "business"}).limit(10)
6
7 // 3. trova le location con zip code 10011 e tag business
8 db.locations.find({zip: "10011", tags: "business"})
```

Si noti come nell'esempio sopra le query 2. e 3. differiscano dalla query 1. , infatti nella prima query andiamo ad effettuare un controllo di uguaglianza con un valore unico, mentre nelle altre due il tag « `business` » si trova inserito all'interno di una lista, per cui il controllo sarà effettuato sugli elementi della lista e basterà trovare un un elemento della lista che corrisponda al valore che stiamo cercando.

Ci piacerebbe andare a memorizzare anche le *coordinate* di una posizione, in modo tale da andare in seguito a ricercare locations vicine ad alcune coordinate.

Esempio: Locations v2 - Implementazione di un semplice sistema di coordinate

```JSON
location = {
   name: "10gen East Coast",
   address: "134 5th Avenue 3rd Floor",
   city: "New York",
   zip: "10011",

   tags: ["business", "offices"],
   latlong: [40.0, 72.0]
}
```

Per quanto noi siamo consapevoli che il campo `latlong` corrisponda a delle coordinate, quel campo per MongoDB non è altro che una lista. Anche se MongoDB è nativamente un sistema **schema-less**, si rende a volte necessario aggiungere degli schemi parziali per garantire più efficienza. A questo scopo vengono creati degli **indici**:

```JavaScript
db.locations.ensureIndex({latlong: "2d"})
```

Questo comando ci permette non solo di rendere le nostre query più efficienti, ma anche di andare a forzare il fatto che i valori per il campo `latlong` siano bidimensionali (" `2d` ").

Per andare ora ad effettuare query che ci permettano di ottenere location vicine a delle certe coordinate possiamo andare ad utilizzare gli **operatori spaziali**:

```JavaScript
db.locations.find({latlong:{$near[40,70]}})
```

È comunque importante menzionare il fatto che per quanto sia possibile andare a memorizzare informazioni spaziali, MongoDB non è sicuramente il sistema più consono a questo scopo. Esistono infatti soluzioni più efficienti e studiate proprio per questo caso d'uso.

Ipotizziamo ora di voler aggiungere la possibilità per gli utenti di aggiungere delle *note* e dei *commenti* su ogni location.

> Esempio: Locations v3 - Aggiunta la possibilità di lasciare commenti

```JSON
location = {
  name: "10gen East Coast",
  address: "134 5th Avenue 3rd Floor",
  city: "New York",
  zip: "10011",

  tags: ["business", "offices"],
  latlong: [40.0, 72.0],
  tips: [ // lista di oggetti complessi
    {
      user: "nosh", date: "6/26/2010",
      tip: "stop by for office hours on Thursdays",
    },
    {...},
  ]
}
```

Ipotizzando che la v3 sia la versione completa che ci serve per la collection **locations**, andiamo avedere quali sono gli indici che ci sarà necessario definire per avere più efficienza:

```JavaScript
db.locations.ensureIndex({tags:1})
db.locations.ensureIndex({name:1})
db.locations.ensureIndex({latlong:"2d"})
```

Quando andiamo a creare un indice su una lista, questo verrà creato su ogni elemento della lista. Assieme alle possibilità già viste per effettuare query è anche disponibile la funzionalità delle regular expression:

```JavaScript
db.locations.find({name:^typeaheadstring/})
```

Andiamo ora a vedere come sfruttare le operazioni CRUD viste in precedenza applicate a questo contesto. Per andare ad inserire gli elementi nella collection utilizziamo il comando `insert`:

```JavaScript
db.locations.insert(location) // per definizione di location
di vedano gli esempi (v3 in particolare)
```

Per andare a modificare una specifica location andiamo ad utilizzare il comando `update`, specificando una query e come andremo a modificare tale documento; in questo caso andremo ad aggiungere un tip, ipotizzando che questo non fosse già presente:

```JavaScript
db.locations.update(
  {name: "10gen HQ"}, // query
  // push  è usato per aggiungere elementi ad una lista (tips)
  {$push: {tips:
      {
        user: "nosh", date: "2/26/2010",
        tip: "stop by for office hours on Thursdays",
      }
    }}
  }
)
```

Rappresentazione dei check-in

Per andare a rappresentare i vari check-in degli utenti abbiamo a disposizione varie scelte:

- Possiamo scegliere come con i vari tips, di associarli alla collection delle locations,memorizzando per ogni location una lista di check-in
- Possiamo andare a creare una collection `user` all'interno della quale memorizzeremo per ogni utente i check-in da questo effettuati
- Possiamo utilizzare una nuova collection specifica per i check-in che verrà gestita allo stesso modo di come trattiamo una relazione **many-to-many**

La scelta dell'approccio da utilizzare dipende più che altro dall'utilizzo che andremo a fare dei dati: in particolare, in questo caso, dal tipo di statistiche che vogliamo estrarre (o che vogliamo estrarre più frequentemente rispetto alle altre):

- Ci potrebbe interessare capire per ogni utente quale luogo è stato più frequentato, in questo caso forse è meglio salvare i check-in nella collection degli utenti
- Ci potrebbe interessare capire quale è location più frequentata tra tutte, e in questo caso sarebbe utile avere i check-in come attributo delle locations
- Nel caso in cui abbiamo bisogno di entrambe le statistiche, probabilmente sarebbe il caso di utilizzare una collection separata per i soli check-in

Utenti con check-in

Andiamo a mostrare come sia possibile mostrare i check-in come proprietà di un utente. La modalità non dovrebbe soprendere dal momento che il meccanismo è analoga quello per i *tips* nella collection delle *location*.

```JSON
user = {
  name: "nosh",
  email: "nosh@10gen.com",
  ... // altre proprietà dell'utente
  checkins: [
    {
      location: "10gen HQ",
      timestamp: "9/20/2010, 10:12:00",
```

```
        ... // altre proprietà
    },
    ... // altri check-in dell'utente
  ]
}
```

Per andare ad estrarre delle statistiche è possibile utilizzare le seguenti query:

```javascript
// estrazione di tutti gli utenti che hanno effettuato un check-in in una location
db.users.find({"checkins.location": "10gen HQ"})

// estrae i 10 utenti che hanno effettuato più check-in in una location
db.users.find({"checkins.location": "10gen HQ"}).sort({ts:-1}).limit(10)

// estrae quanti utenti hanno effettuato un check-in in una location dopo un certo timestamp
db.users.find({
  "checkins.location": "10gen HQ",
  timestamp: {$gt: ...$}}
).count()
```

Evidentemente è ancora possibile calcolare statistiche riguardo alle specifiche location, ma è necessario in questo caso andare a scorrere tutti i record della collection `users`, risultando potenzialmente inefficiente nel caso in cui ogni utente abbia effettuato molti check-in.

Rappresentazione separata dei check-in

Possiamo scegliere di gestire separatamente la collection dei vari check-in, per fare ciò andremo a salvare un campo `checkins` all'interno dei record `user` che sarà costituito da una serie di references ai record della collection `checkins`

```json
user = {
  name: "nosh",
  email: "nosh@10gen.com",
  ... // altre proprietà
  checkins = [e4af242f, cfeb950a, a542e63e]
}
```

L'utilizzo di `ObjectID` ci consente di avere accesso in lettura molto efficiente, tuttavia nel caso in cui avessimo bisogno di proprietà di utenti e locazioni che non sono presenti all'interno dei record della collection `checkins` in quel caso tali attributi dovrebbero essere replicati al loro interno, in modo da evitare di dover eseguire operazioni di aggregazione. Questo potrebbe risultare problematico, specialmente nel caso in cui sia necessario eliminare dei dati, infatti duplicando le informazioni, sarebbe complicato capire cosa andare ad eliminare e dove andarlo a fare.

2.2.4. Operazioni di Aggregazione in MongoDB

All'interno di MongoDB le operazioni di aggregazione funzionano in maniera molto diversa da come funzionano in un database relazionale.

Pipeline di Aggregazione

All'interno di MongoDB abbiamo a disposizione una **pipeline di aggregazione**.

Esempio: Semplice pipeline di aggregazione in MongoDB

```javascript
db.orders.aggregate([
  // filtraggio
  {$match: {status: "A"}},
  // raggruppamento
  {$group: { id: "$cust id", total: {$sum: "$amount"}}},
])
```

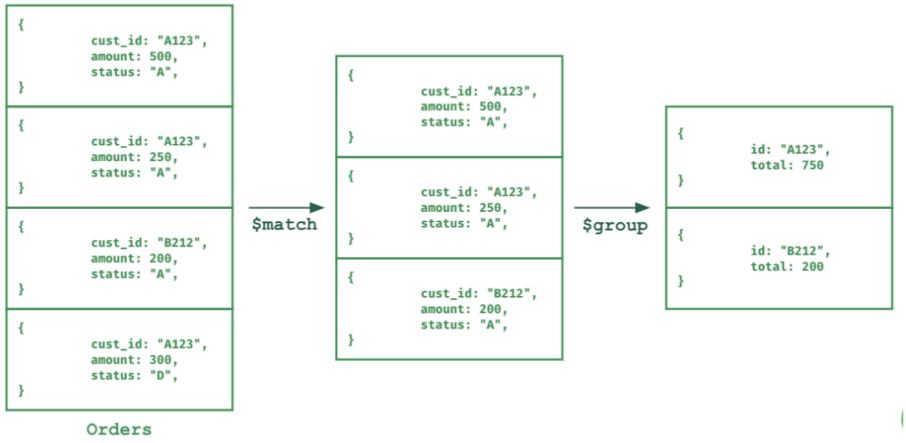

Figura 2.8: Rappresentazione grafica della pipeline di aggregazione specificata nel codice sopra

Nell'esempio di Figura 2.8 si nota come i passaggi principali di cui questa è costituita sono due: **matching** e **grouping**. Tuttavia queste non sono le uniche operazioni che è possibile effettuare in una pipeline di aggregazione. Andiamo di seguito a mostrare varie operazione assieme ad una breve descrizione:

- `match` : filtra i documenti in ingresso in base a una certa condizione
- `group` : raggruppa i documenti in sulla base di attributi comuni e calcola valori aggregati per ogni gruppo
- `project` : consente di modificare la struttura dei documenti in ingresso in vari modi, ad esempio modificando i nomi degli attributi, creando nuovi attributi o eliminando attributi esistenti
- `sort` : ordina i documenti in ingresso sulla base dei criteri che vengono specificati
- `limit` : limita il numero di documenti in uscita a un certo numero
- `skip` : salta un certo numero di documenti in ingresso
- `unwind` : consente di "esplodere" il contenuto di una lista, ottenendo un documento per ogni elemento della lista. Si usa tipicamente per andare a filtrare o raggruppare in

base a valori che si trovano all'interno di liste che non sarebbero altrimenti accessibili in maniera diretta
- **`geonear`**: consente di effettuare un'operazione molto simila all'operatore `$near`, ma all'interno di una pipeline di aggregazione e con maggiore flessibilità (più parametri possono essere specificati)

Come visto nell'esempio di sopra, all'interno dell'operazione di **grouping** è possibile utilizzare varie funzioni di aggregazione per calcolare valori aggregati sui gruppi creati. Le funzioni più comuni sono:

- `$sum`: somma i valori di un certo attributo
- `$avg`: calcola la media dei valori di un certo attributo
- `$min`: calcola il valore minimo di un certo attributo
- `$max`: calcola il valore massimo di un certo attributo
- `$push`: crea una lista con tutti i valori di un certo attributo

> Esempio: Pipeline di aggregazione su una collection 'sales'

JSON
```
[
  { item: "apple", qty: 10, store: "A", region: "north"},
  { item: "apple", qty: 5, store: "B", region: "south"},
  { item: "banana", qty: 7, store: "A", region: "north"},
  { item: "banana", qty: 8, store: "B", region: "south"},
  { item: "banana", qty: 3, store: "C", region: "north"}
]
```

Possiamo applicare la seguente pipeline di aggregazione:

JavaScript
```
db.sales.aggregate([
  // filtriamo i vari documenti
  { $match: { region: "north" },  // filtra per regione 'north'
    qty: {$gte: 5},               // e quantità maggiore o uguale a 5
  },

  {
    $group: {
      _id: "$item", // raggruppa per item
      totalQty: { $sum: "$qty" }, // somma le quantità per ogni item
      stores: { $push: "$store" }, // crea una lista di store per
      ogni item
      }
    }
])
```

Il risultato di questa pipeline sarà il seguente:

```json
[
  { "_id": "apple", "totalQty": 10, "stores": ["A"] },
  { "_id": "banana", "totalQty": 10, "stores": ["A", "C"] }
]
```

È possibile andare ad applicare un'operazione nominata **aggregazione "by window"**, della quale vediamo un esempio.

Esempio: Aggregazione 'by window'

```javascript
db.cakeSales.aggregate([
  $setWindowFields: {   // inizio dell'operazione di windowing
    partitionBy: "$state", // partiziona per stato
    sortBy: {orderDate: 1}, // ordina per data
    output: { // definizione del campo di output
      cumulativeQuantityForState: {
        $sum: "$quantity",
        window: {
          documents: ["unbounded", "current"] // considera documenti dal primo a quello corrente (somma cumulativa)
        }
      }
    }
  }
])
```

Dall'esempio di sopra possiamo comprendere alcuni aspetti legati al **windowing**:

- `$setWindowFields` serve a permettere di calcolari funzioni *windowed* su ogni documento; permettendo di aggiungere o sostituire campi basati sui valori in una determinata finestra
- `$partitionBy` divide i documenti in gruppi separati su cui calcolare la funzione di finestra; in questo caso i calcoli saranno effettuati separatamente per ogni stato
- `$sum` calcola la somma dei valori del campo `quantity` all'interno della finestra specificata
- `window: {documents: ["unbounded", "current"]}` specifica che la finestra deve includere tutti i documenti dal primo fino a quello corrente, permettendo così di calcolare una **somma cumulativa**

Oltre all'aggregazione "by window" è possibile effettuare un'altra tipologia di aggregazione, detta "**bucket aggregation**". Di seguito ne vediamo un esempio.

Esempio: Bucket Aggregation

```JavaScript
{
  $bucket: {
    groupBy: <expression>, // espressione su cui basare il raggruppamento
    boundaries: [ <lowerbound1>, <lowerbound2>, ... ], // definizione dei confini dei bucket
    default: <literal>, // bucket di default per valori fuori dai confini
    output: { // definizione dei campi di output per ogni bucket
    <field1>: { <accumulator1> : <expression1>  },
    ...
    <fieldN>: { <accumulatorN> : <expressionN> }
  }
}
```

Per quanto le varie aggregazioni presentate possano risultare concettualmente simili è importante capire quali siano le differenze tra di esse. Andremo perciò a mostrarle nella seguente tabella.

- **`group`**: raggruppa secondo un valore discreto, calcolando un documento per gruppo
- **`bucket`**: raggruppa per intervalli di valori, calcolando un documento per intervallo (bucket) specificato
- **`window`**: calcola valori cumulativi o su finestre mobili, calcolando un documento per ogni documento in ingresso; la peculiarità in questo caso è che si rende necessario specificare un ordine sui documenti in ingresso

Vediamo di seguito alcuni ulteri esempi di operazioni di aggregazione in MongoDB.

Esempio: Aggregazione 1

```JavaScript
db.things.aggregate([
  { $group: {_id: $parity$, sum: {$sum: $value} }}
])

> { "result" : [
    { "_id": "even", "sum": 102 },
    { "_id": "odd", "sum": 97 }
  ]
}
```

Esempio: Aggregazione 2

```javascript
db.zipcodes.aggregate([
  { $group: {
    _id:  "$state",
    totalPop: { $sum: "$pop" },
  }},
  {
    $match: { totalPop: { $gte: 10^6 } }
  }
])
```

Esempio: Aggregazione 3

La seguente query consente di trovare per ogni stato la città più grande e la più piccola in termini di popolazione:

```javascript
db.zipcodes.aggregate([
  { $group: {_id: {"$state", city: "$city"}, pop: {$sum: "$pop"}}},
  { $sort: {pop: 1}},
  { $group: {
    _id: "$_id.state",
    biggestCity: {$last: "$_id.city"},
    biggestPop: {$last: "$pop"},
    smallestCity: {$first: "$_id.city"},
    smallestPop: {$first: "$pop"}
  }},
  { $project: {
    _id: 0, // non mostrare il campo _id
    state: "$_id",
    biggestCity: {name: "$biggestCity", population: "$biggestPop"},
    smallestCity: {name: "$smallestCity", population: "$smallestPop"},
  }}
])
```

Esempio: Aggregazione 4

La query seguente mostra il funzionamente dell'operatore `$geonear` all'interno di una pipeline di aggregazione:

```javascript
db.places.aggregate([
  {
    $geoNear: {
      near: {type: "Point", coordinates: [ -73.9667, 40.78 ]},
```

```
5         distanceField: "dist.calculated",
6         maxDistance: 2,
7         query: { type: "public" },  // filtra per tipo 'public'
8         includeLocs: "dist.location",
9         spherical: true
10      }
11    }, // ... altri step della catena
12  ])
13
14  > {
15    "_id": 8,
16    "name": "Sara D. Roosevelt Park",
17    "type": "public",
18    "location": {
19      "type": "Point", "coordinates": [ -73.9935, 40.7186 ]
20    },
21    "dist": {
22      "calculated": 1.8259649934237,
23      "location": {
24        "type": "Point", "coordinates": [ -73.9935, 40.7186 ]
25      }
26    }
27  }
```

Map Reduce in MongoDB

MongoDB supporta nativamente l'utilizzo di Map-Reduce per effettuare operazioni di aggregazione sui dati memorizzati. Di seguito viene mostrata la sintassi per effettuare questa operazione.

```
1  db.collection.mapReduce(                                    JavaScript
2     <mapFunction>,    // funzione di mapping
3     <reduceFunction>, // funzione di riduzione
4     {
5       out: <collection>, // nome della collection di output
6     }
7  )
```

Vediamo di seguito un esempio di utilizzo di questo comando per replicare i risultati della pipeline di aggregazione in Figura 2.8.

Esempio: Algoritmo MapReduce in MongoDB

```
1  db.orders.mapReduce(                                        JavaScript
```

```
2    function() {emit(this.cust_id, this.amount);}, // map function
3    function(key, values) {return Array.sum(values);}, // reduce function
4    {
5      query: {status: "A"}, // filtro per status 'A'
6      out: "order_totals"   // output nella collection 'order_totals'
7    }
8  )
```

In generale l'algoritmo di MapReduce è estremamente flessibile e potente, tuttavia presenta alcuni svantaggi:

- Le performance sono generalmente inferiori rispetto all'utilizzo della pipeline di aggregazione, specialmente per operazioni semplici
- La scrittura delle funzioni di mapping e riduzione richiede l'utilizzo di JavaScript, il che può risultare scomodo per chi non ha familiarità con questo linguaggio
- La manutenzione del codice può risultare più complessa rispetto all'utilizzo della pipeline di aggregazione, specialmente per operazioni complesse

Alcune evoluzioni di MongoDB

A partire dalla versione 3.2 di MongoDB sono state introdotte alcune funzionalità che vanno a migliorare le capacità di aggregazione del sistema. In particolare è stata introdotta la possibilità di utilizzare un **left-outer join**. L'utilizzo di questo operatore è possibile all'interno delle pipeline di aggregazione combinato a tutti gli altri operatori già visti e viene utilizzato tramite il comando `lookup`. Il comportamento che ci attendiamo da questo operatore è visibile in Figura 2.9.

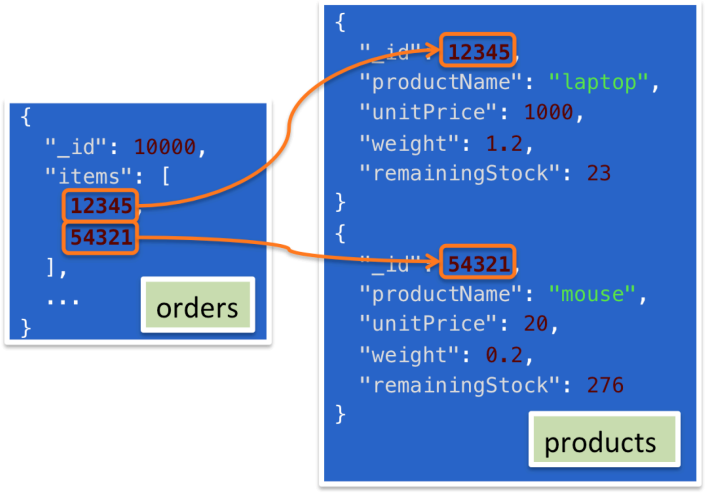

Figura 2.9: Esempio di utilizzo dell'operatore `$lookup` per effettuare un left-outer join

Di seguito andiamo a mostrare la sintassi del comando

```JavaScript
{
  $lookup: {
    from: <collection to join>,
    localField: <field from the input documents>,
    foreignField: <field from the documents of the "from" collection>,
    as: <output array field>
  }
}
```

A partire da MongoDB 3.4 e 3.6 sono state introdotte ulteriori funzionalità:

- `graphLookup`: consente di eseguire ricerche ricorsive all'interno di una gerarchia di documenti, permettendo di esplorare relazioni complesse tra i dati (**aggregazione ricorsiva**)
- **lookup** con condizioni di join multiple
- **views**: permettono di creare viste virtuali basate su query di aggregazione, consentendo di presentare i dati in modi diversi senza duplicarli fisicamente

Altre importanti funzionalità introdotte nelle versioni successive includono:

- **Transazioni multi-documento**
- **Transazioni distribuite**: consentono di eseguire transazioni che coinvolgono più shard in un cluster distribuito
- **Views materializzate**: viste che memorizzano fisicamente i risultati di una query di aggregazione per migliorare le performance
- **Unione di Pipeline**: consente di combinare i risultati di più pipeline di aggregazione in un'unica pipeline
- **Funzioni di aggregazione personalizzate**: permette di definire funzioni di aggregazione personalizzate per operazioni specifiche non coperte dalle funzioni predefinite
- **Supporto alle time series**: ottimizzazioni specifiche per la gestione di dati temporali, come serie storiche di misurazioni o eventi

2.2.5. Todo Maybe: forse torneremo qui per paralre di sharded deployments

3. 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.

3.1. Architettura di un R-DBMS

Quello che vediamo in Figura 3.10 è una rappresentazione dell'organizzazione interna delle componenti di un DBMS relazionale.

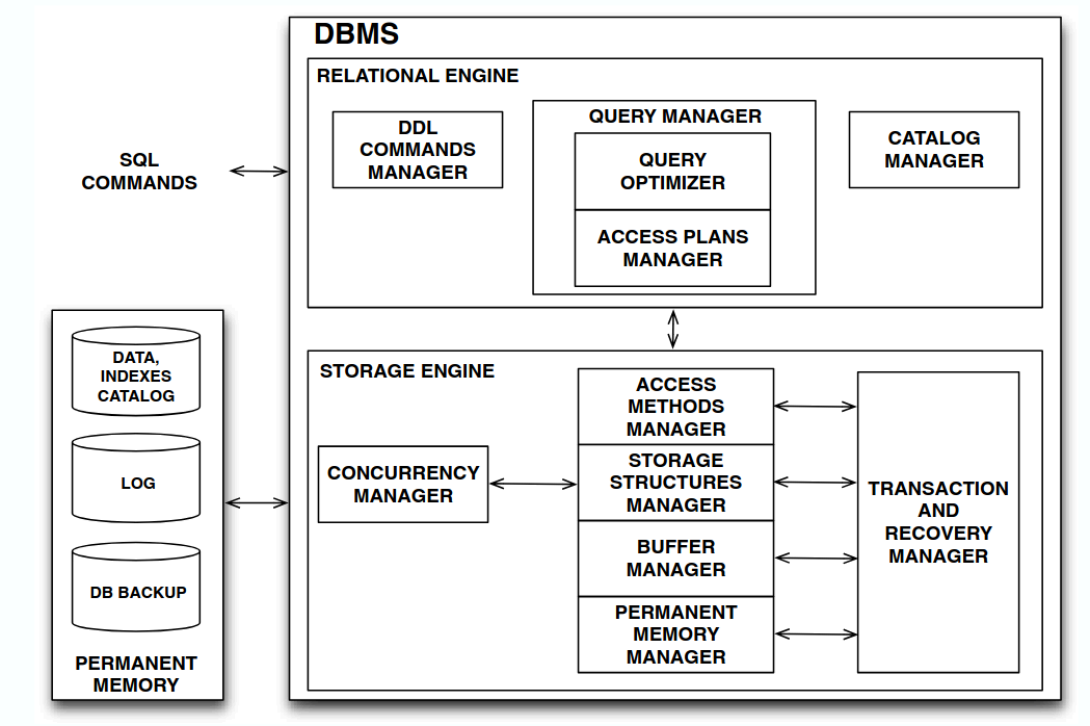

Figura 3.10: Organizzazione interna delle componenti di un R-DBMS

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.

3.2. 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 memorizzata 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.

3.2.1. 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 ❌

3.2.2. 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)`. Figura 3.11 mostra un esempio del funzionamento di questo algoritmo.

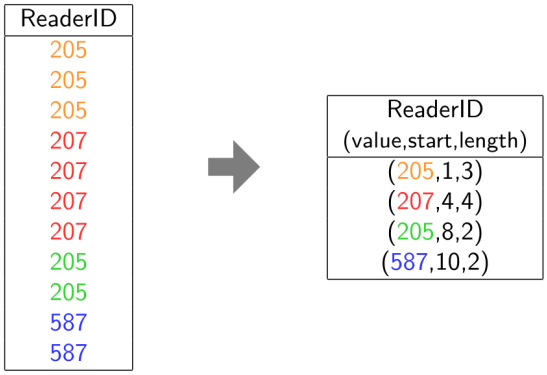

Figura 3.11: Esempio di Run-Length Encoding

> **Osservazione**
>
> 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. Figura 3.12 mostra un esempio del funzionamento di questo algoritmo.

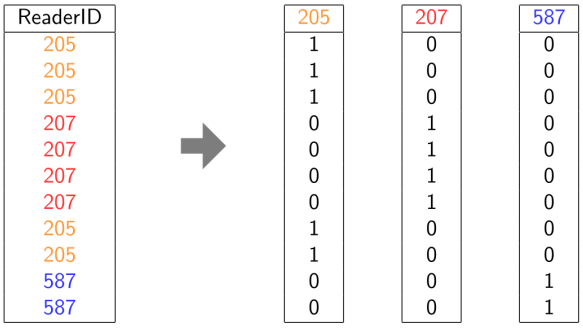

Figura 3.12: Esempio di Bit-vector encoding

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 ⟶ 205, 2 ⟶ 207 e 3 ⟶ 587 otteniamo la rappresentazione di Figura 3.13.

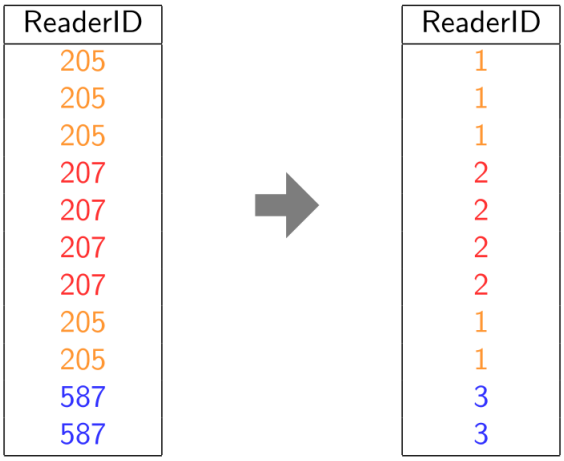

Figura 3.13: Esempio di dictionary-encoding

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 ⟶ 1002, 1010, 1004, 2 ⟶ 1008, 1006 otteniamo la rappresentazione di Figura 3.14

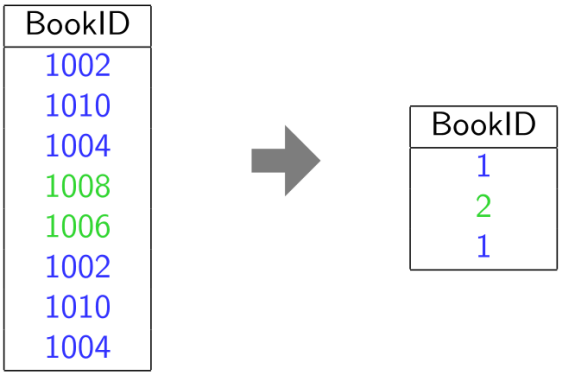

Figura 3.14: Esempio di dictionary-encoding per sequenze

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). Figura 3.15 mostra un esempio del funzionamento di questo algoritmo.

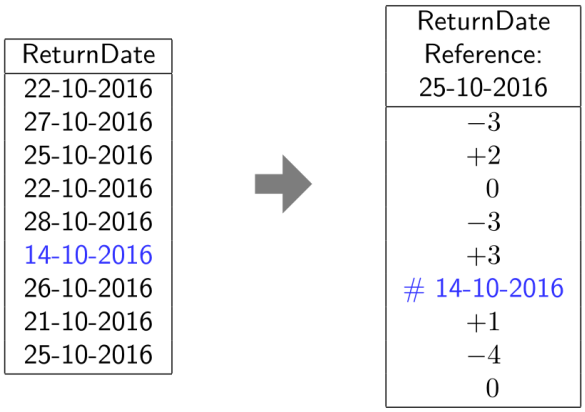

Figura 3.15: Esempio di Frame-of-Reference encoding con valori oltre l'offset massimo

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. Figura 3.16 mostra un esempio del funzionamento di questo algoritmo.

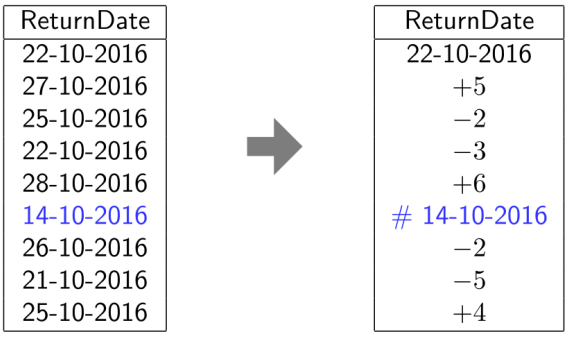

Figura 3.16: Esempio di encoding differenziale con valori oltre l'offset massimo

> **Osservazione**
>
> Si noti come in Figura 3.16 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.

> **Esempio: 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:
>
> $$\{(205, 4), (207, 2)\}$$

> **Osservazione**
>
> 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.

3.2.3. WiredTiger

4. Extensible Record Stores

Indice delle figure